Progress®



Progress® Data Objects:
Guide and Reference

OpenEdge®

Copyright

© **2019 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.**

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Deliver More Than Expected, Icenium, Kendo UI, Kinvey, NativeScript, OpenEdge, Powered by Progress, Progress, Progress Software Developers Network, Rollbase, SequeLink, Sitefinity (and Design), Sitefinity, SpeedScript, Stylus Studio, TeamPulse, Telerik, Telerik (and Design), Test Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. Analytics360, AppServer, BusinessEdge, DataDirect Autonomous REST Connector, DataDirect Spy, SupportLink, DevCraft, Fiddler, JustAssembly, JustDecompile, JustMock, Kinvey Chat, NativeChat, NativeScript Sidekick, OpenAccess, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

February 2019

Last updated with new content: Release 12.0

Updated: 2019/03/01

Table of Contents

Chapter 1: Overview of Progress Data Objects, Services, and Catalogs.11

Run-time architecture and data access....12
App development options....14
Progress Kendo UI support....14
NativeScript support....15
Other UI support....16

Chapter 2: Using the JSDO dialect of the Kendo UI DataSource 17

Features of the JSDO dialect....18
Sample Kendo UI DataSource instantiations using the JSDO dialect....21
Sample instantiation for an OpenEdge Data Object Service....21
Sample instantiation for a Rollbase Data Object Service....26

Chapter 3: Using the Progress Data Source for NativeScript and Angular....29

Chapter 4: Using the JSDO and the Data Source Modules in an Existing NativeScript App....35

Chapter 5: Using JSDOs to create mobile and web clients....37

What is new in JSDO 5.0?....38
JSDO overview....39
JSDO classes and objects....39
How a JSDO maps to a Data Object resource....40
How JSDO memory works....41
Methods of the JSDO and JSRecord classes....45
Asynchronous and synchronous execution....49
Properties of a JSDO....56
Requirements for using a JSDO....57
Creating and managing access to a JSDO instance....59
Accessing standard CRUD and Submit operations....61
Read operation example....64
Create operation example....68
Update operation example....72
Delete operation example....78

Submit operation example....83
Accessing custom Invoke operations....89
Asynchronous vs. synchronous method execution....90
Invoke operation example....90
Managing JSDO login sessions....95
Requirements for creating a JSDO login session....96
Using default web pages to support client app login....99
Handling changes in session online status....99
Using protected web resources....102
Dynamic data binding for alternate client platforms....103
Using the UIHelper class....104
Using a custom template....106

Chapter 6: JSDO class and object reference....109

progress.data.JSDO class....110
progress.data.JSDOSession class....124
progress.data.JSRecord class....133
progress.data.PluginManager class....135
progress.data.Session class....136
progress.ui.UIHelper class....142
request object....146

Chapter 7: JSDO properties, methods, and events reference....149

acceptChanges( ) method....153
acceptRowChanges( ) method....156
add( ) method....158
addCatalog( ) method (JSDOSession class)....159
addCatalog( ) method (Session class)....164
addItem( ) method....168
addLocalRecords( ) method....169
addPlugin( ) method....171
addRecords( ) method....175
afterAddCatalog event....179
afterCreate event....181
afterDelete event....183
afterFill event....185
afterInvoke event....186
afterLogin event....188
afterLogout event....190
afterSaveChanges event....192
afterUpdate event....195
assign( ) method (JSDO class)....198
assign( ) method (UIHelper class)....199

async property....200
authenticationModel property (JSDOSession class)....201
authenticationModel property (Session class)....201
autoApplyChanges property....202
autoSort property....203
batch property....205
beforeCreate event....206
beforeDelete event....207
beforeFill event....208
beforeInvoke event....209
beforeSaveChanges event....210
beforeUpdate event....211
caseSensitive property....213
catalogURIs property....215
clearItems( ) method....215
clientContextId property....216
connected property....216
data property....217
deleteLocal( ) method....218
display( ) method....219
fill( ) method....220
find( ) method....227
findById( ) method....229
fnName property....230
foreach( ) method....231
getData( ) method....233
getErrors( ) method....233
getErrorString( ) method....239
getFormFields( ) method....240
getFormRecord( ) method....241
getId( ) method....242
getListViewRecord( ) method....243
getPlugin( ) method....244
getProperties( ) method....245
getProperty( ) method....246
getSchema( ) method....247
getSession( ) stand-alone function....248
hasChanges( ) method....253
hasData( ) method....255
invalidate( ) method....256
invalidateAllSessions( ) stand-alone function....258
invocation method....262
invoke( ) method....264
isAuthorized( ) method....268
jsdo property....272

JSDOs property....272
jsrecord property....272
jsrecords property....273
lastSessionXHR property....274
login( ) method (JSDOSession class)....275
login( ) method (Session class)....280
loginHttpStatus property....284
loginResult property....285
loginTarget property....285
logout( ) method (JSDOSession class)....286
logout( ) method (Session class)....288
name property (JSDO class)....290
name property (JSDOSession class)....291
objParam property....291
offline event....292
online event....294
onOpenRequest property....295
ping( ) method (JSDOSession class)....297
ping( ) method (Session class)....300
pingInterval property....303
readLocal( ) method....304
record property....305
rejectChanges( ) method....305
rejectRowChanges( ) method....308
remove( ) method....311
response property....312
saveChanges( ) method....314
saveLocal( ) method....330
services property....331
serviceURI property....333
setDetailPage( ) method....333
setFieldTemplate( ) method....334
setItemTemplate( ) method....335
setListView( ) method....336
setProperties( ) method....337
setProperty( ) method....338
setSortFields( ) method....339
setSortFn( ) method....341
showListView( ) method....343
sort( ) method....344
subscribe( ) method (JSDO class)....347
subscribe( ) method (JSDOSession class)....348
subscribe( ) method (Session class)....349
success property....350
table reference property (JSDO class)....351

table reference property (UIHelper class)....353
unsubscribe( ) method (JSDO class)....353
unsubscribe( ) method (JSDOSession class)....355
unsubscribe( ) method (Session class)....356
unsubscribeAll( ) method....356
useRelationships property....357
userName property....358
xhr property....358

Chapter 8: Data type mappings for Data Object Services....361

JavaScript data type overview....361
OpenEdge ABL to JavaScript data type mappings....363
Rollbase object to JavaScript data type mappings....365

1

Overview of Progress Data Objects, Services, and Catalogs

Progress® Data Objects rely on standard web technologies to allow a client application running on the Internet to access data resources from supported Progress Data Object Services made available using web servers. These Data Object Services employ web technologies to support a broad range of data access options, depending on the resource, from read/write access of simple sets of data elements to direct invocation of business logic on the server that can control complex transactions over that data. Currently, the supported data resources include those that can be provided by OpenEdge® application servers and Rollbase® servers running in a public or private cloud. The supported clients include both mobile and web apps built using JavaScript, HTML5, and jQuery-based UI libraries, depending on the tools you use to build these apps.

Note that Progress Data Objects represent the Progress-supported implementation of the Cloud Data Object, which is an open source project on GitHub. This documentation is a guide and reference to the Progress Data Objects implementation only. For more information on the Cloud Data Object project itself, see http://clouddataobject.github.io/.

The current Progress Data Objects release supports two basic app development options: developing mobile apps using the Telerik® Platform and developing both mobile and web apps using other development tools.

For details, see the following topics:

- Run-time architecture and data access
- App development options

Run-time architecture and data access

A *Progress Data Object Service* is a web service built using a supported transport (currently, REST) and a Data Service Catalog. This service runs in a web application on a web server. The *Progress Data Service Catalog* is a JSON file that describes the schema for one or more Data Object resources and the operations that can be invoked on these resources by the client. A Data Object *resource* represents a set of data on a server that can be accessed by the client using specific operations supported by that resource. This set of data can include either a single table or multiple tables containing one or more records of data. A Data Object *operation* reads or writes resource data, or otherwise executes business logic associated with the resource, in a single network request. The possible schema and operations for a given resource depends on the type of its Data Object Service, which can be (currently) either OpenEdge or Rollbase.

A *Progress Data Object* provides access to a single resource provided by a Data Object Service and consists of both a server and client Data Object that exchange resource data over the web as JSON. The server Data Object is an object that implements the schema of the resource and its supported set of operations. The client Data Object consists of a set of JavaScript classes and objects known together as the *JavaScript Data Object* (JSDO). The classes and objects of the JSDO allow the client app to create one or more JSDO instances to access one or more Data Object resources provided by a Data Object Service. A single JSDO instance provides access to a single resource using a login session (*JSDO login session*) that the JSDO establishes in the web application hosting the Data Object Service that provides its resource.

The classes and objects of the JSDO include:

- **progress.data.JSDO class on page 110** — Allows you to create and manage a JSDO instance for a given Data Object resource and to execute supported operations on that resource. These operations are accessed as JSDO methods that hide all the details of the JSON encoding/decoding and REST protocols used to invoke these operations over the web.
- **progress.data.JSDOSession class on page 124** — Allows you to asynchronously create a JSDO login session in the web application that hosts a given Data Object Service and to access the Data Service Catalog, which is typically hosted by the same web application.
- **progress.data.JSRecord class on page 133** — Represents a single record of data associated with a Data Object resource that is stored in the memory of the JSDO instance.
- **progress.data.Session class on page 136** — As an alternative to the `JSDOSession` class, allows you to either synchronously or asynchronously create a JSDO login session in the web application that hosts a given Data Service and to access the Data Service Catalog, which is typically hosted by the same web application.
- **progress.ui.UIHelper class on page 142** — Allows you to dynamically bind JSDO data to a JavaScript UI using certain supported UI frameworks.
- **request object on page 146** — Returns the results of a single operation on the server Data Object and its resource.

In order to create a JSDO instance, the client app first reads the Data Service Catalog in order to configure the instance with the schema and operations that are specifically supported by the specified Data Object resource. Note, again, that a given Data Object Service and its Catalog can support one or more single-table and multi-table resources from which the client can create JSDO instances. The types of resources that a Data Object Service can support depends on the capabilities of the data server that the Data Service is created to access. You can then create a given JSDO instance to access any particular single-table or multi-table resource that the specified Data Object Service supports.

The types of resources and operations that are available in this release of Progress Data Object Services include those supported by:

- **An OpenEdge application server** — Single-table resources based on Advanced Business Language (*ABL*) *temp-tables* or single-table and multi-table resources based on ABL datasets (*ProDataSets*). In addition to one or more temp-tables, a ProDataSet resource can also support OpenEdge before-imaging, as does the JSDO instance that accesses it. Each resource is implemented on the server by an OpenEdge *Business Entity*, which is an ABL procedure or class-based object (the server Data Object) that encapsulates the resource schema and its supported operations. A JSDO instance created for an OpenEdge Data Object Service can therefore provide access to either a single temp-table resource and its operations or a single ProDataSet resource and its operations.

  The Data Object operations on OpenEdge resources can include any or all of *Create*, *Read*, *Update*, and *Delete* (CRUD), *Invoke* (which allows the client to directly invoke ABL business logic on the application server), and for ProDataSet resources that support before-imaging, *Submit*, which can batch multiple CUD operations with complex transaction management. Any JSDO that accesses an OpenEdge resource can accept or reject the results of individual CUD operations based on record change errors returned from the server. If it uses a Submit operation to access a ProDataSet resource that supports before-imaging, the JSDO can accept or reject all record changes involved in a single transaction that is synchronized with the server.

- **A Rollbase server** — Single-table resources based on *Rollbase objects*. A Rollbase object thus implements the server Data Object that encapsulates the schema and supported operations for a given single-table resource. The available Rollbase object resources provided by a given Rollbase Data Object Service are those in a Rollbase view from which the Data Object Service is created. A JSDO instance created for a Rollbase Data Object Service can therefore provide access to a single Rollbase object resource and its operations only.

  The Data Object operations on Rollbase resources can include *Create*, *Read*, *Update*, and *Delete* (CRUD), and *Invoke* (which allows a client to access Rollbase objects that are related to the single Rollbase object resource for which the JSDO instance is created). Note that Rollbase resources do not support before-imaging or Submit operations. A JSDO instance that accesses a Rollbase resource can accept or reject individual CUD operations based on each record-change error returned from the server.

A JSDO supports a few basic methods for preparing and invoking supported operations on Data Object resources. Once you have established a JSDO login session and created a JSDO instance to access a given Data Object resource, you can load the JSDO (*JSDO memory*) with data from the resource, modify the data in JSDO memory, update the resource with these JSDO memory modifications, and perform other business logic using the following basic JSDO methods:

- **`fill( )`** — Calls a Read operation on the resource to initialize and load data (in the form of *record objects*) into JSDO memory.
- **`add( )`, `assign( )`, or `remove( )`** — A basic method that creates, updates, or deletes (respectively) an individual record object in JSDO memory.
- **`saveChanges( )`** — Depending on the resource, the record objects currently changed in JSDO memory, and how you invoke it, either this method individually calls one or more Create, Update, and Delete (CUD) operations on a single record object of the resource, with each operation sent to the server one record-change request at a time, or this method calls a single Submit operation that updates the resource with all the current record object changes in JSDO memory in a single request to the server. If the resource supports before-imaging, the results of multiple record object changes from a Submit operation can also be handled as a batch in JSDO memory. If the resource does not support before-imaging, the results of any single CUD or Submit operation can only be handled one record object at a time in JSDO memory.
- **Invocation method** — A custom-named JSDO method that calls an Invoke operation and returns the results of a corresponding routine that executes on the server. A JSDO can have as many invocation methods as routines that are supported by the resource for remote access on the server. For OpenEdge resources, the names of these methods are determined by the ABL Business Entity developer, who maps each invocation method to an ABL routine that the method remotely calls on the Business Entity. For Rollbase object resources, the names of these methods are automatically generated by Rollbase for any invocation methods that need to be defined for access by a JSDO. Rollbase maps each such invocation method to a Rollbase

server method that reads and returns the data from a given Rollbase object that is related to each Rollbase object resource. For all resources of a given Progress Data Object Service, the mappings for invocation methods are defined in the Data Service Catalog. Note that any data returned from an invocation method is not automatically added to JSDO memory, but can be used either to update existing record objects in JSDO memory, or to add new record objects using the JSDO `add( )` or `addRecords( )` method.

In addition, the JSDO provides numerous properties, methods, and events that support the execution and management of these basic methods for modifying the data in JSDO memory and invoking resource operations. For more information, see Using JSDOs to create mobile and web clients on page 37.

App development options

Before you can build and test client apps (web or mobile) to access Progress Data Objects using JSDO technology, you must obtain access to one or more Progress Data Object Services, depending on the app development option that you plan to use:

- **OpenEdge Data Object Services** — Create one or more ABL Business Entities to expose OpenEdge temp-tables and ProDataSets as Data Object resources in an OpenEdge Data Object Service. Then, create the Data Object Service to generate and deploy a Data Service Catalog for these Data Object resources. For more information on creating ABL Business Entities to implement Data Objects, see the sections on Data Object Services in *OpenEdge Development: Web Services*.

  For information on creating and deploying Data Object Services to OpenEdge application servers and accessing the Data Service Catalog for a given Data Object Service, see the topics on Data Objects and Data Object Services in the *Progress Developer Studio for OpenEdge Online Help*.

- **Rollbase Data Object Services** — Generate a Data Service Catalog with selected Rollbase objects as its Data Object resources. For more information on generating a Data Service Catalog to access Rollbase data from the Telerik Platform, see the Rollbase documentation topic, Creating Progress Data Catalogs for use in the Telerik Platform.

The current release of Progress Data Objects support two options for developing mobile and web apps with access to Progress Data Object Services.

These include:

- **Developing mobile and web apps using Progress Kendo UI®** — See Progress Kendo UI support on page 14.
- **Developing HTML5 apps using other development tools** — See Other UI support on page 16.

Progress Kendo UI support

Progress Kendo UI is an HTML5 user interface framework for building interactive and high-performance mobile and web apps. This framework comes with a library of 70+ UI *widgets* (an abundance of data-visualization gadgets), a client-side data source, and a built-in MVVM (Model-View-ViewModel) library. Kendo UI also provides AngularJS and Bootstrap integration. For more information on Kendo UI, see:

- **Kendo UI** — Product Overview
- **Kendo UI widgets** — Bundle Support for Kendo UI Components

Access to Progress Data Object Services through the JSDO from Kendo UI is currently supported for:

- Web app development using the Kendo UI Builder by Progress on page 15

As you develop a web app to access Progress Data Object Services using Kendo UI, the Kendo UI Builder accesses the JSDO instance for a given Data Object resource using the *JSDO dialect of the Kendo UI DataSource*. A Kendo UI DataSource is an object that allows you to bind tabular data to Kendo UI widgets from a local or remote data service. In the Kendo UI Builder, you can specify access to Progress Data Objects as a remote data service using a *Progress Data Service*.

You must use the JSDO dialect of the Kendo UI DataSource to access any Progress Data Object resource using supported Kendo UI widgets. The Kendo UI Builder provides support for implementing this access without additional coding. You can thus use one or more Kendo UI DataSources to bind Progress Data Object resources to Kendo UI widgets like the Kendo UI Grid and various views that you build using it.

For an overview and reference to the JSDO dialect of the Kendo UI DataSource, see Using the JSDO dialect of the Kendo UI DataSource on page 17.

Although a Kendo UI DataSource generally manages all access to JSDO data bound to a Kendo UI widget, you might have the need to interact with a JSDO instance directly. In addition, a JSDO can work with OpenEdge ProDataSets and manage transactions where as the Kendo UI DataSource is limited to a single table. For more information, see Using JSDOs to create mobile and web clients on page 37

Kendo UI Builder by Progress

The Kendo UI® Builder by Progress® facilitates the modernization of existing Progress® OpenEdge® desktop business applications by moving the application user interface (UI) to the web using Kendo UI. These OpenEdge applications can be simple to more complex applications containing multiple, feature-specific modules. Applications that already conform to the OpenEdge Reference Architecture (OERA) are especially well suited for modernization using the Kendo UI Builder. However, you can modernize any OpenEdge application with its UI separated from its business logic running on an OpenEdge application server, which can be either an instance of Progress Application Server for OpenEdge or the classic Progress® OpenEdge® AppServer®.

Kendo UI Builder tooling supports the design and development of a modern and responsive web UI in the form of a deployable OpenEdge web app that accesses one or more ABL application services implemented as OpenEdge Data Object Services. This tooling supports UI upgrades for future versions of the initial web app over time, with little or no additional coding, using customizable templates and meta-data from which the deployable web app is generated.

Although, the Kendo UI Builder supports JSDO CRUD access for OpenEdge Data Object Services only, the tool also provides read-only access to other types of remote data services, such as REST and OData.

For more information on the Kendo UI Builder, see *Kendo UI Builder by Progress: Modernizing OpenEdge Applications*.

NativeScript support

NativeScript® is an open source framework for developing cross-platform, yet native, Android and iOS apps. NativeScript provides a platform to create Android and iOS apps using either JavaScript or TypeScript with Angular support and CSS. It renders UIs with the native platform's rendering engine, ensuring native-like performance and user experience (UX).

To know more about NativeScript, visit https://www.nativescript.org/.

You can create a NativeScript app using TypeScript and Angular to access a Data Object resource using Progress JSDO and Progress Data Source for NativeScript and Angular. TypeScript is the programming language used with this model.

To learn how to access a Data Object resource in a NativeScript app, visit https://wbt.progress.com/progress/nd/courses/Accessing-PDO-using-NativeScriptApp/.

Other UI support

To develop a mobile or web app using simple JavaScript code editors or other HTML and JavaScript IDE's, you can program a JSDO instance and its data as a data source for static HTML UI components. Minimally, you can write a little JavaScript code to map field references in JSDO memory to HTML elements so that data updates in the HTML elements are reflected in JSDO memory and data updates in JSDO memory are reflected in the HTML elements.

You also need to manage JSDO login sessions in order to load one or more Progress Data Service Catalogs and use the Data Object Services that they define to create and invoke operations on JSDO instances. Unlike using Kendo UI DataSources to manage all access to JSDO data for Kendo UI widgets, to access JSDO data that you want to bind to other types of UI elements and frameworks, you directly invoke methods on the JSDO to read and update resource data or to invoke other business logic on the server. For more information, see Using JSDOs to create mobile and web clients on page 37.

In order to work with JSDO instances in other development environments, you need to download the appropriate JSDO libraries. For more information, see the Cloud Data Object page on GitHub.

2

Using the JSDO dialect of the Kendo UI DataSource

To provide data to a Kendo UI widget, such as a Kendo UI Grid, you use a Kendo UI DataSource. This is a JavaScript class (`kendo.data.DataSource`) that allows you to define an instance that provides either local data values or data values from a remote data service of some type that you define using various configuration properties. The DataSource configuration for a remote data service primarily defines the transport options for reading and modifying the data and the schema of the data. For more information, see the Kendo UI documentation on the kendo.data.DataSource.

Although you can define a DataSource to access virtually any remote data service, the Telerik Kendo UI and Backend Services internally support different *dialects* of the Kendo UI DataSource, each of which supports specialized transport and schema definitions for accessing a given remote data service. Thus, the JavaScript Data Object (JSDO) dialect of the Kendo UI DataSource supports access to the JSDO and its associated Progress Data Object Service as a remote data service. In doing so, it takes advantage of the built-in data management features of the JSDO and its schema as defined by Data Service Catalog.

However, note that Kendo UI widgets have particular Kendo UI DataSource requirements, and not all of them support the JSDO dialect. The following table summarizes the Kendo UI widgets that support data access through a Kendo UI DataSource and if they support the JSDO dialect.

Table 1: Kendo UI support for DataSources

Widgets with a DataSource property	Support for the JSDO dialect?	Notes
`ComboBox`	Yes	–
`DropDownlist`	Yes	–

Widgets with a DataSource property	Support for the JSDO dialect?	Notes
`Gantt`	No	Requires `GanttDataSource`
`Grid`	Yes	–
`ListView`	Yes	–
`MultiSelect`	Yes	–
`PivotGrid`	No	Requires `PivotDataSource`
`Scheduler`	No	Requires `SchedulerDataSource`
`TreeList`	No	Requires `TreeListDataSource`
`TreeView`	No	Requires `HierarchicalDataSource`

Note: Any mobile versions of the widgets in this table are supported in exactly the same way, for example, the Kendo UI Mobile ListView.

For details, see the following topics:

- Features of the JSDO dialect
- Sample Kendo UI DataSource instantiations using the JSDO dialect

Features of the JSDO dialect

The JSDO dialect of the Kendo UI DataSource supports:

- **The Kendo UI Grid and other Kendo UI widgets** in much the same way as other dialects and custom instantiations of the Kendo UI DataSource. However, some standard DataSource property and method behavior is implemented based on a transport that defines a JSDO instance as its remote data service.
- **Access to a single-table JSDO instance or any single table from a multi-table JSDO instance**. You use a separate DataSource for each table of a shared multi-table JSDO in order to show master-detail relationships.
- **JSDO dialect transport properties: `jsdo`, `tableRef`, and `countFnName`**, which specify, respectively, the JSDO configured for the DataSource, the name of the JSDO table that the DataSource accesses (optional for single-table JSDOs), the name of the JSDO Invoke operation method that the DataSource must call to return the total number of records in the server result set for the JSDO table (required only for server paging).

  Note: In OpenEdge Release 11.6.3 or later, you can annotate a Count operation method in a Business Entity that you implement to return the total number of records. With this annotation, you do not need to specify the `countFnName` transport property for the corresponding Kendo UI DataSource. For more information, see the documentation on coding Business Entities for access by Kendo UI DataSources in your OpenEdge release.

- **Pre-configured DataSource CRUD operations** using the individual JSDO CRUD or Submit operations already defined for the JSDO instance—there is no need to configure each DataSource CRUD operation individually.
- **Initial values for Create operations**, as defined for the JSDO instance by the resource in the Data Service Catalog for which it is created.
- **The option to configure a DataSource to instantiate its own JSDO** for private, single-table access, as well as to share an existing JSDO instance with other DataSources.
- **Access to the JSDO instance directly** in DataSource event handlers or Promise deferred functions, for example, to call `addRecords( )`, the local storage APIs, or invoke methods on the JSDO.
- **The standard `sync( )` method** on the DataSource, which calls the JSDO `saveChanges( )` method, depending on a user's interaction with the connected Kendo UI widget.
- **The standard `batch` configuration property** on the DataSource, which indicates if the JSDO calls `saveChanges( )` to invoke a Data Object Service Submit operation (if available in the resource) to send a group of record changes across the network, or to invoke each Data Object Service Create, Update, or Delete operation individually across the network.
- **The standard `serverPaging`, `serverFiltering`, and `serverSorting` configuration properties** of the DataSource, which allow a remote data service to manage the paging, filtering, and sorting features (respectively) of Kendo UI. If any or all of these server properties are set to `true`, the JSDO passes the corresponding property settings for these features to the Data Object Service Read operation to be managed on the server. If any or all of these server properties are set to `false` (the default), the DataSource manages the corresponding features in its own local memory using the data that has already been returned by the Data Object Service Read operation.
- **The `progress.data.JSDOSession` class to create a JSDO login session** for the DataSource that is validated with either Anonymous, Basic or Form-based authentication over HTTP or HTTPS.
- **Additional JSDO dialect transport properties: `autoSave` and `readLocal`** (available in Progress Data Objects Version 4.3 or later), which indicate, respectively, 1) if the DataSource `sync( )` method automatically saves pending JSDO data updates to the server or leaves them unsaved in JSDO memory, and 2) if the DataSource `read( )` method always reads JSDO data from the server or from the data already loaded into JSDO memory, as follows:

  1. **`autoSave`** — When set to `true` (the default) and the DataSource `sync( )` method is called, `sync( )` invokes the JSDO `saveChanges( )` method, saving all changes pending for the DataSource table (and all other tables) in JSDO memory to the server. When `autoSave` is set to `false` and the DataSource `sync( )` method is called, `sync( )` leaves all changes pending for the tables in JSDO memory **without** saving them to the server.
  2. **`readLocal`** — When set to `false` (the default) and the DataSource `read( )` method is called, the DataSource reads and re-initializes JSDO memory with the specified server data for the DataSource table (and all other resource tables) by calling the JSDO `fill( )` method. When `readLocal` is set to `true` and the DataSource `read( )` method is called, the DataSource reads all data for its table **only** from the data already initialized in JSDO memory, including any pending changes since the most recent call to `fill( )`.

  While these properties can be used for a DataSource created for a table without any table relations, these properties are most useful for reading and updating related tables in hierarchical DataSources for the following supported use case: where two related tables (parent and child) reside in the same JSDO instance that you create for an OpenEdge ProDataSet resource, and you create a separate DataSource (parent and child) for each table in that same JSDO instance. You therefore:

1. Create the JSDO for a single ProDataSet resource with the parent and child tables.
2. Create the DataSource for each table, setting the `jsdo` transport property for each DataSource to reference the same JSDO instance that you have created.
3. Set the `tableRef` transport property for each DataSource to the name of its corresponding parent or child table.
4. Specify custom settings for both the `autoSave` and `readLocal` properties **only** in the transport of the child DataSource.

Thus, the parent DataSource transport uses the default values for `autoSave` and `readLocal`, while the child DataSource transport uses your custom settings, with `autoSave` set to `false` and `readLocal` set to `true`.

At this point, calling `read( )` on the parent DataSource initializes JSDO memory for both the parent and child tables, with the data loaded in the child table limited to records that are related to the data in the parent. Any changes made to the child table remain pending in JSDO memory, even if `sync( )` is called on the child DataSource. When `sync( )` is called on the parent DataSource, any changes made to both the parent and child tables are then saved to the server.

For more information on managing hierarchical DataSources for this supported use case using Kendo UI Grid widgets to access the parent and child DataSources, see the sample and white paper located at the following Cloud Data Object project page:
https://github.com/CloudDataObject/sample-Hierarchical_PDS_Template.

Note: The brief example described in Sample instantiation for an OpenEdge Data Object Service on page 21 of this document assumes that only a single DataSource is created to access the specified JSDO instance and its single table.

- **Additional JSDO dialect transport property: `useArrays`** (available in Progress Data Objects Version 4.3 or later), which when set to `false` (the default) tells the DataSource to treat each element of a resource table array field as a separate scalar field, with the name *`field-ref_integer`*, where *`field-ref`* is the name of the array field and *`integer`* is a one (1)-based integer that qualifies the array field name based on the order of its value in the original array. When set to `true`, the DataSource treats any array field as a standard JavaScript array, which requires additional coding to reference the value of each array element. For more information, see the table reference property (JSDO class) on page 351 description.

Note: Array fields are supported only for OpenEdge Data Object resource tables. An array field in OpenEdge can be defined for an ABL temp-table using the `EXTENT` option. For more information, see the description of the `DEFINE TEMP-TABLE` statement in *OpenEdge Development: ABL Reference*.

- **Error handling** is supported by an `error` event callback function that provides access to JSDO error information for any single Data Object Service CRUD or Submit operation on a DataSource that returns an error. To access this error information, you have the option of inspecting the callback object parameter, which provides some error information in the value of its `errorThrown` property and of its `xhr` property, depending on the type of error returned.

  As of Progress Data Objects Version 4.3 or later, you can also call the JSDO `getErrors( )` method on the JSDO table reference accessed by the DataSource for details of any error or errors returned from invoking a DataSource operation. Using this method, you have no need to inspect the `xhr` property of the callback object parameter. The error information returned by this method can include all types of errors from invoking a DataSource operation. The errors actually returned depend on the type and configuration of the Data Object resource that handles the operation on the server and the types of errors returned from a given operation request.

Sample Kendo UI DataSource instantiations using the JSDO dialect

Following are sample Kendo UI DataSource instantiations using the JSDO dialect for both OpenEdge and Rollbase Data Object Services. Specific configuration features of the JSDO dialect are highlighted in bold.

Note that the main difference between DataSources instantiated for OpenEdge and Rollbase Data Object Services is that DataSources for Rollbase Object Services do **not** support batch mode (setting the DataSource `batch` property to `true`). One result of this is that error handling for DataSource record changes is simpler when using a Rollbase Data Object Service. For more information, see the description of error handling for each sample DataSource instantiation.

Sample instantiation for an OpenEdge Data Object Service

The following DataSource is instantiated to access an OpenEdge ProDataSet resource that supports before-imaging and the Submit operation, `dsCustomerOrd`:

Table 2: Sample for an OpenEdge Data Object Service

```
myjsdo = new progress.data.JSDO({ name: 'dsCustomerOrd' });

var dataSource = new kendo.data.DataSource( {
    type: "jsdo",
    serverPaging: true,
    serverFiltering: true,
    filter: { field: "State", operator: "eq", value: "MA" },
    serverSorting: true,
    sort: { field: "State", dir: "desc" },
    transport: {
        jsdo: myjsdo,
        tableRef: "ttCustomer",
        countFnName: "count",
        autoSave: false,
        readLocal: true
    },
    batch: true,
    error: function(e) {
        var dataErrors, error = "", i, j;

        console.log('Error: ', e);
        if (e.errorThrown) {
            error = "\n" + e.errorThrown.message;
        }
        dataErrors = this.transport.jsdo.ttCustomer.getErrors();
        for (var i = 0; i < dataErrors.length; i++) { /* each error */
            error += "\n" + JSON.stringify(dataErrors[i]);
        }
        alert("JSDO error(s) returned: " + error);
        e.preventDefault();
    }
} );
```

Note that this example DataSource accesses a JSDO that is instantiated prior to instantiating the DataSource itself. This also assumes that a user login session has been created for the JSDO and a Data Service Catalog describing the `dsCustomerOrd` resource has already been added to the JSDO session. For more information on creating a user login session for a JSDO and adding a Catalog to the session, see progress.data.JSDOSession class on page 124.

Following is a brief description of the configuration options specified in the sample definition for the JSDO dialect of the Kendo UI DataSource shown above:

- **Setting the standard `type` property to `"jsdo"`** — Specifies that the Kendo UI DataSource is being instantiated for the JSDO dialect.
- **Setting the standard `serverPaging`, `serverFiltering`, or `serverSorting` properties to `true`** — If `server*` properties are `true`, the specified processing (paging, filtering, sorting) is performed on the server, and if `server*` properties are `false`, the specified processing is done locally by the Kendo UI DataSource.

  For example, if the `serverFiltering` property is `true`, the settings for its related `filter` property on the DataSource (or from any more recent invocation of the `filter( )` method on the DataSource) are passed to the resource's Read operation on the server to filter the data before it is returned to the client. If `serverFiltering` is `false`, the latest DataSource `filter` settings are used to filter the data that already populates the DataSource. The same is true of the remaining `server*` configuration properties and their related property settings, such as the settings of the `serverPaging` property and its related `pageSize` property, and the settings of the `serverSorting` property and its related `sort` property.

  Note: When any `server*` properties are `true`, the format of the related property values that the DataSource passes to the resource's Read operation depends on the Data Object Service type (OpenEdge or Rollbase). For more information, see the description of the JSDO fill( ) method on page 220.

  If all of these `server*` properties are `false`, the JSDO receives the records for the single (or specified) table of its resource, in whatever quantity and order they are received from the resource's most recent Read operation on the server, and passes them to the DataSource which then manages all paging, filtering, and sorting of the records in its own internal memory.

  Note: The JSDO dialect of the DataSource does **not** support setting the `serverGrouping` or `serverAggregates` configuration properties to `true` in this release. You should either leave these properties **not** set, or set them to `false`. That is, any data grouping and aggregate features of Kendo UI can only be managed by the Kendo UI DataSource itself. If you set any of these properties to `true` in this release, the respective feature is **ignored by both** the server and the JSDO dialect of the Kendo UI DataSource.

- **Setting the JSDO dialect `jsdo` transport property** — The `jsdo` property specifies the JSDO that is accessed by the DataSource. In the sample, an existing instance of the JSDO (`myjsdo`) is specified. You can also specify the name of a JSDO resource as a string, and the DataSource will create the JSDO instance for you. This is the same resource name you would use to create the JSDO yourself. In this sample, you would specify the `jsdo` value as `"dsCustomerOrd"`.
- **Setting the JSDO dialect `tableRef` transport property** — Required only for a JSDO created for a resource that is a multi-table ProDataSet, the `tableRef` property specifies the name of the JSDO table whose data the DataSource handles. So, for an OpenEdge ProDataSet, you specify the name of the corresponding temp-table in the Business Entity (`ttCustomer` in the sample). If you want to access the data in all tables in the ProDataSet, you need to create a separate Kendo UI DataSource for each table accessed by the JSDO and attach it to a Kendo UI widget appropriately.
- **Setting the JSDO dialect `countFnName` transport property** — The `countFnName` property specifies the name of the Invoke operation method in the Business Entity that returns the total number of records in

the server result set to the Kendo UI DataSource when using server paging (`serverPaging == true`). For more information on implementing this method, see the sections on updating Business Entities for access by Telerik DataSources in *OpenEdge Development: Web Services* from the latest release of OpenEdge. Note that for a JSDO that reads data from a multi-table ProDataSet, this Invoke operation method only returns the total number of records in the result set (in all pages) of the parent table.

Note: In OpenEdge Release 11.6.3 or later, you can annotate a Count operation method in a Business Entity that you implement to return the total number of records. With this annotation, you do not need to specify the `countFnName` transport property for the corresponding Kendo UI DataSource. For more information, see the documentation on coding Business Entities for access by Kendo UI DataSources in your OpenEdge release.

- **Setting the JSDO dialect `autoSave` and `readLocal` transport properties** — These settings are most useful if the `tableRef` property specifies the name of a child table that is related to a parent table residing in the same JSDO instance specified by the `jsdo` property. For more information, see the description of these properties in Features of the JSDO dialect on page 18.
- **Setting the standard `batch` property** — Specifies how record changes are processed when the `sync( )` method is called on the DataSource, depending on the `batch` property setting (`false` by default). For this release of the JSDO dialect of the DataSource, if `batch == true`, the JSDO resource **must** define a Data Object Service Submit operation. The following table shows how the DataSource responds to the setting of its `batch` property, depending on whether a Submit operation is defined in the JSDO:

Table 3: JSDO dialect response to the batch property setting on the DataSource

DataSource `batch` property setting	Does JSDO define a Submit operation?	DataSource response when saving changes
`false`	No	The transport gets a change for a single record. This record change is sent to the server one record at a time, using the corresponding Data Object Service Create, Update, or Delete operation (by calling `saveChanges(false)` on the JSDO).
`true`	No	Not supported in this release. The DataSource throws an exception if saving changes to the server with `batch == true` when the JSDO does **not** define a Submit operation.
`false`	Yes	The transport gets a change for a single record. This record change is sent to the server one record at a time, using the Data Object Service Submit operation (by calling `saveChanges(true)` on the JSDO).
`true`	Yes	The transport gets changes for one or more records in an array. All the changes are grouped by the JSDO and sent to the server using the Data Object Service Submit operation (by calling `saveChanges(true)` on the JSDO).

Note: For a standard Kendo UI DataSource with `batch == true`, multiple record changes are grouped together so that all record changes of a given type (create, update, or destroy) are sent to the server at one time by the DataSource transport. However in this release, the transport for the JSDO dialect can **only** send the entire set of record changes (all create, update, and destroy changes together) in a single Data Object Service Submit operation.

- **Handling the parameter (`e`) and associated JSDO errors returned by the DataSource `error` event handler function** (Updated for Progress Data Objects Version 4.3 or later) — The `error` event handler function is called when an error occurs while performing CRUD or Submit operations on a remote data service (the JSDO in this case). An error can thus be returned when invoking the `read( )` method or the `sync( )` method on the DataSource, which in this case invokes a Submit operation on any changed data in the JSDO.

  As shown in Table 2: Sample for an OpenEdge Data Object Service on page 21, the `errorThrown` property of the event parameter (`e`) is set, along with some additional properties, depending on the types of errors returned. To process these errors for a JSDO, you can look at the following:

  - `e.errorThrown.message`
  - The result of calling `getErrors( )` on the JSDO table reference that the DataSource is accessing

  If an error occurs after invoking the DataSource `sync( )` method when saving changes in the JSDO to the server, the following generic message is typically returned in `e.errorThrown.message`: `"JSDO: Error while saving changes."`. This message is returned if one or more errors occurs while saving JSDO changes, regardless of the `batch` property setting on the DataSource. So for example, if `batch == true`, and multiple errors occur for a single Data Object Service Submit operation, this one error message is returned for all of them.

  Additional errors from calls to the JSDO dialect `read( )` or `sync( )` method can be returned in other property settings of the `e` parameter, such as `e.xhr`, as well as from the JSDO table itself, depending on the type of error. However, all of these errors can be returned using a single call to the JSDO `getErrors( )` method on the JSDO table reference.

  This JSDO method thus returns different types of errors that result from calling the:

  - JSDO dialect `read( )` method, which ultimately calls the JSDO `fill( )` method to invoke a Progress Data Object Read operation. These include errors from executing the Read operation itself.
  - JSDO dialect `sync( )` method, which ultimately calls the JSDO `saveChanges( )` method to invoke Progress Data Object CUD or Submit operations. These include errors from executing the operations themselves. They, in turn, can include different types of errors depending on if the Data Object resource supports before-imaging or not.

    Note: Submit operations must always process data with before-image support. Note also that even when the DataSource invokes `sync( )` for one record change at a time (with `batch == false`), if the JSDO defines a Submit operation, the DataSource invokes a separate Submit operation for each record that is changed.

  - JSDO dialect `read( )` or `sync( )` method regardless of the Data Object operation. These include errors that originate from the network, web server, or web application that hosts the Data Object Service, or otherwise result from the operation being inaccessible.

The error handler in the sample handles all these types of errors. The JSDO `getErrors( )` method is invoked on the JSDO table (`ttCustomer`) that the DataSource references and returns an array of objects, each of which contains properties describing an error depending on the error type. The method call stores this array in a variable (`dataErrors`), as shown in the following fragment from the previous sample:

```
myjsdo = new progress.data.JSDO({ name: 'dsCustomerOrd' });

var dataSource = new kendo.data.DataSource( {
    type: "jsdo",
    . . .
    transport: {
        jsdo: myjsdo,
        tableRef: "ttCustomer", . . .
    },
    batch: true,
    error: function(e) {
        var dataErrors, error = "", i, j;

        console.log('Error: ', e);
        if (e.errorThrown) {
            error = "\n" + e.errorThrown.message;
        }
        dataErrors = this.transport.jsdo.ttCustomer.getErrors();
        for (var i = 0; i < dataErrors.length; i++) { /* each error */
            error += "\n" + JSON.stringify(dataErrors[i]);
        }
        alert("JSDO error(s) returned: " + error);
        e.preventDefault();
    }
} );
```

Note: In the sample, `this` refers to the Kendo UI DataSource, allowing access to its properties, methods, and events.

The sample then loops through the `dataErrors` array to log the contents of each error object as a string using the standard `JSON.stringify( )` method. The properties that can be returned for each `getErrors( )` error object include:

- **`errNum`** — An error number that appears only in errors from operation routines running on the server.
- **`error`** — A string containing an error message, depending on the error type (`type` property value).
- **`id`** — The internal ID of any single record object associated with the error. This value can be used to return the record and its current field values by calling the `findById( )` method on the JSDO table reference.
- **`responseText`** — A string that can contain additional error information for general network and server errors, regardless of the operation.
- **`type`** — Identifies the type of error, indicated by a numeric constant.

Some errors returned by `getErrors( )` are exclusive of other errors, while others can be returned as part of a series of errors, depending on the operations and the data for which the errors are generated. For more information on these error object properties and how they can be returned by `getErrors( )`, see the getErrors( ) method on page 233 description.

So, when the sample `error` function executes, it:

1. Logs the entire contents of `e` to the console log.
2. Creates a multi-line error string (`error`) starting with the contents of `e.errorThrown.message`.

3. Concatenates one or more error lines to the multi-line error string from the contents of each error object returned by `getErrors( )`.
4. Displays an alert box showing the contents of the multi-line error string.

Note: An OpenEdge date/time value can be stored using one of three possible OpenEdge data types that contain, respectively, a date only, a date and time only, or a date, time, and time zone. The JSDO stores each date/time value as a JavaScript `string` formatted according to the capabilities of the original OpenEdge data type. However, the JSDO dialect of the Kendo UI DataSource converts and works with these JSDO date/time values as JavaScript `Date` objects, which maintain complete date, time, and time zone information for every JSDO date/time value, regardless of its original OpenEdge data type. To enforce a consistent exchange of data/time information between JavaScript `Date` objects and OpenEdge date/time data types, the JSDO therefore automatically follows specific rules when exchanging date/time values between JavaScript `Date` objects and `string` fields that represent a specific OpenEdge data/time data type. For more information, see OpenEdge ABL to JavaScript data type mappings on page 363.

Sample instantiation for a Rollbase Data Object Service

The following DataSource is instantiated to access a Rollbase object resource, `Customers`:

Table 4: Sample for a Rollbase Data Object Service

```
myjsdo = new progress.data.JSDO({ name: 'Customers' });

var dataSource = new kendo.data.DataSource( {
    type: "jsdo",
    serverPaging: true,
    serverFiltering: true,
    filter: { field: "State", operator: "eq", value: "MA" },
    serverSorting: true,
    sort: { field: "State", dir: "desc" },
    transport: {
        jsdo: myjsdo,
        /* tableRef: "Customers", */
        countFnName: "count",
    },
    error: function(e) {
        var dataErrors, error = "", i, j;

        console.log('Error: ', e);
        if (e.errorThrown) {
            error = "\n" + e.errorThrown.message;
        }
        dataErrors = this.transport.jsdo.ttCustomer.getErrors();
        for (var i = 0; i < dataErrors.length; i++) { /* each error */
            error += "\n" + JSON.stringify(dataErrors[i]);
        }
        alert("JSDO error(s) returned: " + error);
        e.preventDefault();
    }
} );
```

Note that this example DataSource accesses a JSDO that is instantiated prior to instantiating the DataSource itself. This also assumes that a user login session has been created for the JSDO and a Data Service Catalog describing the `Customers` resource has already been added to the JSDO session. For more information on creating a user login session for a JSDO and adding a Catalog to the session, see progress.data.JSDOSession class on page 124.

Following is a brief description of the configuration options specified in the sample definition for the JSDO dialect of the Kendo UI DataSource shown above:

- **Setting the standard `type` property to `"jsdo"`** — Specifies that the Kendo UI DataSource is being instantiated for the JSDO dialect.
- **Setting the standard `serverPaging`, `serverFiltering`, or `serverSorting` properties to `true`** — If `server*` properties are `true`, the specified processing (paging, filtering, sorting) is performed on the server, and if `server*` properties are `false`, the specified processing is done locally by the Kendo UI DataSource.

  For example, if the `serverFiltering` property is `true`, the settings for its related `filter` property on the DataSource (or from any more recent invocation of the `filter( )` method on the DataSource) are passed to the resource's Read operation on the server to filter the data before it is returned to the client. If `serverFiltering` is `false`, the latest DataSource `filter` settings are used to filter the data that already populates the DataSource. The same is true of the remaining `server*` configuration properties and their related property settings, such as the settings of the `serverPaging` property and its related `pageSize` property, and the settings of the `serverSorting` property and its related `sort` property.

  Note: When any `server*` properties are `true`, the format of the related property values that the DataSource passes to the resource's Read operation depends on the Data Object Service type (OpenEdge or Rollbase). For more information, see the description of the JSDO fill( ) method on page 220.

  If all of these `server*` properties are `false`, the JSDO receives all the records for the single table of the Rollbase resource, in whatever order they are received from the resource's most recent Read operation on the server, and passes them to the DataSource which then manages all paging, filtering, and sorting of the records in its own internal memory.

  Note: The JSDO dialect of the DataSource does **not** support setting the `serverGrouping` or `serverAggregates` configuration properties to `true` in this release. You should either leave these properties **not** set, or set them to `false`. That is, any data grouping and aggregate features of Kendo UI can only be managed by the Kendo UI DataSource itself. If you set any of these properties to `true` in this release, the respective feature is **ignored by both** the server and the JSDO dialect of the Kendo UI DataSource.

- **Setting the JSDO dialect `jsdo` transport property** — The `jsdo` property specifies the JSDO that is accessed by the DataSource. In the sample, an existing instance of the JSDO (`myjsdo`) is specified. You can also specify the name of a JSDO resource as a string, and the DataSource will create its own instance of the JSDO for you. This is the same resource name you would use to create the JSDO yourself. In this sample, you would specify the `jsdo` value as `"Customers"`.
- **Setting the JSDO dialect `tableRef` transport property** — Required only for a JSDO created for a multi-table resource, the `tableRef` property specifies the name of the JSDO table whose data the DataSource handles. So, for a Rollbase object, which is a single-table resource, you can specify the name of the resource itself (`Customers` in the sample) or leave it unspecified (or commented out as in the sample). Note that to access other Rollbase object resources provided by the Data Object Service, you need to create a separate Kendo UI DataSource and corresponding JSDO for each Rollbase object, and attach it to a Kendo UI widget appropriately.
- **Setting the JSDO dialect `countFnName` transport property** — The `countFnName` property specifies the name of the server JavaScript method that returns the total number of records in the server result to the Kendo UI DataSource when using server paging (`serverPaging == true`). Note that for a Rollbase Data Object Service, the name of this method is always `"count"`, as shown in the sample. Although it always takes the same value, you must always specify the value for `countFnName`, because the DataSource does not know that its JSDO is connected to a Rollbase or an OpenEdge Data Object Service.

- **Setting the standard `batch` property is not supported** — Note that Rollbase Data Object Services do not support setting the DataSource `batch` property to `true`. (The sample uses the `batch` property's default value of `false`.) The Data Object Service Create, Update, and Delete operations process the data only for a single record change in each network request to a Rollbase object resource.

- **Handling the parameter (`e`) and associated JSDO errors returned by the DataSource `error` event handler function** (Updated for Progress Data Objects Version 4.3 or later) — The `error` event handler function is called when an error occurs while performing CRUD operations on a remote data service (the JSDO, in this case). An error can thus be returned when invoking the `read( )` method or the `sync( )` method on the DataSource.

  As shown in Table 4: Sample for a Rollbase Data Object Service on page 26, the `errorThrown` property of the event parameter (`e`) is set, along with some additional properties, depending on the types of errors returned. To process these errors for a JSDO, you can look at the following:

  - `e.errorThrown.message`
  - The result of calling `getErrors( )` on the JSDO table reference that the DataSource is accessing

  If an error occurs when invoking the DataSource `sync( )` method when saving changes in the JSDO to the server, the following generic message is typically returned in `e.errorThrown.message`: `"JSDO: Error while saving changes."`. This message is returned if one or more errors occurs while saving JSDO changes.

  Additional errors from calls to the JSDO dialect `read( )` or `sync( )` method can be returned using a single call to the JSDO `getErrors( )` method on the JSDO table reference. This method can return an array of error objects (`dataErrors` in the sample), with each object containing properties depending on the type of error:

  - **`errNum`** — An error number that appears only in errors from operation routines running on the server.
  - **`error`** — A string containing an error message, depending on the error type (`type` property value).
  - **`id`** — The internal ID of any single record object associated with the error. This value can be used to return the record and its current field values by calling the `findById( )` method on the JSDO table reference.
  - **`responseText`** — A string that can contain additional error information for general network and server errors, regardless of the operation.
  - **`type`** — Identifies the type of error, indicated by a numeric constant.

  The sample loops through the `dataErrors` array to log the contents of each error object as a string using the standard `JSON.stringify( )` method. However, for each CRUD operation on a Rollbase object resource, the `getErrors( )` method typically returns only a single error object containing a `type` and `error` property. In some cases, a `responseText` property might also be returned with additional information for a network or Rollbase server error. For more information on the possible output from `getErrors( )`, see the getErrors( ) method on page 233 description.

  So, when the sample `error` function executes, it:

  1. Logs the entire contents of `e` to the console log.
  2. Creates a multi-line error string starting with the contents of `e.errorThrown.message`.
  3. Concatenates one or more error lines to the multi-line error string for each error returned by `getErrors( )` (typically only one additional error line for a Rollbase resource error).
  4. Displays an alert box showing the contents of the multi-line error string.

3

Using the Progress Data Source for NativeScript and Angular

Description

The Progress Data Source is an npm module that provides seamless integration between client apps built with the Angular framework (that is, NativeScript and Angular web apps) and a Progress Data Object Service. It is available as an npm package on http://www.npmjs.com, which you can download and use in your NativeScript app or your web app.

The Data Source module allows you to access data as well as to perform CRUD operations on a data resource from a supported Data Object Service. It is a TypeScript implementation. Therefore, along with JavaScript code, a declaration file is also provided in the package. It defines the module's types and function signatures.

The declaration file declares two classes:

- **DataSource** — This provides the integration layer to access the remote data resource. (Internally, it supports access to the JSDO.)
- **DataSourceOptions** — This provides options to a Data Source instance.

DataSourceOptions Properties

Property	Type	Description
countFnName?	string	• The name of the remote invoke operation in the BusinessEntity that returns the total number of records in the data set. • Used when server paging is performed.

Property	Type	Description

filter?	any	• Use it to specify client-side filtering of data.
		• Similar to the filter property of the `kendo.data.DataSource` class.
		Example:
		`filter: {`
		`field: "State",`
		`operator: "eq",`
		`value: "MA"`
		`}`

jsdo	progress.data.JSDO	• This property is required. • It specifies the JSDO instance, which provides access to the resources of a Progress Data Object Service. • All other properties of the DataSourceOptions class are optional.
mergeMode?	number	An integer that represents a merge mode to use.Possible values are: • progress.data.JSDO.MODE_APPEND • progress.data.JSDO.MODE_MERGE • progress.data.JSDO.MODE_REPLACE The default value is **MODE_MERGE**.
readLocal?	boolean	• If true, the read method does not perform a read operation on the remote service (if there is existing local data), but just returns the local data. • If false (the default), the read method allows access to data on the remote service. • It is useful to set readLocal to true for a child Data Source, when you develop a hierarchical app, for example.
skip?	number	Use it to specify how many records in the result set must be skipped before a page of data is returned.

Property	Type	Description

sort?	any	• Use this object to specify sorting of data.
		• Similar to the sort property of the `kendo.data.DataSource` class.
		Example:
		`sort: {`
		`field: "State",`
		`dir: "desc"`
		`},`

tableRef?	string	It refers to the temp table that is to be accessed as a resource. **Note:** This property is required only if the specified jsdo represents a multi-table DataSet.
top?	number	Use it to specify the number of records to be returned from the result set.

DataSource Functions

Table 5: DataSource Functions

Function/API	Description	Input	Output

read(params?: progress.data.FilterOptions): Observable<DataResult>	It asynchronously retrieves data from the remote data resource. (Internally, it calls the jsdo.fill() method.)	• progress.data.FilterOptions:
		`interface FilterOptions {`
		`filter?: any;`
		`id?: any;`
		`skip?: number;`
		`sort?: any;`
		`top?: number;`
		`}`
		• The data retrieved is based on the specified filter property.
		• If none of the optional parameters is specified, then all the data is retrieved.
			• Returns an Observable.
			• If it succeeds, the observer's next() function returns an array of record objects along with the total number of record objects returned.
			• For a failure, the observer's error() function returns an Error object containing an error message.

Function/API	Description	Input	Output
getData(): <Array<object>>	It returns an array of record objects that currently resides in local memory	NA	Returns an array of record objects.
create(data: object): object	It creates a new record in local memory.	data: An object that contains the details of the new record.	• Returns the new record on success. • On failure, an exception is thrown.
findById(id: string): object	It returns a copy of the record with the specified ID.	id **Note:** The existing implementation of the function uses JSDO's internal _id as the id property.	• Returns the record with the specified ID. • On failure, it returns null.
update(data: any): boolean	It updates a record in local memory.	data: An object that contains the updated details for the record.	• Returns true on success. • Else returns false.
remove(data: any): boolean	It deletes a record from local memory.	data: An object with a valid id of the record to be deleted.	• Returns true if successful. • Else returns false.
hasCUDSupport(): boolean	It indicates whether CUD (Create, Update and Delete) support is available in the underlying JSDO.	NA	• Returns true if CUD support is available. • Else returns false.

Function/API	Description	Input	Output
hasSubmitSupport(): boolean	It indicates whether the underlying JSDO has Submit support (that is, provides a Submit operation).	NA	• Returns true if the corresponding JSDO has Submit support. • Otherwise, returns false.
saveChanges(): Observable<Array<object>>	It synchronizes the Data Object service with all the record changes (CUD) pending in local memory. It is an asynchronous call.	NA	• Returns an Observable. • If it succeeds, the observer's next() function returns an object containing all changed rows. • For a failure, the observer's error() function returns an Error object containing an error message.

Example

The following example illustrates the usage of some Data Source APIs in a NativeScript Master Detail starter template (`tns-template-master-detail-progress-ng`) that is meant to access the Data Object resource, Customer.

To use the DataSource and the DataSourceOptions classes in a NativeScript app, first import the classes in your service file (`Customer.Service.ts`, which takes care of retrieving and updating data), as shown below:

```
import { DataResult, DataSource, DataSourceOptions } from "@progress/jsdo-nativescript";
```

Next, add the imported DataSource class as a property and then instantiate it:

```
export class CustomerService {
    private dataSource: DataSource;
    createSession(callback) {
        progress.data.getSession({
            serviceURI,
            catalogURI,
            authenticationModel: 'anonymous'
        }).then(() => {
            this.dataSource = new DataSource({
                jsdo: new progress.data.JSDO({
                    name: 'Customer'
                })
            });
            this.readCustomers(callback);
        }, () => {
            console.log('Error while creating session.');
        });
    }
```

As you can see, 'Anonymous' is configured as the authentication model.

Then, you can use the Data Source APIs as per your needs. For example, the following lines of code implement the read API to read data from the Customer resource:

```
readCustomers(callback) {
        this.dataSource.read({
            filter: {
                field: 'State',
                operator: 'eq',
                value: 'HI'
            }
        }).subscribe((myData: DataResult) => {
            callback(myData.data);
        });
    }
```

Again, to create a new record in the backend Customer resource, the following code snippet is useful:

```
dataSource.create(<object for new record>)
    dataSource.saveChanges().subscribe((myData) => {
      // Code for successful operation
  }, (error: Error) => {
      // Code to handler error
);
```

The following lines show how you can implement the update API:

```
dataSource.update(<object for existing record - includes _id property>)
    dataSource.saveChanges().subscribe((myData) => {
      // Code for successful operation
  }, (error: Error) => {
      // Code to handler error
);
```

Finally, the remove API can be used as follows:

```
dataSource.remove(<object for existing record - includes _id property>)
    dataSource.saveChanges().subscribe((myData) => {
      // Code for successful operation
  }, (error: Error) => {
      // Code to handler error
);
```

4

Using the JSDO and the Data Source Modules in an Existing NativeScript App

High-level tasks:

1. Install the `@progress/jsdo-nativescript` npm package in your NativeScript project by running the command, `npm install @progress/jsdo-nativescript`.
2. Import the `@progress/jsdo-core` and `@progress/jsdo-nativescript` packages into the app's code-behind file.
3. In the code-behind file:
   a. Add code to create a JSDOSession instance.
   b. Add code to create a Data Source instance.
   c. Modify the app's primary component.html file to display the new data.
   d. Save your changes.
   e. Run the app on an emulator.

The following example shows the changes that need to be made to the **Cars** app (generated using the `tns-template-master-detail-ng` template) to use the JSDO and the Data Source modules:

1. Install `@progress/jsdo-nativescript`.
2. In **\app\cars\shared\car.service.ts**:
   - Add the following import statements.

```
import { progress } from "@progress/jsdo-core";
import { DataSource, DataSourceOptions, DataResult } from
"@progress/jsdo-nativescript";
```

```
import { Observable, from } from "rxjs";
import { catchError } from "rxjs/operators";
```

- Comment out the following lines of code.

```
load(): Observable<any> {
        return new Observable((observer: any) => {
            const path = "cars";

            const onValueEvent = (snapshot: any) => {
                this._ngZone.run(() => {
                    const results = this.handleSnapshot(snapshot.value);
                    observer.next(results);
                });
            };
            firebase.addValueEventListener(onValueEvent, `/${path}`);
        }).catch(this.handleErrors);
}
```

- Add the following lines of code.

```
private dataSource: DataSource;
    createSession(successFn, errorFn): void {
            const serviceURI =
"https://oemobiledemo.progress.com/OEMobileDemoServices";

            progress.data.getSession({
                serviceURI,
                catalogURI: serviceURI + "/static/SportsService.json",
                authenticationModel: "anonymous"
            }).then((object) => {
                this.dataSource = new DataSource({
                    jsdo: new progress.data.JSDO({ name: "Customer" })
                });
                successFn();
            }, () => errorFn());
    }

load(): Observable<any> {
        const promise = new Promise((resolve, reject) => {
            this.createSession(() => {
                this.dataSource.read().subscribe((myData: DataResult) => {
                        resolve(myData.data);
                    });
                }, (error) => {
                  const message = (error && error.message) ? error.message : "Error
 reading records.";
                    reject(new Error(message));
                });
        });
        return from(promise).pipe(catchError(this.handleErrors));
}
```

3. In **\app\cars\car-list.component.html**, change **car.name** to **car.Name**.

```
<Label [text]="car.Name" class="text-primary font-weight-bold"></Label>
```

4. Save the changes.

As a result of these changes, when you launch the Cars app on the emulator, you will see that the app is connected to OEMobileDemoServices and that it displays customer names instead of car names.

5

Using JSDOs to create mobile and web clients

As described in Overview of Progress Data Objects, Services, and Catalogs on page 11, Progress Data Objects support the development of mobile and web apps by providing access to OpenEdge and Rollbase data that you obtain from server-side Data Objects using corresponding client-side Data Objects, which are instances of the JavaScript Data Object (JSDO). With the help of additional JavaScript classes and objects, JSDO instances access these data resources through Data Object Services running on an appropriate OpenEdge or Rollbase server. For a reference to the basic JavaScript classes and objects that support JSDO access, see the JSDO class and object reference on page 109 and JSDO properties, methods, and events reference on page 149.

The following topics describe how to use the features of a JSDO and its associated JavaScript objects to access Data Object Services, regardless of the mobile or web app development platform.

For details, see the following topics:

- What is new in JSDO 5.0?
- JSDO overview
- Creating and managing access to a JSDO instance
- Accessing standard CRUD and Submit operations
- Accessing custom Invoke operations
- Managing JSDO login sessions
- Dynamic data binding for alternate client platforms

What is new in JSDO 5.0?

This section briefly describes what is new in JSDO 5.0.

ES6 Promises

Promises, like callback functions, provide a mechanism to handle asynchronous operations. Prior to JSDO 5.0, JQuery Promises were used to handle asynchronous operations. Since JavaScript natively supports Promises with ES6, ES6 Promise is now the preferred method of handling asynchronous operations in JSDO 5.0.

ES6 Promises are also required where JSDO provides a seamless integration between a NativeScript app and a Progress Data Object resource. This is due to the fact that NativeScript has built-in support for ES6 Promises, whereas JQuery Promises cannot be used with either NativeScript or Angular.

In JSDO 5.0, the following changes have been made to the JSDO library:

- Using Promises is no longer optional.
- Asynchronous JSDO APIs use ES6 Promises by default.
- Exceptions in asynchronous APIs call the error handler.

Note: JSDO 5.0 can still use JQuery Promises, if required.

JSDO 5.0 as an npm package

JSDO 5.0 is available as an npm package. This is primarily to support mobile app development in NativeScript, which uses npm to install libraries.

JSDO and Data Source for NativeScript and Angular provide a seamless way of integrating a NativeScript app with a Data Object resource. You can perform either of the following actions:

- Install and use these npm packages in an existing NativeScript app to access a Data Object resource.
- Use the starter template—`tns-template-master-detail-progress-ng`—which, too, is available as an npm package, to create a NativeScript app.

To install the JSDO and Data Source packages in an existing NativeScript project, run the following npm command:

```
npm install @progress/jsdo-nativescript
```

To create a NativeScript app using the starter template, run the following NativeScript command:

```
tns create project_foldername --template tns-template-master-detail-progress-ng
```

JSDO type definitions

JSDO 5.0 includes TypeScript declaration files for the JSDO and the Progress Data Source. These files provide type checking with the TypeScript compiler. Development tools use these files for intelli-sense and refactoring support.

Progress Data Source for NativeScript and Angular

The Data Source component provides seamless integration between client apps built with the Angular framework (that is, NativeScript and Angular web apps) and a Progress Data Object Service. It is available as an npm package on www.npmjs.com, which you can download and use in your NativeScript or web app.

The Data Source module allows you to access data as well as to perform CRUD operations on a data resource from a supported Progress Data Object Service. It is a TypeScript implementation. Therefore, along with the JavaScript code, a declaration file is also provided in the package. The declaration file defines the module's types and function signatures.

JSDO overview

A JSDO is an object designed to simplify access to relational data in a mobile app. It does this by providing JavaScript methods to execute the Data Object operations supported by a single Data Object resource and by supporting an internal data store (*JSDO memory*) to cache the data that is defined by and returned from the Data Object resource to the mobile app. This data is stored in the form of one or more JSDO table objects (*tables*) that map to corresponding elements of the Data Object Service resource that model relational data.

The JSDO relies on a JSON file (*Data Service Catalog*) that defines the Data Object resource it is accessing. This resource definition includes the schema (data model) for the data supported by the resource as well as the definitions for JSDO methods that call the operations supported by the resource. The schema of the Data Object resource, therefore, determines both the structure of the data in JSDO memory and how Data Object operations can interact with it. Other JSDO methods allow the mobile app to read and manipulate the data in JSDO memory, preparing it for ultimate update on the server when supported Data Object operations are called. In addition, JSDO memory provides features to help bind its data to a UI by mapping the data to HTML elements of a mobile app.

The following sections briefly describe:

- JSDO classes and objects on page 39
- How a JSDO maps to a Data Object resource on page 40
- How JSDO memory works on page 41
- Methods of the JSDO and JSRecord classes on page 45
- Asynchronous and synchronous execution on page 49
- Properties of a JSDO on page 56
- Requirements for using a JSDO on page 57

JSDO classes and objects

The JSDO provides the following classes and objects that support the creation and access to JSDO instances:

- **`progress.data.JSDO` class** — Allows you to create JSDO instances for Data Object resources that can asynchronously execute the resource-supported Data Object operations and exchange the data defined by these resources between a mobile app and the server where the data is stored, as described in the progress.data.JSDO class on page 110 topic. For more information, see the remaining topics of this JSDO overview on page 39.
- **`progress.data.JSDOSession` class** — Allows you to create a `JSDOSession` object that asynchronously manages a JSDO login session between the mobile app and a web application running on the server. This enables a JSDO to access the resources provided by a Data Object Service that is supported by this web application, as described in the progress.data.JSDOSession class on page 124 topic. Of the two supported options (`JSDOSession` and `Session`), this is the Progress-recommended class for managing JSDO login sessions. For more information, see Managing JSDO login sessions on page 95.

- **`progress.data.JSRecord` class** — References a record object in JSDO memory, as described in the progress.data.JSRecord class on page 133 topic. For more information, see How a JSDO maps to a Data Object resource on page 40 and How JSDO memory works on page 41.
- **`progress.data.Session` class** — Allows you to create a `Session` object that can manage a JSDO login session, either synchronously or asynchronously, between the mobile app and a web application running on the server. This enables a JSDO to access the resources provided by a Data Object Service that is supported by this web application, as described in the progress.data.Session class on page 136 topic. This is an alternative to the `progress.data.JSDOSession` class, which Progress recommends for managing JSDO login sessions. For more information, see Managing JSDO login sessions on page 95.
- **`progress.ui.UIHelper` class** — Provides basic features for dynamically mapping JSDO data to Web page elements in mobile apps that might be built with HTML tools other than the Telerik AppBuilder, as described in the progress.ui.UIHelper class on page 142 topic. For more information, see Dynamic data binding for alternate client platforms on page 103.
- **Request object** — Contains data both sent in HTTP requests for Data Object operations when they are executed and returned in HTTP responses when the Data Object operation execution completes, as described in the request object on page 146 topic and the topics of this JSDO overview on page 39. For more information, see Accessing standard CRUD and Submit operations on page 61 and Accessing custom Invoke operations on page 89.
- **Table and field references** — References to objects and their properties in JSDO memory that correspond to resource table objects and the fields within record objects contained by these table objects, as described in the table reference property (JSDO class) on page 351 topic. For more information, see Table and field references on page 41 and Working records on page 42.

How a JSDO maps to a Data Object resource

A JSDO is an instance of the `progress.data.JSDO` JavaScript class that you create to access exactly one Data Object resource provided by a Data Object Service. As described in Run-time architecture and data access on page 12, a Data Object resource provides access to either a single-table or a multi-table resource. The exact type of resource that a Data Object provides depends on the Data Object Service (OpenEdge or Rollbase) and the data model selected to create the resource. A Data Object resource is created with a set of standard operations to access its data model. These can include a standard set of Create, Read, Update, Delete (CRUD), and Submit operations, or only a single Read operation, depending on the options chosen to create the Data Object resource. In any case, the same basic set of Data Object operations access the data for every Data Object resource regardless of the data model it supports. The server API that implements these operations have prescribed signatures that depend on the type of data model selected for the Data Object resource. For an OpenEdge resource, this server API consists of ABL routines running on an OpenEdge application server; for a Rollbase resource, this API consists of server-side JavaScript functions running on a Rollbase server.

The prescribed relationship between the standard Data Object operations and the data model of a Data Object resource allows the `JSDO` class to provide a corresponding set of standard JavaScript methods that implicitly map to the standard operations of a Data Object resource, no matter what data model it supports. In addition, for every JSDO, the internal structure of JSDO memory reflects the schema of the particular data model supported by the Data Object resource. Therefore, the standard methods of a JSDO work on the data in its JSDO memory according to the schema defined for the resource data model.

The basic unit of data for a Data Object resource is the table record, which is represented in JSDO memory as a JavaScript *record object* (progress.data.JSRecord class on page 133). The standard Read operation returns a set of records from the server for one or more tables, depending on the data model. The set of records that the Data Object Read operation returns depends on the implementing API routine and an optional filter parameter that you pass to it from the mobile app. The standard Create, Update, and Delete operations of a Data Object resource operate on one table record per server request, according to the implementation of the corresponding API. The Submit operation can operate on an OpenEdge ProDataSet resource for which before-imaging is enabled, for any number of created, updated, and deleted OpenEdge temp-table records per server request. Thus a Submit operation can support multi-record transactions with created, updated, and deleted records. The corresponding standard JSDO methods call these operations accordingly, loading temp-table records into JSDO memory from the server, or creating, updating, and deleting records on the server from posted changes to the record objects in JSDO memory.

A Data Object resource can also support custom Invoke operations which can be implemented by the server API, depending on the Data Object Service and resource. For an OpenEdge resource, the ABL routine that implements an Invoke operation can have any signature defined with supported ABL data types (see Data type mappings for Data Object Services on page 361). For a Rollbase resource, the JavaScript function that implements an Invoke operation always returns a table of records that are related to the Rollbase resource object. A JSDO created for a Data Object resource with Invoke operations has corresponding custom JavaScript invocation methods that map to each Invoke operation. Calling an invocation method on the JSDO calls the implementing server API routine, passing the required input parameters and returning its output parameters and any return value to the mobile app with no direct effect on JSDO memory. The mobile app can do whatever it needs to do with the results of an invocation method, including merging any returned data with JSDO memory, as appropriate.

Note: For more information on coding the ABL routines that implement the Data Object operations for an OpenEdge resource, see the information on Data Object Services in *OpenEdge Development: Web Services*.

How JSDO memory works

JSDO memory stores table records as record objects according to the schema of the Data Object resource that is mapped to the JSDO. If the data model is for a single table, JSDO memory can contain only record objects for the specified table.

If the data model is for a multi-table resource (such as an OpenEdge ProDataSet), JSDO memory can contain record objects for all the tables defined in the multi-table resource. By default, record objects for a multi-table resource are maintained in JSDO memory according to any data relations defined between the tables. This means, for example, that when a JSDO method finds a record object of a parent table, if a method is then called to search through the record objects of a table that is a child of that parent, the search will find only record objects that are related to the record object found in the parent; if new record objects are added to the same child table, they are added with key fields set implicitly in relation to the parent record object. The JSDO also supports a `boolean` run-time option (the `useRelationships` property) that toggles between honoring these data relations and ignoring them when you access table data in JSDO memory.

Table and field references

You can access the data in JSDO memory using table references. A *table reference* is a property on the JSDO that references a given JavaScript table object as defined by the schema of the resource data model. So, JSDO memory contains one table reference property for each table object referenced in JSDO memory. The name of each property is the same as the name of a corresponding table (or object) defined in the schema of Data Object resource, and it is specified with the same letter case as the table or object name in the resource.

JSDO methods that operate on JSDO memory operate either on the entire data store, in which case they are called on the JSDO itself, or on one table reference at a time, in which case they are called directly on the corresponding table reference property. As a short hand, methods that operate on a single table reference can be called on the JSDO if the JSDO schema includes only one table. For example, given a JSDO referenced by `dsOrderEntry` whose JSDO memory references the schema for several temp-tables of an OpenEdge ProDataSet, including `ttCustomer`, the JSDO `fill( )` and `foreach( )` methods can be called as follows:

```
dsOrderEntry.fill( );
dsOrderEntry.ttCustomer.foreach( function ( record-object ) { ... } );
```

In this example, the `fill( )` method is called on the `dsOrderEntry` JSDO to load the available data for the Data Object resource into JSDO memory by calling the standard Data Object Read operation. Then, the `foreach( )` method is called on the `ttCustomer` property of the JSDO to loop through all the record objects loaded from the `ttCustomer` temp-table, allowing each one to be accessed from within the function that is passed as a parameter.

You can access the data in each record object of a table reference using a field reference. A *field reference* is a property on a JSDO table reference that references a field in the current working record (if available) as specified by the schema of the resource table data model. Each field reference property has the name and data type of a field in the table schema and its current value in the working record.

The JSDO also supports access to the elements of one-dimensional array fields using either standard JavaScript subscripts on an array field reference or JSDO-defined array-element references. A *JSDO array-element reference* is an individual field reference property whose name consist of the array field name appended with a one-based integer that corresponds to each element of the array.

For more information on using table reference and field reference properties to access tables and fields in a JSDO, see the description of the table reference property (JSDO class) on page 351.

A *working record* is the current record object that is implicitly available on a table reference after certain JSDO methods are called that set a working record. For more information on field references and working records, see Working records on page 42.

Working records

When a JSDO method is called, depending on the method and the situation, it might result in setting a working record for one or more of the table references. A *working record* is a record object that is associated with a table object in JSDO memory and that is available to reference implicitly using the table reference. If a table reference has a working record set for it, you can then reference any field value in the working record using its field reference property on the table reference. The name of a field reference property is the same as the name of the corresponding field in the table schema that is defined for the Data Object resource. You can also save the record object returned as a working record for future reference (as described in this topic), allowing you to access its field values at any time during the same JSDO session.

For example in Table and field references on page 41, when the `foreach( )` method in the example returns from execution (based on the return value of its function parameter), it leaves a working record set for the `ttCustomer` table reference. You can assign a value to the `Name` field of this working record using the `Name` field reference property on the table reference:

```
dsOrderEntry.ttCustomer.Name = "Evellyn Doe";
```

This is one of the supported mechanisms to update a field value in JSDO memory. The other is to call the `assign( )` method directly on a table reference or on a saved record object, itself.

Also, if JSDO memory has a multi-table data model with active data-relations and a working record is set in a parent table reference, a search through the records of a child table reference reflects the data-relation. For example, with a working record set for a `ttCustomer` that is a parent of `ttOrder`, calling the `foreach( )` method on `ttOrder` only loops through records related to the working record in `ttCustomer`.

Note: Table references for tables referenced in JSDO memory use a flat reference model. In other words, regardless of data-relations, you reference the property for each table reference directly on the JSDO (for example, `ttOrder` in `dsOrderEntry.ttOrder.Ordernum`).

Depending on the results of JSDO methods that you call, JSDO memory either leaves a working record set for each table reference or leaves the working record undefined for one or more table references. Essential programming for a JSDO involves calling appropriate methods to locate working records in JSDO memory in order to read and modify their contents (see also Record IDs on page 45). The documentation for each JSDO method (see JSDO properties, methods, and events reference on page 149) indicates whether and how it leaves working records set when it completes execution.

The following table lists each JSDO method that has an effect on working record settings and generally how it affects them.

Table 6: JSDO methods that affect working record settings

This JSDO method...	Leaves working records...
`acceptChanges( )`	Set depending on the changes accepted
`acceptRowChanges( )`	Set depending on the changes accepted
`add( )`	Set for the affected table
`addLocalRecords( )`	Set for each JSDO table reference depending on the specified merge mode
`addRecords( )`	Set for each JSDO table reference depending on the specified merge mode
`create( )`	Set for the affected table
`fill( )`	Set for every table of the JSDO according to any active data-relations and the current table sort order
`find( )`	If a record is found, set for the affected table, any child tables, and the effects of the callback function
`findById( )`	Set for the affected table
`foreach( )`	Set for the most recent record found in the affected table, any child tables, and the effects of the callback function
`read( )`	Set for every table of the JSDO according to any active data-relations and the current table sort order
`readLocal( )`	Not set for any table of the JSDO

This JSDO method...	Leaves working records...
`rejectChanges( )`	Set depending on the changes rejected
`rejectRowChanges( )`	Set depending on the changes rejected
`remove( )`	No longer set for the affected table and its child tables
`saveChanges( )`	Not longer set for every table in the JSDO

If a method sets a working record, you can reference the field values of the record object using field reference properties on the table reference, as described in the previous section. You can also use the `record` property on the table reference to return a `JSRecord` object reference to the working record, which you can reference directly or store separately to access contents of the record object later. A `JSRecord` object provides a `data` property that references a separate JavaScript object containing the actual field reference properties for the record (see also progress.data.JSRecord class on page 133). Also as a convenience, if the JSDO supports only a single table, you can access the `record` property for the working record directly on the JSDO reference itself.

So, using the previous example with `ttCustomer`, you can access the `Name` field of the working record using a `JSRecord` object reference in the following ways:

```
var custName = dsOrderEntry.record.data.Name; // Single table only
var custName = dsOrderEntry.ttCustomer.Name; // One table of many
var custName = dsOrderEntry.ttCustomer.record.data.Name;
var custRecord = dsOrderEntry.ttCustomer.record; // Stored for later reference
var custName = custRecord.data.Name; // Later reference
```

Once you store the record object of a working record for later reference, the stored record object remains available using its `JSRecord` reference even when it is no longer the working record for the table. Note that Progress does not recommend using the `data` property to **write** a value to a field because the record object is **not** marked as changed in JSDO memory and won't be updated on the server. To update a field value in a `JSRecord` object so the change is reflected on the server, either call the `assign( )` method directly on the `JSRecord` reference or assign the value directly to the field reference property that you reference directly on the table reference where the appropriate working record is set.

Note: One reason to use the `record.data` property on a table reference is to **read** a field value when the field happens to have the same name as a JSDO method that you can call on any table reference.

Caution: Because the record object is not marked as changed in JSDO memory, **never** use the `data` property on a `JSRecord` reference to update the value of a field. Otherwise, the change in value will never be reflected on the server. Use the `data` property to **read** the field value only.

Record IDs

One difference between the JavaScript record objects in JSDO memory and the table records that they represent on the server is that the JSDO creates each record object with a local record ID. This is an internal field reference with the OpenEdge-reserved name that uniquely identifies the record in JSDO memory. This record ID has no relationship to any values maintained to identify records on the server (such as the `RECID` and `ROWID` values maintained for the records of an OpenEdge database). Instead, this record ID is used by the JSDO `progress.ui.UIHelper` class to map the record objects in JSDO memory to HTML elements in mobile and web apps. The JSDO provides the `getId( )` method for you to retrieve this record ID from any record object in JSDO memory and store it for future reference in order to efficiently retrieve that record object again from JSDO memory.

Note: The value assigned to the internal record ID for any given record object can change with each invocation of the `fill( )` or `saveChanges( )` method. For more information on these methods, see Methods of the JSDO and JSRecord classes on page 45.

Methods of the JSDO and JSRecord classes

Every JSDO has a number of built-in methods, many of which interact with JSDO memory. Some are called on the JSDO itself, while most are called on a JSDO table reference. Most execute synchronously, but some execute asynchronously as noted. For more information, see Asynchronous and synchronous execution on page 49. A few JSDO methods are also available on a `JSRecord` object.

Some methods affect working record settings in JSDO memory as noted, while all other methods have no effect on working record settings.

A JSDO includes built-in methods for:

- **Executing standard Data Object operations and calling their implementing server API:**
  - **`fill( )` (or its alias, `read( )`)** — Executes the Read operation on a Data Object resource, loading the records sent from the server into JSDO memory according to the resource data model; the first record of every loaded table reference is set as its working record.
  - **`saveChanges( )`** — Executes the Create, Update, Delete (CUD), or Submit operation on a Data Object resource for each record that has changed in JSDO memory since the last execution of the `fill( )` (or alias `read( )`) or `saveChanges( )` method; no working records are set for any tables in JSDO memory.

  You call these methods on the JSDO and they always execute asynchronously, with results returned using handlers either or both for named events and jQuery Promises.
- **Executing custom Data Object Invoke operations that directly call a server API.** In order to call a server API, the Data Object resource defines a uniquely named, JSDO invocation method that calls each Invoke operation supported by the resource. These invocation methods can return results from their corresponding Invoke operations, but they have no direct effect on JSDO memory. You can call a JSDO invocation method using one of the following mechanisms:
  - **Calling the method directly on the JSDO** — This mechanism provides the option to execute the Invoke operation either asynchronously (the default) or synchronously. To pass input parameters to the implementing server API, you pass an object to the invocation method that contains properties with the values for the operation parameters. You can return asynchronous results using handlers for named events, and you can return synchronous results as an object method value.

Note: For an OpenEdge resource, each property that corresponds to an input parameter of an implementing ABL routine has the same name (and letter case) as the corresponding ABL input parameter, and has a JavaScript data type that maps to the ABL data type of the input parameter (see OpenEdge ABL to JavaScript data type mappings on page 363).

- **Calling the method by using the JSDO `invoke( )` method API** — This mechanism always executes the Invoke operation asynchronously. In this case, you pass both the name of the invocation method and any input object that the invocation method requires to the `invoke( )` method, which you call directly on the JSDO. You can return the asynchronous results using handlers either or both for named events and jQuery Promises.

- **Updating JSDO memory:**

  - **`acceptChanges( )`** — Called on the JSDO or a JSDO table reference, this method accepts changes to the data in JSDO memory for the specified table reference or for all table references of the specified JSDO; the effect on working record settings depends on the changes accepted. This method works only if you have set the `autoApplyChanges` property to `false`.

  - **`acceptRowChanges( )`** — Called on a JSDO table reference or a `JSRecord` reference, this method accepts changes to the data in JSDO memory for a specified record object; the effect on working record settings depends on the changes accepted. This method works only if you have set the `autoApplyChanges` property to `false`.

  - **`add( )` (or its alias, `create( )`)** — Called on a JSDO table reference, this method creates a new record object for the table in JSDO memory; the new record is set as the working record for the table reference.

  - **`addRecords( )`** — Called either on a JSDO or on a JSDO table reference, this method merges record objects from a merge object passed as a method parameter with the specified existing record objects in JSDO memory. The merge object must follow the same schema as JSDO memory itself. Merge modes determine how to handle record objects with duplicate key fields, if specified. The case sensitivity for merges on `string` fields can be changed by setting the `caseSensitive` property. This method sets working records for all JSDO table references depending on the merge mode that is used.

  - **`assign( )` (or its alias, `update( )`)** — Called on a JSDO table reference or a `JSRecord` reference, this method updates field values for an existing record in JSDO memory from property values in an object that you pass as a parameter to the method.

    Note: An `assign( )` method is also available on a table reference of a `progress.ui.UIHelper` object. This method updates the field values of an existing record from values in an associated HTML page. For more information, see Dynamic data binding for alternate client platforms on page 103.

  - **`rejectChanges( )`** — Called on the JSDO or a JSDO table reference, this method rejects changes to the data in JSDO memory for the specified table reference or for all table references of the specified JSDO; the effect on working record settings depends on the changes rejected. This method works only if you have set the `autoApplyChanges` property to `false`.

  - **`rejectRowChanges( )`** — Called on a JSDO table reference or a `JSRecord` reference, this method rejects changes to the data in JSDO memory for a specified record object; the effect on working record settings depends on the changes rejected. This method works only if you have set the `autoApplyChanges` property to `false`.

  - **`remove( )`** — Called on a JSDO table reference or a `JSRecord` reference, this method deletes an existing record in JSDO memory; no working record is set for the table reference or any of its child table references.

These methods always execute synchronously. You save the changes that these methods make in JSDO memory to the server by calling the `saveChanges( )` method on the JSDO, which executes asynchronously.

- **Moving data between JSDO memory and local device storage:**
  - **`addLocalRecords( )`** — Called on the JSDO, this method merges record objects from a specified local storage area with the existing record objects in JSDO memory. The local storage area must follow the same schema as JSDO memory itself. Merge modes determine how to handle record objects with duplicate key fields, if specified. The case sensitivity for merges on `string` fields can be changed by setting the `caseSensitive` property. This method sets working records for all JSDO table references depending on the merge mode that is used.
  - **`deleteLocal( )`** — Called on the JSDO, this method removes all data and changes stored in the specified local storage area, and removes the storage area.
  - **`hasChanges( )`** — Called on the JSDO, this method indicates if JSDO memory has any pending changes (with or without before-image data) and is typically used to determine if there are any changes in JSDO memory that you might want to save to a local storage area.
  - **`hasData( )`** — Called on the JSDO or a JSDO table reference, this method indicates if either all tables, or a specified table, in JSDO memory contains record objects (with or without pending changes) and is typically used to determine if there is any data in JSDO memory that you might want to save to a local storage area, for example, to avoid losing if you decide to replace JSDO memory with other records from a different local storage area.
  - **`readLocal( )`** — Called on the JSDO, this method removes all data in JSDO memory and replaces it with the data in a specified local storage area, including pending changes, if any. After execution, this method leaves no working record settings in the JSDO.
  - **`saveLocal( )`** — Called on the JSDO, this method removes any existing data in a specified local storage area and replaces it with the data in JSDO memory as specified by a data mode, which indicates whether JSDO data and changes or only the data changes are saved to the storage area.
- **Sorting record objects in JSDO memory:**
  - **`setSortFields( )`** — Specifies or clears the record fields on which to automatically sort the record objects for a table reference after you have set its `autoSort` property to `true`. When enabled, this automatic sorting occurs in JSDO memory after supported JSDO operations update the associated data.
  - **`setSortFn( )`** — Specifies or clears a user-defined sort function on which to automatically sort the record objects for a table reference after you have set its `autoSort` property to `true`. When enabled, this automatic sorting occurs in JSDO memory after supported JSDO operations update the associated data.
  - **`sort( )`** — Sorts the existing record objects for a table reference in JSDO memory using either specified sort fields or a specified user-defined sort function. This method sorts the record objects for a table reference whether or not the associated data has changed.

  These methods are all called on a JSDO table reference, and always execute synchronously. The case sensitivity of sorting on `string` fields can be changed by setting the `caseSensitive` property.
- **Searching for record objects in JSDO memory:**
  - **`find( )`** — Searches for a record in a referenced table according to the criteria defined by a function that you pass, and returns the record object if the function indicates it has been found; sets any record found as the working record for the table reference, and (assuming the `useRelationships` property is `true`) sets the working record for any child table references to the first record that is related to the parent working record.

- **`findById( )`** — Searches for a record in a referenced table with the specified record ID, and returns the record object if found; sets any record found as the working record for the table reference, and (assuming the `useRelationships` property is `true`) sets the working record for any child table references to the first record that is related to the parent working record.
- **`foreach( )`** — Loops through the records of a referenced table, and allows a function that you pass to access each record object and perform whatever actions you define until the function tells the method to stop looping or the method has reached the end of the record set; the record for each iteration is set as the working record for the table reference, and (assuming the `useRelationships` property is `true`) sets the working record for any child table references to the first record that is related to the parent working record. When the loop terminates, the last working record set remains the working record for the table reference.

These methods are called on a JSDO table reference and always execute synchronously.

- **Returning data and information from JSDO memory:**
  - **`getData( )`** — Called on a JSDO table reference, this method returns an array of record objects for a referenced table.
  - **`getErrors( )`** — Called on a JSDO table reference, this method returns all types of error information associated with the table reference from the most recent call to the `fill( )` or `saveChanges( )` method. This error information is returned as an array of objects, each of which has properties that return the information for a specific error according its type.

    Note: Updated for Progress Data Objects Version 4.3 or later.

  - **`getErrorString( )`** — Called on a JSDO table reference or a `JSRecord` reference, this method returns the value of any before-image returned in the specified record object after a record change operation that includes before-image data.
  - **`getId( )`** — Called on a JSDO table reference or a `JSRecord` reference, this method returns the unique internal record ID for the specified record object.
  - **`getSchema( )`** — Called on a JSDO table reference, this method returns an array of objects, one for each field, that defines the schema of the referenced table.

  These methods always execute synchronously.

- **Managing JSDO event subscriptions:**
  - **`subscribe( )`** — Subscribes a given event handler function to a named event on a JSDO or on a JSDO table reference.
  - **`unsubscribe( )`** — Unsubscribes a given event handler function from a named event on a JSDO or on a JSDO table reference.

  These methods are called on the JSDO or on a JSDO table reference, and execute synchronously. For more information on JSDO events and managing JSDO event subscriptions, see Asynchronous and synchronous execution on page 49.

- **Creating and accessing JSDO user-defined properties:**
  - **`getProperties( )`** — Returns an object containing the names and values of the user-defined properties defined in the current JSDO instance.
  - **`getProperty( )`** — Returns the value of the specified JSDO user-defined property.

- **`setProperties( )`** — Replaces all user-defined properties in the current JSDO instance with the user-defined properties defined in the specified object.
- **`setProperty( )`** — Sets the value of the specified JSDO user-defined property.

These methods are called on the JSDO only, execute synchronously. For more information on user-defined properties, see Properties of a JSDO on page 56.

Note: Applies to Progress Data Objects Version 4.3 or later.

Asynchronous and synchronous execution

As described in the previous section, most JSDO methods operate locally on the client and execute synchronously in JavaScript. *Synchronous execution* means that the client waits for the method to complete execution and return its results before executing the next JavaScript statement or evaluating the next term of an expression. These results often include a method return value.

Other JSDO methods that access data, or otherwise operate on the server, execute asynchronously in JavaScript. These include the JSDO `fill( )` and `saveChanges( )` methods, which execute standard Data Object operations on the server, as well as the default execution of custom invocation methods (with or without use of the JSDO `invoke( )` method), which execute custom Data Object Invoke operations on the server. *Asynchronous execution* means that immediately after the client calls the method, it continues to execute the next JavaScript statement or to evaluate the next term of an expression whether or not the asynchronous method completes and returns results. These results are returned to one or more callback functions that you define and that execute on the client according to certain conditions.

You can define a *callback function* (or *callback*) for an asynchronous method that can execute in one of the following ways:

- **As a handler subscribed to a named event** — Which executes after invoking the asynchronous method when a specified condition for firing the named event occurs. Thus, multiple callbacks can execute when the asynchronous method completes, depending on how many you subscribe to events that fire in response to invoking the method.
- **As a callback registered using a method of a deferred Promise object** — Which executes after invoking the asynchronous method when a specified condition occurs that is associated with the Promise object method that registered the callback. This Promise object, itself, is returned as the deferred value of the asynchronous method, allowing you to call one or more Promise object methods that register associated callbacks. Thus, multiple callbacks can execute when the asynchronous method completes, depending on how many of the conditions occur that are associated with the Promise object methods that you have called to register these callbacks.

Thus, the results of an asynchronous method execution only become available to the mobile app when either or both of the following occur, and in the following order:

1. An event associated with the asynchronous method fires, and a callback function that is subscribed to the event executes and receives the results as one or more input parameters.
2. A condition occurs that is associated with a deferred Promise object method that you have called to register a callback, and the callback executes and receives the results as one or more input parameters.

The callbacks that are available to return the results of asynchronous JSDO method calls depend on:

- The named events supported for a given JSDO method call. For more information, see Asynchronous execution using named events on page 50.

- Whether your environment supports Promises. For more information, see Asynchronous execution using Promises on page 52.

For more information on the differences between JSDO methods that execute asynchronously and JSDO methods that execute synchronously, see Comparing asynchronous and synchronous execution on page 50.

Comparing asynchronous and synchronous execution

Asynchronous execution is mandatory for the methods that execute the standard Data Object CRUD and Submit operations. These operations usually involve server access to data sources, which are typically databases. The Data Object Read operation executed by the `fill( )` method can involve reading and returning hundreds to thousands (or more) records from a multi-table resource across the network. The Data Object Create, Update, Delete (CUD), and Submit operations, which require writes to, and lock management of, databases, are executed across the network by the `saveChanges( )` method one record at a time for CUD operations and multiple records at a time for the Submit operation, and for as many records as are marked changed in JSDO memory.

This means that completion of these operations can require detectable wait times. If they were executed synchronously, the mobile app user would be prevented from doing any work within the app while these methods are executing. With asynchronous execution, they can perform other tasks, such as setting application options, while waiting for a list of customers and orders, for example, to be displayed in a list view, or while waiting for notification that changes they have posted have been saved to the database. For more information on the execution options for methods that execute the standard Data Object CRUD and Submit operations, see Accessing standard CRUD and Submit operations on page 61.

Asynchronous execution is also the default for custom invocation methods, again, because these operations execute across the network and can involve complex database interactions on the server. However, it is also possible that an invocation method might perform a simple function and return a primitive value used in an expression—for example, a percentage or a credit limit. In this case, you might prefer to execute the method synchronously in the expression in order to complete the calculation with its return value. This you can do by passing `false` as an additional `boolean` parameter that indicates that the invocation method is to be executed synchronously. For more information on the execution options for invocation methods, see Accessing custom Invoke operations on page 89.

Asynchronous execution using named events

For all JSDO methods that execute asynchronously, the JSDO supports a set of named events to which you can subscribe *event handler functions*, callback functions that execute with a signature defined to receive the results for a particular named event.

These JSDO named events fire before and after each Data Object operation executes, as well as before and after the following asynchronous JSDO methods whose execution invokes these operations:

- Standard `fill( )` method
- Standard `saveChanges( )` method
- Asynchronously executed custom invocation methods and the standard `invoke( )` method

In addition, the JSDO provides a `subscribe( )` method that allows you to subscribe event handler callbacks to each of its events and provides the `unsubscribe( )` and `unsubscribeAll( )` methods to remove these event handler subscriptions. For more information, see Managing JSDO event subscriptions on page 53.

Therefore, the JSDO supports the following `before*` and `after*` events for each asynchronous method:

- **For the standard `fill( )` method**:
  - **`beforeFill` (or its alias, `beforeRead`)** — Fires before the Data Object Read operation executes. For more information, see the description of the beforeFill event on page 208.

  - **`afterFill` (or its alias, `afterRead`)** — Fires after the Data Object Read operation completes execution. For more information, see the description of the afterFill event on page 185.

- **For the standard `saveChanges( )` method**:
  - **`beforeSaveChanges`** — Fires before the method executes any Data Object operations. For more information, see the description of the beforeSaveChanges event on page 210.
  - **`beforeDelete`** — Fires before any Data Object Delete operation executes and before any record delete initiates as part of a Submit operation. For more information, see the description of the beforeDelete event on page 207.
  - **`afterDelete`** — Fires after each Data Object Delete operation completes execution and after any record delete completes as part of a Submit operation. For more information, see the description of the afterDelete event on page 183.
  - **`beforeCreate`** — Fires before any Data Object Create operation executes and before any record create initiates as part of a Submit operation. For more information, see the description of the beforeCreate event on page 206.
  - **`afterCreate`** — Fires after each Data Object Create operation completes execution and after any record create completes as part of a Submit operation. For more information, see the description of the afterCreate event on page 181.
  - **`beforeUpdate`** — Fires before any Data Object Update operation executes and before any record update initiates as part of a Submit operation. For more information, see the description of the beforeUpdate event on page 211.
  - **`afterUpdate`** — Fires after each Data Object Update operation completes execution and after any record update completes as part of a Submit operation. For more information, see the description of the afterUpdate event on page 195.
  - **`afterSaveChanges`** — Fires after all the Data Object operations executed by the method have completed (including a single Submit operation, if applicable). For more information, see the description of the afterSaveChanges event on page 192.

Note: These events are listed in general firing order for a single call to the `saveChanges( )` method. When there are multiple changes to the same record, the order and number of changes is optimized to send the fewest number of operation requests to the server.

Note: If `saveChanges( )` executes for a Data Object resource (and operation) defined to support before-image data, and an error is raised while a record change is applied to the database on the server, the record change operation also completes with an error, and with a possible value returned as the value of `getErrorString( )` called on the associated record object.

- **For any asynchronously executed custom invocation methods and the standard `invoke( )` method**:
  - **`beforeInvoke`** — Fires before the specified Data Object Invoke operation executes. For more information, see the description of the beforeInvoke event on page 209.
  - **`afterInvoke`** — Fires after the specified Data Object Invoke operation completes execution. For more information, see the description of the afterInvoke event on page 186.

Note: When you subscribe an event handler to an Invoke operation event, you also specify the name of the custom JSDO invocation method so the event handler can identify which Invoke operation is about to be, or has been, executed. Note also that when you execute an invocation method synchronously, these Invoke events **do not** fire because all synchronous Invoke operation results are available as the return value of the invocation method itself.

Note: The `saveChanges( )` method can fire all of the `before*` and `after*` events for Create, Update, and Delete (CUD) operations for any number of records in JSDO memory, whether they are fired for one CUD operation per record, as part of one network request at a time, or are all fired in the context of a single network request (as part of a single Submit operation) as creates, updates, and deletes of multiple records. In other words, there is a unique event that fires **before** every record is created, updated, or deleted, and one that fires **after** every record is created, updated, or deleted, whether it is done one record at a time across the network or as part of a batch of records in a single network request. In addition, there is one `beforeSaveChanges` event that fires **before** the `saveChanges( )` method initiates any record creates, updates, and deletes and one `afterSaveChanges` event that fires **afterall** of these record creates, updates, and deletes complete, whether they are executed in multiple CUD operation requests across the network or are all executed in the context of a single Submit operation request across the network.

Note: All **after** events always fire whether or not Data Object operations complete successfully.

Asynchronous execution using Promises

The JSDO allows most asynchronous methods to return results using jQuery Promises (if available) in addition to any results returned using named events (see Asynchronous execution using named events on page 50). This Promise support is available for the following standard JSDO methods:

- `fill( )`
- `invoke( )`
- `saveChanges( )`

A *Promise* is a deferred object that a supported asynchronous method returns immediately as its value when you call it. You can then call methods on the returned Promise object to register callback functions that execute for the asynchronous method when results are available depending on conditions associated with each Promise object method. You provide your own code to handle the results returned by each callback that you register, much like the code you provide in a named event handler callback.

Note: To use Promises to return asynchronous JSDO results, your environment must support jQuery Promises or the exact equivalent. Note also that jQuery Promises are both required and available to return asynchronous JSDO results when you access resources of a Progress Data Object Service using the JSDO dialect of the Kendo UI DataSource. (The jQuery libraries are included with Kendo UI.) For more information, see Using the JSDO dialect of the Kendo UI DataSource on page 17.

The primary Promise methods for use with a JSDO object include `done( )`, `fail( )`, and `always( )`, which allow you to separately handle the successful, unsuccessful, and all results (respectively) of a JSDO method execution. Note that for any given JSDO method, the parameter lists for all Promise method callbacks have the same signature. Note also that the callback for the `always( )` method executes with the same values in its parameter list as the callback for the `done( )` or `fail( )` method, whichever one executes with results. The descriptions of the JSDO methods that return results using Promises show several examples of their use. For more information on the jQuery implementation of Promises and the additional Promise methods that are available, see the jQuery topics at the following Web locations:

- **Category: Deferred Object** — http://api.jquery.com/category/deferred-object/
- **Promise Object** — http://api.jquery.com/Types/#Promise
- **.promise** — http://api.jquery.com/promise/

The `fill( )` and `saveChanges( )` methods similarly allow you to return their asynchronous results using either or both named events and Promises. Note that directly calling a custom JSDO invocation method asynchronously only returns results using named events. To return asynchronous results of an invocation method using either or both named events and Promises, you must call the invocation method using the API provided by the standard JSDO `invoke( )` method.

The signatures for Promise callbacks are identical to the signatures of the callbacks for the `after*` event for each asynchronous JSDO method that supports Promises as follows:

- **`fill( )`** — `afterFill` event callback signature
- **`invoke( )`** — `afterInvoke` event callback signature
- **`saveChanges( )`** — `afterSaveChanges` event callback signature

This allows you to share callbacks for both subscription to appropriate named events and registration by appropriate Promise methods.

Note that when using both named events and Promises to return results for a given asynchronous method, the results are returned to callbacks in the following general order:

1. To `before*` event callbacks
2. To `after*` event callbacks
3. to Promise callbacks.

For a more detailed description of results handling for named events and Promises, see the description of each JSDO method that supports Promises.

Managing JSDO event subscriptions

You can subscribe event handlers to JSDO events using either the `subscribe( )` method on the JSDO or by setting appropriate properties in an initialization object that you can pass to the constructor used to instantiate the JSDO. If you use the `subscribe( )` method after the JSDO is instantiated and its JSDO memory has been loaded with records, you can also subscribe to events for the Data Object Create, Update, and Delete (CUD) operations that either execute for a specific table reference or for all table references of the JSDO.

When you subscribe an event handler function to a JSDO event, the parameter list for the function must match the parameter list defined for the event. However, all JSDO event handlers receive a reference to the JSDO as its first parameter and a reference to a request object as its last parameter that contains event results (see the following section). All handlers for `after*` events receive a `boolean` parameter that indicates the success of the Data Object operation (or operations for `afterSaveChanges` without Submit). All handlers for events fired by Data Object CUD operations receive a `JSRecord` object parameter that represents the record object in JSDO memory that is created, updated, or deleted. For more information on the required parameter list for each event, see the reference entry for the event in JSDO properties, methods, and events reference on page 149.

Regardless of how you subscribe event handlers to an event, you can remove an event subscription for one or more an event handlers using the `unsubscribe( )` or `unsubscribeAll( )` method. If an event has no event handler subscribed and the event fires, it returns no results to the app.

Handling asynchronous and synchronous execution results

For the synchronous JSDO methods that do not execute Data Object operations (such as `add( )` or `find( )`), the results for both successful and unsuccessful execution are as defined for each method. For more information, see the reference entry for the method in JSDO properties, methods, and events reference on page 149.

For methods that execute Data Object operations asynchronously (`fill( )`, `saveChanges( )`, and invocation methods), the results for each event or Promise are returned in a general request object that is passed as the last parameter to the event or Promise callback function. For invocation methods that you execute synchronously, a reference to this same request object is available as the return value of the method. This request object has a number of properties whose values depend on the event or Promise callback that returns the object. Some properties (including the `jsdo`, `jsrecord`, and `success` properties) duplicate the settings of the other parameters passed to the event or Promise callback. The settings of other properties provide additional information appropriate for the event or Promise callback. In addition, as of Progress Data Objects 4.3 or later, you can call the `getErrors( )` method on a JSDO table reference in any callback to return errors associated with the table reference from the most recent call to `fill( )` or `saveChanges( )` (but **not** from a call to an invocation method).

One of the most important properties on the request object is the `response` property. This property is set only for the `after*` events of all Data Object operations. Depending on the type of Data Object (OpenEdge or Rollbase), it references JavaScript data that a Data Object operation returns for a successful or unsuccessful completion.

For a standard Data Object CRUD or Submit operation that completes successfully, this property references JavaScript that contains data for the returned single-table or multi-table resource converted from any valid JSON returned for the operation over the network. If no valid JSON is returned, this property is `undefined`.

For a custom Invoke operation that completes successfully, the `response` property references an object that contains properties with the values of any output parameters and return value of the server routine. For an Invoke operation on an OpenEdge resource, a property for an ABL output parameter has the same name (and letter case) as the corresponding output parameter, and has a JavaScript data type that maps to the ABL data type of the output parameter (see OpenEdge ABL to JavaScript data type mappings on page 363). Any return value from the routine is returned as the OpenEdge-defined property, `_retVal`, also with an ABL-mapped JavaScript data type.

For an OpenEdge Data Object operation that completes with one or more ABL errors, the `response` property references an error object that contains two OpenEdge-defined properties:

- **`_retVal`** — A `string` with the value of any ABL `RETURN ERROR` string or `ReturnValue` property for a thrown `AppError` object.
- **`_errors`** — An array of JavaScript objects, each of which contains properties with the ABL error message string and error number for one of possibly many ABL-returned errors.

For CUD or Submit operations on OpenEdge Data Object resources that support before-imaging, if the `response` property returns no error object, you can return any error messages stored with the before-image data for a record by calling the `getErrorString( )` method on the record object. For more information, see the getErrorString( ) method on page 239 description.

For a Rollbase Data Object operation that completes with an error, the `response` property is a simple `string` containing information about the error.

For any Data Object operations that complete with errors originating from the network, web server, or web application that hosts the Progress Data Object Service, you can inspect the `xhr` property on the request object to identify the error information. This property references the XMLHttpRequest object used to send and receive operation requests. For more information on this object, see the software development documentation for your web browser or mobile device.

For more information on the request object and its available properties, see the request object on page 146 description.

Note that you can avoid any inspection of the `response` or `xhr` properties to return errors from calls to `fill( )` or `saveChanges( )` by calling the `getErrors( )` method. This method returns **all** errors associated with a specified table reference, regardless of how they originated, whether from the resource operations themselves or from the network, web application, or web server that hosts the Data Object Service. For more information, see the getErrors( ) method on page 233 description.

General error handling for Data Object operations

For any Data Object operation that completes with an error of any type (operation execution, web application, server, or network), the `success` property of the returned request object (and the *`success`* parameter passed to any event or Promise callback) is set to `false`. As noted in the previous topic (see Handling asynchronous and synchronous execution results on page 54), error information from operation execution can be found using two separate mechanisms:

- Calling the `getErrors( )` method on a JSDO table reference to return associated errors from the most recent call to `fill( )` or `saveChanges( )`.
- Inspecting the `response` and `xhr` properties of the request object for information on operation errors and on web application, web server, or network errors, respectively, from the most recent call to `fill( )`, `saveChanges( )`, or an invocation method.

Note that for a Data Object Create, Update, Delete (CUD), or for a Submit operation (on a before-image resource only), if **any** error, at any point causes the operation to complete unsuccessfully, by default (with the `autoApplyChanges` property set to `true`), the record or records in JSDO memory are reverted prior to any changes that caused the operation or operations to execute by invoking `saveChanges( )`, and the intended record changes are preserved as values of appropriate request object properties.

For example, a failed Create operation causes the added record to be removed from JSDO memory. The record object (or objects for Submit with before-imaging) that were originally added for the operation are then available for your re-use as the value of the `jsrecord` property (or for Submit with before-imaging, as the value of the `jsrecords` property) of the request object (or of the *`record`* parameter passed to the `afterCreate` event callback). A similar reversion occurs for the Update and Delete operations (or updated and deleted records of a Submit with before-imaging), with the field values of any updated record reverted to their original values, and any deleted record added back to JSDO memory. Again, the original updated or deleted record or records are available using the appropriate `jsrecord` or `jsrecords` property of the request object.

Note: When the Data Object resource (and operation) supports before-imaging, if a Data Object CUD or Submit operation completes with an error for a given record change applied to the server database, the associated record object can include a before-image error string as part of its data that can be set to a message describing the error. You can return this value, if defined, by invoking the `getErrorString( )` method on the record object.

However, you can change this default behavior by setting the `autoApplyChanges` property to `false`. In this case, when the operation completes (successfully or unsuccessfully) you can manually accept or reject all, or selected, record changes in JSDO memory using one off the following methods:

- `acceptChanges( )`
- `acceptRowChanges( )`
- `rejectChanges( )`
- `rejectRowChanges( )`

Note that when `autoApplyChanges` is `true`, corresponding errors returned from a call to `saveChanges( )` are cleared from JSDO local memory (and also the `response` and `xhr` properties of returned request objects) at the following points, which ever comes first:

- Execution completes for the final callback subscribed to an operation event or registered to the Promise object returned by `saveChanges( )`.
- A corresponding `accept*Changes( )` or `reject*Changes( )` method is executed.

However, if you need access to errors from the most recent call to `fill( )` or `saveChanges( )` beyond these points, you can return them for any associated JSDO table reference by calling `getErrors( )`, regardless of the setting of `autoApplyChanges` or the execution of `accept*Changes( )` and `reject*Changes( )` methods. These errors are thus available for return until the next call to either `fill( )` or `saveChanges( )`.

For more information on handling errors for Data Object operations, see the descriptions of the methods that invoke these operations:

- fill( ) method on page 220
- saveChanges( ) method on page 314
- invocation method on page 262
- invoke( ) method on page 264

Properties of a JSDO

Every JSDO has several built-in properties and supports the creation of user-defined properties to manage its state. We have already introduced the following JSDO built-in properties:

- Table and field reference properties that provide access to table data loaded into JSDO memory and the `record` property, which provides access to individual record objects on a given table reference (see How JSDO memory works on page 41)
- The `autoApplyChanges` property on a JSDO, which indicates whether record change operations that complete with any kind of error cause the affected records to automatically revert to their values prior to the record change in JSDO memory (see General error handling for Data Object operations on page 55)

Four additional built-in properties are available for access on a JSDO:

- **`autoSort`** — A `boolean` property on a JSDO and its table references that indicates if record objects are sorted automatically at the completion of a supported JSDO operation that updates JSDO memory. In addition to setting this property to `true` (the default), in order to enable automatic sorting, you must invoke one or both of the `setSortFields( )` and `setSortFn( )` methods to specify how associated record objects are to be sorted—using specified sort fields or a user-defined sort function, respectively. You can set and reset this property at any time during the life of a JSDO to change what table references in JSDO memory are automatically sorted.
- **`caseSensitive`** — A `boolean` property on a JSDO and its table references that indicates if comparisons of `string` fields performed by supported JSDO operations on record objects in JSDO memory are case sensitive or case-insensitive for the affected table references. The setting of this property affects `string` comparisons **only** for merging record objects into JSDO memory using the `addRecords( )` method and for the sorting of record objects in JSDO memory, including automatic sorting using the `autoSort` property and manual sorting using the `sort( )` method. You can set and reset this property at any time during the life of a JSDO to change the case sensitivity of these supported `string` comparisons for selected table references in JSDO memory. However, note that any default `string` comparisons that **you** perform directly using JavaScript are case sensitive following JavaScript rules, regardless of the `caseSensitive` property setting.

- **name** — A `string` property that returns the name of the Data Object resource for which the JSDO is created. You must set this value either directly or indirectly using the `progress.data.JSDO` class constructor when you instantiate a JSDO. You can read the property on the JSDO after the JSDO is created.
- **useRelationships** — A `boolean` property that when set to `true` makes all data-relations active in JSDO memory so that searches on a child table reference involve only records related to its parent. In addition, record objects created in a child table have their key fields automatically set in relation to their parent. When set to `false`, searches on a child table reference involve all record objects stored in the JSDO table, regardless of data-relations, and any record objects created in a child table have no automatic settings for their key fields. You can set and reset this property at any time during the life of a JSDO to change the effect of data-relations on JSDO memory.

In addition to these JSDO built-in properties, you can set additional initialization properties in the class constructor, along with the JSDO `name` property itself. These initialization properties, which you cannot access after the JSDO is created, allow you to specify that certain JSDO methods are called during instantiation, including:

- Automatically calling the `fill( )` method to initialize JSDO memory as the object is created
- Subscribing handler callbacks to JSDO events, especially to the `beforeFill` and `afterFill` (or the `beforeRead` and `afterRead`) events in order to handle the automatic execution of the `fill( )` method

For more information on setting the JSDO `name` and initialization properties in the `JSDO` class constructor, see Creating and managing access to a JSDO instance on page 59.

Any JSDO also supports the creation of user-defined properties that you can use to store custom state information to help manage a given JSDO instance. You can create and access these user-defined properties using a set of JSDO built-in methods. Note that the JSDO reserves a subset of user-defined properties to support some of its behavior (currently, only the `"server.count"` user-defined property). For more information on working with user-defined properties see the descriptions of the following JSDO methods:

Note: Applies to Progress Data Objects Version 4.3 or later.

- setProperty( ) method on page 338
- setProperties( ) method on page 337
- getProperty( ) method on page 246
- getProperties( ) method on page 245

Requirements for using a JSDO

The basic requirements for using a JSDO include having network access to the Data Object resources that it reads and writes and setting up a JSDO login session to access these Data Object resources.

To use a JSDO in a mobile or web app, you need to ensure that the following actions have been completed:

1. Obtain the URI for the Data Object Service location (web application) and also the Data Service Catalog that supports each Data Object resource you want to access. For more information on supported Data Object Services and how to obtain location information for them, see App development options on page 14.
2. Create and initialize a JSDO login session between your mobile or web app and the specified web application. The standard way you can do this is to invoke the stand-alone function, `progress.data.getSession( )` (available in Progress Data Objects Version 4.3 or later), which returns a jQuery Promise for you to handle the associated network requests asynchronously.

Note: You can call this function as many times as required to create and initialize separate login sessions for additional web applications that host Data Object Services you need to access using a JSDO.

This function combines the work of the following login session actions and methods:

a. Creates an instance of the `progress.data.JSDOSession` class, which encapsulates the login session and supports asynchronous request management of the session using jQuery Promises as well as named events. You must ensure that you initialize the login session to use the same authentication model as the specified web application. For more information on coding for security considerations such as the web server authentication model, see Managing JSDO login sessions on page 95.

b. Calls the `login( )` method on the `JSDOSession` object with any credentials you pass to `getSession( )` in order to establish the login session for the specified web application.

c. Assuming that the login session is successfully established, the function calls the `addCatalog( )` method on the object to load the Data Service Catalog you have specified for a given Data Object Service. For more information, see Managing JSDO login sessions on page 95.

3. Once `getSession( )` successfully executes and initializes the login session with the specified Data Service Catalog, you can instantiate a `progress.data.JSDO` class to create the JSDO for any supported Data Object resource you need to access. For more information, see Creating and managing access to a JSDO instance on page 59.

Note: You can call `addCatalog( )` on the `JSDOSession` object returned by `getSession( )` in order to initialize the login session for additional Data Object Services supported by the same web application. Whether or not you initialize an existing login session to access additional Data Object Services, you must instantiate a separate JSDO for each Data Object resource you want to access.

Note: You can also create a `JSDOSession` by manually instantiating the `progress.data.JSDOSession` class. In this case, you must also manually invoke `login( )` and `addCatalog( )` on the `JSDOSession` instance before creating an associated JSDO.

4. With a JSDO instance, you can read, update, and write resource data, or execute resource business logic, as defined for the Data Object resource for which you have created the instance. For more information, see Accessing standard CRUD and Submit operations on page 61 and Accessing custom Invoke operations on page 89.

5. Once your mobile or web app is done with its JSDO instances and no longer needs access to the login sessions you have created using the `getSession( )` stand-alone function, you can terminate and invalidate access to all of them by invoking the the stand-alone function, `progress.data.invalidateAllSessions( )` (available in Progress Data Objects Version 4.4.1 or later). This function invokes the `invalidate( )` method on each `JSDOSession` object that is created and initialized in the app. The `invalidate( )` method, in turn, both invokes the `logout( )` method to terminate the associated login session and renders its `JSDOSession` object permanently unavailable (*invalidates* it) from creating additional login sessions using its `login( )` method. For more information, see Creating and managing access to a JSDO instance on page 59.

Note: You can invalidate multiple `JSDOSession` instances using `invalidateAllSessions( )` only if they were all created using the `getSession( )` stand-alone function. For any `JSDOSession` objects that you instantiate manually, you can only log them out or invalidate them permanently by calling `logout( )` or `invalidate( )` on each such `JSDOSession` instance manually.

Note: For some applications, instead of invoking the stand-alone `invalidateAllSessions( )` function, you might want to manually invoke **either** the `invalidate( )` method **or** the `logout( )` method on each `JSDOSession` object whose login session you want to terminate. For more information, see the notes on the progress.data.JSDOSession class on page 124.

For more information on the stand-alone functions for use with the JSDO framework, see the descriptions of the getSession( ) stand-alone function on page 248 and invalidateAllSessions( ) stand-alone function on page 258.

Creating and managing access to a JSDO instance

The following code fragment shows a simple JavaScript coding sequence for creating and using a JSDO referenced by the variable `jsdoCustomer` to read resource data. The entire fragment executes based on the successful creation and initialization of its login session using the `progress.data.getSession( )` stand-alone function, which returns a jQuery Promise:

```
var serviceURI = "http://oemobiledemo.progress.com/OEMobileDemoServicesForm/",
catalogURI =
"http://oemobiledemo.progress.com/OEMobileDemoServicesForm/static/CustomerService.json";

// create a JSDOSession for a specified web application
// that requires Form authentication
try {
    // create and initialize login session
    progress.data.getSession( {
            authenticationModel : progress.data.Session.AUTH_TYPE_FORM,
            serviceURI : serviceURI,
            catalogURI : catalogURI,
            username: "formuser",
            password: "formpassword"
    })
    .then(function( session, result ) { // 1

        // Create a JSDO and fill it with data
        var jsdoCustomer = new progress.data.JSDO('Customer');
        return jsdoCustomer.fill()

    }, function(result, info) {
        console.log("getSession failed");
    })
    .then(function (jsdo, result, request) { // 2
        console.log("Fill went okay!");
        // Do other JSDO related tasks here
    }, function (jsdo, result, request) {
        console.log("Not filling it today");
        // Do JSDO related failure stuff here
    })
    .always(function () { // 3
        // By the time we get here, we should be done with all JSDO related stuff
        // and cleaning up
        return progress.data.invalidateAllSessions();
    })
    .then(function( result, info ) {  // 4
        console.log("All sessions are invalidated!");
    }, function() {
        console.log("A session failed validation!");
    });
} catch (ex) {  // 5
    console.log("Exception: " + ex.message);
}
```

The fragment relies on a public Progress web application (`serviceURI`) that is protected using Form authentication and provides access to the Data Service Catalog (`catalogURI`) for the `CustomerService` Data Object Service. Once `getSession( )` returns its Promise, all further processing depends on results from a `try..catch` and a chain of Promise methods invoked as follows:

1. The initial Promise `then( )` method callbacks handle the success of the `JSDOSession` instantiation and login, where a successful resolution first invokes the instantiation of a JSDO instance (`jsdoCustomer`) for the `Customer` resource, then invokes the instance's `fill( )` to read the resource data and return the method's Promise with the results.
2. The second Promise `then( )` method callbacks handle the success of invoking the `fill( )` method, where the callback for a successful resolution invokes other JSDO tasks, such as presenting the data for viewing and update. This might also include the creation and initialization of additional login sessions that provide other Data Object Services for access by other JSDO instances.
3. After all processing based on the initial login session is complete (successful or not), the third Promise `always( )` method callback invokes the `progress.data.invalidateAllSessions( )` stand-alone function to clean up all login sessions that have been created in the chain, which also cleans up the JSDO instances for which they were created.
4. The fourth Promise `then( )` method callbacks handle the success of invoking the `invalidateAllSessions( )` stand-alone function.
5. As always, the `catch` block handles any exception thrown in the process of invoking methods, functions, and statements within the `try` block.

Note: If you know initially what login sessions you want to create and initialize using the `getSession( )` stand-alone function, you can start the Promise chain using a jQuery `when( )` method:

`$.when(`*`p1,p2,p3`*`. . .,`*`px`*`).then({function(){},function(){}}). . .`, where

`p1,p2,p3``. . .,`*`px`* are references to the Promises returned from one or more invocations of `getSession( )`. In this case, the `then( )` method success callback executes only if all the `getSession( )`-returned Promises that you pass to `when( )` successfully resolve. Otherwise, your JSDO processing is similar to using a single login session as described in the above example.

For more information on:

- **These stand-alone functions** — See the descriptions of the getSession( ) stand-alone function on page 248 and invalidateAllSessions( ) stand-alone function on page 258.
- **The `JSDOSession` methods that these functions invoke** — See the progress.data.JSDOSession class on page 124 description.
- **Instantiating a JSDO instance and the features it provides** — See the progress.data.JSDO class on page 110 description.
- **Implementing JSDO method calls to invoke operations on Data Object resources** — See Accessing standard CRUD and Submit operations on page 61 and Accessing custom Invoke operations on page 89.
- **Managing login sessions to support JSDO access to network resources** — See Managing JSDO login sessions on page 95.

Accessing standard CRUD and Submit operations

After creating a JSDO for a `dsCustomer` resource as explained in the preceding section, you can use standard JSDO methods to read, create, update, and delete records. The sections that follow provide guidance and examples for each of these operations.

Of the currently supported Progress Data Object Services (OpenEdge and Rollbase), only OpenEdge Data Object Services can require significant coding to implement server-side Data Objects. Therefore, the information on the JSDO in these examples refers to OpenEdge ABL used to implement an OpenEdge Data Object created for an OpenEdge DataSet (ProDataSet) resource named `dsCustomer`, which can support one or more OpenEdge temp-tables. For more information on using ABL to implement OpenEdge Data Objects, see the OpenEdge documentation on both using ABL and implementing Data Objects, including the information on Data Object Services in *OpenEdge Development: Web Services*. If you have no role in implementing OpenEdge Data Objects, or your JSDOs access only Rollbase Data Ojbects, you can ignore these ABL references.

For a Rollbase Data Object, each available resource is created for a single available Rollbase object, which is equivalent to a single database table, and the implementation of the Data Object itself is provided automatically by the Rollbase cloud server.

For an OpenEdge Data Object, the ABL methods in the following examples are from an OpenEdge Business Entity that is implemented as a singleton class with the following common code elements:

- **Inheritance —** The OpenEdge-defined abstract class, `OpenEdge.BusinessLogic.BusinessEntity`, which defines inherited ABL methods that the Business Entity calls to implement the standard Data Object CRUD and Submit operations on the `dsCustomer` resource with before-image support:
  - **Create —** `CreateData( )`, called by `CreatedsCustomer( )`
  - **Read —** `ReadData( )`, called by `ReaddsCustomer( )`
  - **Update —** `UpdateData( )`, called by `UpdatedsCustomer( )`
  - **Delete —** `DeleteData( )`, called by `DeletedsCustomer( )`
  - **Submit —** `SubmitData( )`, called by `SubmitdsCustomer( )`
- **Resource data model** — A before-image enabled ProDataSet defined with a single temp-table that is based on the Customer table of the OpenEdge-installed `sports2000` database:

Table 7: Data model for the `dsCustomer` resource

```
DEFINE TEMP-TABLE ttCustomer BEFORE-TABLE bttCustomer
  FIELD Address AS CHARACTER LABEL "Address"
  FIELD Balance AS DECIMAL INITIAL "0" LABEL "Balance"
  FIELD City AS CHARACTER LABEL "City"
  FIELD CustNum AS INTEGER INITIAL "0" LABEL "Cust Num"
  FIELD Name AS CHARACTER LABEL "Name"
  FIELD Phone AS CHARACTER LABEL "Phone"
  FIELD State AS CHARACTER LABEL "State"
  INDEX CustNum CustNum
  INDEX Name Name.

DEFINE DATASET dsCustomer FOR ttCustomer.
```

- **Additional Business Entity class definitions** — Where `customer.i` is an include file that defines the resource data model (see above) and the class supports before-imaging and all the standard Data Object operations (with all tool annotations removed):

  Table 8: Business Entity class class definitions

```
USING Progress.Lang.*.
USING OpenEdge.BusinessLogic.BusinessEntity.


BLOCK-LEVEL ON ERROR UNDO, THROW.


CLASS Customer INHERITS BusinessEntity:

  {"customer.i"}

  DEFINE DATA-SOURCE srcCustomer FOR Customer.

  CONSTRUCTOR PUBLIC Customer():

    DEFINE VAR hDataSourceArray AS HANDLE NO-UNDO EXTENT 1.
    DEFINE VAR cSkipListArray AS CHARACTER NO-UNDO EXTENT 1.

    SUPER(DATASET dsCustomer:HANDLE).

    /* Data Source for each table in dataset. Should be in table order
       as defined in DataSet. */
    hDataSourceArray[1] = DATA-SOURCE srcCustomer:HANDLE.

    /* Skip-list entry for each table in dataset. Should be in temp-table order
       as defined in DataSet. Each skip-list entry is a comma-separated list of
       field names, to be ignored in the ABL CREATE statement. */
    cSkipListArray[1] = "CustNum".

    THIS-OBJECT:ProDataSource = hDataSourceArray.
    THIS-OBJECT:SkipList = cSkipListArray.

  END CONSTRUCTOR.

  /* Methods to implement the
     standard Data Object operations . . . */

END CLASS.
```

This Business Entity class and its constructor define and initialize an ABL handle array (`hDataSourceArray`) with a `DATA-SOURCE` reference to the single OpenEdge database table (`Customer`) that provides the initial data for the `dsCustomer` ProDataSet. If `dsCustomer` contained multiple temp-tables, the constructor would initialize this array with a `DATA-SOURCE` reference for each database table that provides the data.

The constructor also defines and initializes a character array (`cSkipListArray`) that contains an array of comma-separated lists, each of which is a list of fields in each temp-table that and CUD and Submit operations should ignore, because their values are updated by database triggers or other server code. You need to specify as many lists (null strings, if empty) as there are temp-tables, and in the order that the temp-tables appear in the ProDataset.

Finally, the Business Entity constructor, first passes the ProDataSet `dsCustomer` handle to its super constructor and sets the two super-class-defined properties, `ProDataSource` and `SkipList` to the two arrays that it has initialized. The Business Entity then defines the ABL methods that implement the supported Data Object operations, which you can see described under the following topics.

Note: The OpenEdge Business Entity described, here, uses the most basic OpenEdge features to implement an OpenEdge resource with before-imaging, which a JSDO can access in most client environments other than the Telerik Platform. If you are using the Telerik Platform to build a mobile app or want to access the Business Entity as a Rollbase external object, this Business Entity must be revised as described for updating Business Entities for access by Telerik DataSources and Rollbase external objects in *OpenEdge Development: Web Services*.

To access the standard Data Object operations on a given resource using the JSDO, a client typically follows an iterative procedure that includes these general steps:

1. Reads resource data into JSDO memory using the `fill( )` method to send a Read operation across the network along with optional selection criteria passed as a parameter according to Data Object resource requirements and the platform used to implement the client application. The client handles the results using JSDO events or Promises, depending on the availability of Promises and client application requirements.
2. If the resource is writable (not read-only), creates, updates, and deletes any number of record objects in JSDO memory calling the JSDO `add( )`, `assign( )` (or its equivalent), and `remove( )` methods, respectively. This marks each affected record in JSDO memory as a pending record create, update, or delete. For more information, see the description of the add( ) method on page 158, assign( ) method (JSDO class) on page 198, or remove( ) method on page 311.
3. Synchronizes any pending changes in JSDO memory with the corresponding server Data Object and its database, depending on the type of Progress Data Object Service and its implementation. The client invokes this data synchronization by calling the JSDO `saveChanges( )` method in one of the following ways, depending on the Data Object resource implementation:

   - If the resource **does not** support Submit, the client calls `saveChanges( )` with either a single argument of `false` or an empty parameter list (the default). This call sends one or more Create, Update, or Delete (CUD) operation requests across the network, with one operation request sent across the network for each pending record change in JSDO memory. Thus, the respective CUD operation on the server implements each pending record change, with results returned for each operation request in its own network response.
   - If the resource supports Submit, the client typically calls `saveChanges( )` with a single argument of `true`. This sends a single Submit operation request across the network for all pending record changes in JSDO memory. Thus, all pending CUD record changes on the server are implemented by this single Submit operation, with all record-change results returned in a single network response.

     Note: When the resource supports Submit, the client can instead invoke `saveChanges(false)` to send CUD operations individually across the network. In this case, the synchronization of record changes works the same as if the resource does not support Submit, as described above.

4. Handles the results from the call to `saveChanges( )` using JSDO events or Promises, depending on the availability of Promises and client application requirements.

For more information, see the following topics:

- Read operation example on page 64
- Create operation example on page 68
- Update operation example on page 72
- Delete operation example on page 78
- Submit operation example on page 83

Read operation example

To load data into JSDO memory on the client, you call the `fill( )` method on the JSDO, passing an optional parameter to specify selection criteria as either an object or a string. This executes the ABL or Rollbase server routine that implements the Data Object Read operation. Each time `fill( )` is called, all records currently in JSDO memory are cleared and replaced by the records returned by the method.

When you access a Progress Data Service from the Telerik Platform and the JSDO dialect of the Kendo UI DataSource calls `fill( )` on its JSDO, the DataSource passes an object parameter to the method. This object contains properties specified according to DataSource requirements to select, order, and organize the return of the records from the server. The Data Object resource on the server then receives the DataSource selection criteria specified by this object in the form of a Progress-defined JSON Filter Pattern (JFP). Note that the resource Read operation must be programmed to handle this JFP according to the requirements of the specific Progress Data Object Service (OpenEdge or Rollbase). For more information, see the description of the fill( ) method on page 220.

If you are calling this method yourself, you can use a string to specify the selection criteria, again, according to the requirements of the Data Object resource whose data you are reading. If you do not pass a parameter to `fill( )`, the records returned to the client depend entirely on the implementation of the resource Read operation. For more information, see the description of the fill( ) method on page 220.

When you call the `fill( )` method, if jQuery Promises are supported in your environment, it returns a Promise object on which you can register callbacks to handle completion of the Read operation that it executes. Otherwise, you can subscribe event callbacks to handle the results of the Read operation:

Operation results are thus returned as follows:

1. The JSDO fires an `afterFill` event for any callbacks you have subscribed to handle the event.
2. Any returned Promise object executes the Promise callbacks that you have registered, depending on the operation results.

Note that before any callbacks execute, if the Read operation completes successfully, the working record for each JSDO table reference is set to its first record returned, depending on any active parent-child relationships. Thus for each child table reference, the first record is determined by its relationship to the related working record in its parent.

For more information on handling `fill( )` method results, see the description of the fill( ) method on page 220.

Following is an example showing an OpenEdge ABL implementation of the Read operation and its invocation on a JSDO.

Note: The **`bold`** code in the JavaScript examples primarily trace the path of referenced JSDO method and callback parameters, as well as key code elements in the example, including those elements that are directly or indirectly referenced in the example description and notes.

An OpenEdge ABL implementation for Read

For an OpenEdge Data Object resource, the ABL routine that implements a Read operation must have the following signature:

- An `INPUT` parameter of type `CHARACTER`, named `filter`. This parameter is optional for the `fill( )` call that initiates the Read operation; if specified, it defines the criteria for a filtered subset of records to be returned. The format and content of the filter string depend on the application, including the requirements of the OpenEdge Business Entity that implements the resource. However, if the Business Entity implements a Data Object resource for access by the Telerik Platform or as a Rollbase external object, this filter string

always follows the format of the Progress-defined JSON Filter Pattern. For more information, see the sections on updating Business Entities for access by Telerik DataSources and Rollbase external objects in *OpenEdge Development: Web Services*.

- An `OUTPUT` parameter for either a `DATASET`, `DATASET-HANDLE`, `TABLE` or `TABLE-HANDLE`. If the resource supports before-image data, the parameter can only be for a `DATASET` or `DATASET-HANDLE`.

The following example shows a `ReaddsCustomer( )` method that might implement the Data Object Read operation for the ProDataSet resource, `dsCustomer`:

```
  METHOD PUBLIC VOID ReaddsCustomer(INPUT filter AS CHARACTER, OUTPUT DATASET
dsCustomer):

    SUPER:ReadData(filter).

  END METHOD.
```

This implementation simply calls an overload of the `ReadData( )` method defined in the `OpenEdge.BusinessLogic.BusinessEntity` super class, which OpenEdge provides as an aid for implementing Data Object resources that support before-imaging. The invoked `ReadData( )` method provides a default implementation for using the `filter` parameter to identify the data from the server database that fills the specified ProDataSet.

Client JavaScript code: Read

The following examples illustrate calling `fill( )` on the `dsCustomer` JSDO with the single-table resource, `ttCustomer`, to load records from the server database into JSDO memory and using a returned Promise object to handle the results.

This example uses simple error handling to log the contents of each error object returned by the JSDO `getErrors( )` method:

```
var strFilter = 'where CustNum < 100';

dsCustomer.fill(strFilter).done(
    function( jsdo, success, request ) {
        /* for example, add code to display all records in a list */
        jsdo.foreach(function (jsrecord) {
            /* you can reference the fields as jsrecord.data.field . . . */
        });
    }).fail(
    function( jsdo, success, request ) {
        var lenErrors,
        errors,
        errorType;

        /* handle Read operation errors */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) { /* Each error */
            console.log(JSON.stringify(errors[idxError]));
        }
    }
});
```

This example is identical to the previous one but with more complex error handling to log the contents of each error object using more readable output:

```
var strFilter = 'where CustNum < 100';

dsCustomer.fill(strFilter).done(
    function( jsdo, success, request ) {
        /* for example, add code to display all records in a list */
        jsdo.foreach(function (jsrecord) {
            /* you can reference the fields as jsrecord.data.field . . . */
        });
    }).fail(
    function( jsdo, success, request ) {
        var lenErrors,
        errors,
        errorType;

        /* handle Read operation errors */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) { /* Each error */
            switch(errors[idxError].type) {
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
            }
            console.log("READ ERROR: " + errorType + errors[idxError].error);
            if (errors[idxError].responseText) {
                console.log("HTTP FULL TEXT: " + errors[idxError].responseText);
            }
        }
    }
});
```

This sample code:

- Passes a filter parameter, `strFilter`, to the `fill( )` method. This filter causes the method to load only those records with a `CustNum` value lower than 100.
- Calls `done( )` on the returned Promise object to register a callback function that processes the returned records after successful invocation of the Read operation, and calls `fail( )` on the returned Promise to register a callback function that processes the errors after an unsuccessful invocation of the Read operation.

Note: A reference to the same Promise object is returned by each Promise method called in a chain on the `fill( )` method to register the callbacks. You can also save a reference to the Promise object returned by `fill( )` and call each Promise method on it this reference in sequence.

The following example illustrates calling `fill( )` on the `dsCustomer` JSDO to load records from the server database into JSDO memory and using the `afterFill` event to handle the results with the same complex error handling as in the previous example:

```
var strFilter = 'where CustNum < 100';

/* subscribe to event */
dsCustomer.subscribe('afterFill', onAfterFill);

dsCustomer.fill(strFilter);

function onAfterFill(jsdo , success , request ) {
    var lenErrors,
        errors,
        errorType;

    if (success) {
        /* for example, add code to display all records in a list */
        jsdo.foreach(function (jsrecord) {
            /* the code here is executed for each record on the table.
               you can reference the fields as jsrecord.data.field . . . */
        });
    }
    else {
        /* handle Read operation errors */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) { /* Each error */
            switch(errors[idxError].type) {
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
            }
            console.log("READ ERROR: " + errorType + errors[idxError].error);
            if (errors[idxError].responseText) {
                console.log("HTTP FULL TEXT: " + errors[idxError].responseText);
            }
        }
    }
};
```

This sample code:

- Passes a filter parameter, `strFilter`, to the `fill( )` method. This filter causes the method to load only those records with a `CustNum` value lower than 100.
- Subscribes a callback function, `onAfterFill`, to the `afterFill` event to enable processing of the returned records on successful invocation of the Read operation and handling of the returned errors on unsuccessful invocation of the Read operation.

Create operation example

To create a new record in JSDO memory on the client, you call the `add( )` method on a JSDO table reference. The fields of the new record are initialized with the values specified in an object that you pass to the method. For any fields whose values you do not provide in this object, the method provides default values taken from the Data Object schema in the Data Service Catalog.

When each invocation of the `add( )` method completes, the new record becomes the working record for the associated table and the JSDO marks this record as a new record for pending creation on the server. (Note that if the table has child tables, a working record is not set for these child tables.)

To synchronize the server database with new records you have created in JSDO memory **without** using a Submit operation, you call `saveChanges( )` by passing either an empty parameter list to the method or a single parameter value of `false`. This executes the OpenEdge ABL or Rollbase server routine that implements the Data Object Create operation once for each newly created record in JSDO memory. In addition, for any other changed records in JSDO memory, this call also executes the server routine that implements the associated Delete or Update operation once for each associated record change.

Note: When multiple pending record changes exist in JSDO memory, the `saveChanges( )` method invoked without Submit groups invocation of the associated resource operations in the order of 1) all Delete operations, 2) all Create operations, and 3) all Update operations, and invokes each such operation one record at a time over the network.

When you call the `saveChanges( )` method, if jQuery Promises are supported in your environment, it returns a Promise object on which you can register callbacks to handle completion of all record-change operations that it has invoked. Otherwise, you can subscribe event callbacks to handle the results of these operations.

Operation results are thus returned as follows:

1. For each Create operation that completes, the JSDO fires an `afterCreate` event to execute any callbacks you have subscribed to handle that event. The JSDO also fires any `afterDelete` and `afterUpdate` events to execute any callbacks you have subscribed to handle these events.
2. After all record-change operations complete, the JSDO fires an `afterSaveChanges` event to execute any callbacks you have subscribed to handle that event. In addition, any returned Promise object executes the Promise callbacks that you have registered, depending on the combined operation results. (Note that the signatures of all Promise callbacks are identical to the signature of the `afterSaveChanges` event callback.)

After the `saveChanges( )` method and all resource operations that it has invoked successfully complete, no working records are set for the tables in the JSDO.

Note: If successful execution of a resource Create or Update operation results in changes to the record on the server that was sent from the client (for example, an update to a sequence value), JSDO memory is automatically synchronized with these server changes when the request object with these results is returned to the client.

Note: If an error occurs on the server, and the `autoApplyChanges` property has the default value of `true`, any newly created record is automatically deleted from JSDO memory on the client.

For more information on handling `saveChanges( )` method results for individual record-change operations, see the description of the saveChanges( ) method on page 314.

Following is an example showing an OpenEdge ABL implementation of the Create operation and its invocation on a JSDO.

Note: The **bold** code in the JavaScript examples primarily trace the path of referenced JSDO method and callback parameters, as well as key code elements in the example, including those elements that are directly or indirectly referenced in the example description and notes.

An OpenEdge ABL implementation for Create

For an OpenEdge Data Object resource, the signature of the ABL routine that implements a Create operation must have a single `INPUT-OUTPUT` parameter for a `DATASET`, `DATASET-HANDLE`, `TABLE` or `TABLE-HANDLE`. If the resource supports before-image data, the parameter can only be for a `DATASET` or `DATASET-HANDLE`.

The following example shows a `CreatedsCustomer( )` method that might implement the Data Object Create operation for the ProDataSet resource, `dsCustomer`:

```
METHOD PUBLIC VOID CreatedsCustomer(INPUT-OUTPUT DATASET dsCustomer):

  DEFINE VAR hDataSet AS HANDLE NO-UNDO.
  hDataSet = DATASET dsCustomer:HANDLE.

  SUPER:CreateData(DATASET-HANDLE hDataSet BY-REFERENCE).

END METHOD.
```

This implementation simply calls an overload of the `CreateData( )` method defined in the `OpenEdge.BusinessLogic.BusinessEntity` super class, which OpenEdge provides as an aid for implementing Data Object resources that support before-imaging. The invoked `CreateData( )` method provides a default implementation to create a new record in the server database from the single temp-table record in the ProDataSet input to `CreatedsCustomer( )`.

Client JavaScript code: Create

The following examples illustrate calling `add( )` on a table reference to create a record in JSDO memory, then calling `saveChanges( )` on the `dsCustomer` JSDO without Submit to add the new record to the server database (along with any other pending record-change operations), and using both an `afterCreate` event handler and a returned Promise object to handle the results.

This example uses simple error handling to log the contents of each error object returned by the JSDO `getErrors( )` method:

```
/* subscribe to event */
dsCustomer.ttCustomer.subscribe('afterCreate', onAfterCreate);

/* some code that adds a record and sends it to the server */
var jsrecord = dsCustomer.ttCustomer.add( {State : 'MA'} );

/* some other JSDO memory record changes . . . */

dsCustomer.saveChanges(false).done( /* Successful method execution */
    function( jsdo, success, request ) {
        /* all resource operations invoked by saveChanges() succeeded */
        /* for example, process all records based on their State value */
        jsdo.ttCustomer.foreach( function(jsrecord) {
            if (jsrecord.data.State === 'MA') {
                /* process all records for the state of MA . . . */
            }
            if (jsrecord.data.State === 'VT') {
                /* process all records for the state of VT . . . */
```

```
            }
        });

    }).fail( /* Unsuccessful method execution */
    function( jsdo, success, request ) {
        /* one or more resource operations invoked by saveChanges() failed */
        var lenErrors,
            errors,
            errorType;

        /* handle all operation errors */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) { /* Each error */
            console.log(JSON.stringify(errors[idxError]));
        }
    });

function onAfterCreate (jsdo , record , success , request ) {
    /* check for errors on any Create operation */
    if (success) {
        /* do the normal thing for a successful Create operation.
           for example, process new record according to its State value  */
        switch (record.data.State ) {
            case 'MA':
                /* process a record created for MA . . . */
                break;
            case 'VT':
                /* processs a record created for VT . . . */
                break;
            default:
                /* process a record created for any other state . . . */
                break;
        }
    }
    else {
        /* all error messages handled by the Promise.fail() callback */
    }
};
```

This example is identical to the previous one but with more complex error handling to log the contents of each error object using more readable output:

```
/* subscribe to event */
dsCustomer.ttCustomer.subscribe('afterCreate', onAfterCreate);

/* some code that adds a record and sends it to the server */
var jsrecord = dsCustomer.ttCustomer.add( {State : 'MA'} );

/* some other JSDO memory record changes . . . */

dsCustomer.saveChanges(false).done( /* Successful method execution */
    function( jsdo, success, request ) {
        /* all resource operations invoked by saveChanges() succeeded */
        /* for example, process all records based on their State value */
        jsdo.ttCustomer.foreach( function(jsrecord) {
            if (jsrecord.data.State === 'MA') {
                /* process all records for the state of MA . . . */
            }
            if (jsrecord.data.State === 'VT') {
                /* process all records for the state of VT . . . */
            }
        });
```

```
    }).fail( /* Unsuccessful method execution */
    function( jsdo, success, request ) {
        /* one or more resource operations invoked by saveChanges() failed */
        var lenErrors,
            errors,
            errorType;

        /* handle all operation errors */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) { /* Each error */
            switch(errors[idxError].type) {
                case progress.data.JSDO.DATA_ERROR:
                    errorType = "Server Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
                case default:
                    errorType = null; // Unexpected errorType value
                    break;
            }
            if (errorType) { /* log all error text
                console.log("ERROR: " + errorType + errors[idxError].error);
                if (errors[idxError].id) { /* error with record object */
                    console.log("RECORD ID: " + errors[idxError].id);
                    /* possibly log the data values for record with this ID */
                }
                if (errors[idxError].responseText) {
                    console.log("HTTP FULL TEXT: "
                                + errors[idxError].responseText);
                }
            }
            else { /* unexpected errorType */
                console.log("UNEXPECTED ERROR TYPE: "
                            + errors[idxError].type);
            }
        }
    });

function onAfterCreate (jsdo , record , success , request ) {
    /* check for errors on any Create operation */
    if (success) {
        /* do the normal thing for a successful Create operation.
           for example, process new record according to its State value  */
        switch (record.data.State ) {
            case 'MA':
                /* process a record created for MA . . . */
                break;
            case 'VT':
                /* processs a record created for VT . . . */
                break;
            default:
                /* process a record created for any other state . . . */
                break;
        }
    }
    else {
        /* all error messages handled by the Promise.fail() callback */
    }
};
```

This sample code:

- Subscribes a callback function, `onAfterCreate`, to the `afterCreate` event to enable error-checking and manipulation of the new record after the Create operation completes on the server. This includes client processing of the record based on its `State` field value. This callback ignores and defers all error handling to the `fail()` callback registered on the Promise returned by `saveChanges( )`. (Typically, you code similar `afterUpdate` and `afterDelete` event callbacks for the app as required.)
- Adds a record to the `ttCustomer` table, with an initial value of `'MA'` for the `State` field. (**Note:** The table reference can be omitted if `ttCustomer` is the only temp-table in the `dsCustomer` ProDataSet.)
- Calls `saveChanges(false)` (without Submit) to invoke all pending Data Object operations one record at a time on the server, including creation of the new record with its `State` field set to `'MA'`, and thereby synchronizes the content of JSDO memory with the server database for one record-change operation at a time across the network. A returned Promise object handles the overall success or failure of `saveChanges( )` after all pending Create, Update, and Delete (CUD) operations complete, which includes successful processing of all client records based on their `State` field value.

  For a successful `saveChanges( )` invocation without Submit, the `done()` callback processes all client records for selected `State` field values. For an unsuccessful `saveChanges( )` invocation, the `fail()` callback handles all error messages returned in response to invoking each CUD operation across the network.

  Note: This example takes advantage of the default setting of the JSDO `autoApplyChanges` property (`true`), which automatically accepts or rejects changes to JSDO memory based on whether the CUD operation was successful. Note also that errors returned by `getErrors( )` remain available until the next call to `fill( )` or `saveChanges( )`.

Note: Any event handler callback functions (such as `onAfterCreate`) always execute **before** any registered Promise callback methods (such as `done()`). For an invocation of `saveChanges( )` without Submit, the event callbacks execute after each CUD operation completes and returns its results from the server, and the appropriate registered Promise callbacks execute only after **all** results from these CUD operations have been returned from the server, which allows the results of all CUD operations to be processed together on the client.

Update operation example

To modify an existing record in JSDO memory on the client, you can call the `assign( )` method to assign values to one or more fields of either a working record or a specific record in JSDO memory, or you can assign a value directly to the field of a working record. Using the `assign( )` method, you set the values of fields to be updated in an object that you pass as a parameter to the method.

You can call the `assign( )` method on:

- A table reference on the JSDO that has a working record, as in the example below
- A specific `progress.data.JSRecord` object reference

When the `assign( )` method completes, any working record settings that existed prior to calling the method remain unchanged.

You can also assign a value directly to a single field of a JSDO working record as follows:

Syntax

```
jsdo-ref.table-ref.field-ref = value;
jsdo-ref.field-ref = value;
```

Where:

jsdo-ref

The reference to a JSDO, and if the JSDO contains only a single table, an implied reference to any working record that is set for that table.

table-ref

A table reference with the name of a table in *jsdo-ref* memory that has a working record.

field-ref

A field reference on a *table-ref* with the name and value of a field in the working record of the referenced table.

value

The value to set the field referenced by *field-ref*.

After either a successful call to the `assign( )` method or a successful assignment to a *field-ref* of a working record, the JSDO marks the affected record for pending update on the server.

Caution: Never write directly to a *field-ref* that you reference on the `data` property of a *table-ref*; use *field-ref* on the `data` property **only to read** the referenced field value. Writing field values using the `data` property does **not** mark the record for pending update on the server, nor does it initiate a re-sort the record in JSDO memory according to any order you have established using the `autoSort` property. For more information, see the description of the data property on page 217. To mark a record for pending update and automatically re-sort the record according to the `autoSort` property, you must assign a record field value using one of the mechanisms described above.

To synchronize the server database with existing records you have updated in JSDO memory **without** using a Submit operation, you call `saveChanges( )` by passing either an empty parameter list to the method or a single parameter value of `false`. This executes the OpenEdge ABL or Rollbase server routine that implements the Data Object Update operation once for each updated record in JSDO memory. In addition, for any other changed records in JSDO memory, this call also executes the server routine that implements the associated Create or Delete operation once for each associated record change.

Note: When multiple pending record changes exist in JSDO memory, the `saveChanges( )` method invoked without Submit groups invocation of the associated resource operations in the order of 1) all Delete operations, 2) all Create operations, and 3) all Update operations, and invokes each such operation one record at a time over the network.

When you call the `saveChanges( )` method, if jQuery Promises are supported in your environment, it returns a Promise object on which you can register callbacks to handle completion of all record-change operations that it has invoked. Otherwise, you can subscribe event callbacks to handle the results of these operations.

Operation results are thus returned as follows:

1. For each Update operation that completes, the JSDO fires an `afterUpdate` event to execute any callbacks you have subscribed to handle that event. The JSDO also fires any `afterCreate` and `afterDelete` events to execute any callbacks you have subscribed to handle these events.
2. After all record-change operations complete, the JSDO fires an `afterSaveChanges` event to execute any callbacks you have subscribed to handle that event. In addition, any returned Promise object executes the Promise callbacks that you have registered, depending on the combined operation results. (Note that the signatures of all Promise callbacks are identical to the signature of the `afterSaveChanges` event callback.)

After the `saveChanges( )` method and all resource operations that it has invoked successfully complete, no working records are set for the tables in the JSDO.

Note: If successful execution of a resource Create or Update operation results in changes to the record on the server that was sent from the client (for example, an update to a sequence value), JSDO memory is automatically synchronized with these server changes when the request object with these results is returned to the client.

Note: If an error occurs on the server, and the `autoApplyChanges` property has the default value of `true`, any updated record has its changes automatically backed out from JSDO memory on the client.

For more information on handling `saveChanges( )` method results for individual record-change operations, see the description of the saveChanges( ) method on page 314.

Following is an example showing an OpenEdge ABL implementation of the Update operation and its invocation on a JSDO.

Note: The **`bold`** code in the JavaScript examples primarily trace the path of referenced JSDO method and callback parameters, as well as key code elements in the example, including those elements that are directly or indirectly referenced in the example description and notes.

An OpenEdge ABL implementation for Update

For an OpenEdge Data Object resource, the signature of the ABL routine that implements an Update operation must have a single `INPUT-OUTPUT` parameter for a `DATASET`, `DATASET-HANDLE`, `TABLE` or `TABLE-HANDLE`. If the resource supports before-image data, the parameter can only be for a `DATASET` or `DATASET-HANDLE`.

The following example shows a `UpdatedsCustomer( )` method that might implement the Data Object Update operation for the ProDataSet resource, `dsCustomer`:

```
METHOD PUBLIC VOID UpdatedsCustomer(INPUT-OUTPUT DATASET dsCustomer):

  DEFINE VAR hDataSet AS HANDLE NO-UNDO.
  hDataSet = DATASET dsCustomer:HANDLE.

  SUPER:UpdateData(DATASET-HANDLE hDataSet BY-REFERENCE).

END METHOD.
```

This implementation simply calls an overload of the `UpdateData( )` method defined in the `OpenEdge.BusinessLogic.BusinessEntity` super class, which OpenEdge provides as an aid for implementing Data Object resources that support before-imaging. The invoked `UpdateData( )` method provides a default implementation to update an existing record in the server database from the single temp-table record in the ProDataSet input to `UpdatedsCustomer( )`.

Client JavaScript code: Update

The following examples illustrate calling `assign( )` on a table reference in the JSDO to update a record in JSDO memory, then calling `saveChanges( )` on the `dsCustomer` JSDO without Submit to update the corresponding record in the server database (along with any other pending record-change operations), and using both an `afterUpdate` event handler and a returned Promise object to handle the results.

This example uses simple error handling to log the contents of each error object returned by the JSDO `getErrors( )` method:

```
/* subscribe to event */
dsCustomer.ttCustomer.subscribe('afterUpdate', onAfterUpdate);

/* some code that updates a record and sends it to the server */
var jsrecord = dsCustomer.ttCustomer.find(function(jsrecord) {
  return (jsrecord.data.Name === 'Lift Tours');
});
if (jsrecord) {jsrecord.assign( {State: 'VT'} );}

/* some other JSDO memory record changes . . . */

dsCustomer.autoApplyChanges = false;
dsCustomer.saveChanges(false).done( /* Successful method execution */
    function( jsdo, success, request ) {
        /* all resource operations invoked by saveChanges() succeeded */
        /* for example, track the average customer balance in certain states */
        var avgBalance = { MA : 0.0, maCount : 0,
                           VT : 0.0, vtCount : 0 };
        jsdo.ttCustomer.foreach( function(jsrecord) {
            if (jsrecord.data.State === 'MA') {
                /* sum balances for customers in MA . . . */
                avgBalance.MA += jsrecord.data.Balance;
                avgBalance.maCount += 1;
            }
            if (jsrecord.data.State === 'VT') {
                /* sum balances for customers in VT . . . */
                avgBalance.VT += jsrecord.data.Balance;
                avgBalance.vtCount += 1;
            }
        });
        /* compute averages and process further . . . */
        avgBalances.MA = avgBalances.MA / avgBalances.maCount;
        avgBalances.VT = avgBalances.VT / avgBalances.vtCount;

    }).fail( /* Unsuccessful method execution */
    function( jsdo, success, request ) {
        /* one or more resource operations invoked by saveChanges() failed */
        var lenErrors,
            errors,
            errorType;

        /* handle all operation errors */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) { /* Each error */
            console.log(JSON.stringify(errors[idxError]));
        }
    });

function onAfterUpdate (jsdo , record , success , request ) {
    /* check for errors on any Update operation */
    if (success) {
        /* do the normal thing for a successful Update operation.
           for example, process updated record according to its Balance value  */
        if (record.data.Balance > $100000.00) {
```

```
            /* process a high balance condition . . . */
        }
        record.acceptRowChanges();
    }
    else {
        /* all error messages handled by the Promise.fail() callback */
        record.rejectRowChanges();
    }
};
```

This example is identical to the previous one but with more complex error handling to log the contents of each error object using more readable output:

```
/* subscribe to event */
dsCustomer.ttCustomer.subscribe('afterUpdate', onAfterUpdate);

/* some code that updates a record and sends it to the server */
var jsrecord = dsCustomer.ttCustomer.find(function(jsrecord) {
  return (jsrecord.data.Name === 'Lift Tours');
});
if (jsrecord) {jsrecord.assign( {State: 'VT'} );};

/* some other JSDO memory record changes . . . */

dsCustomer.autoApplyChanges = false;
dsCustomer.saveChanges(false).done( /* Successful method execution */
    function( jsdo, success, request ) {
        /* all resource operations invoked by saveChanges() succeeded */
        /* for example, track the average customer balance in certain states */
        var avgBalance = { MA : 0.0, maCount : 0,
                           VT : 0.0, vtCount : 0 };
        jsdo.ttCustomer.foreach( function(jsrecord) {
            if (jsrecord.data.State === 'MA') {
                /* sum balances for customers in MA . . . */
                avgBalance.MA += jsrecord.data.Balance;
                avgBalance.maCount += 1;
            }
            if (jsrecord.data.State === 'VT') {
                /* sum balances for customers in VT . . . */
                avgBalance.VT += jsrecord.data.Balance;
                avgBalance.vtCount += 1;
            }
        });
        /* compute averages and process further . . . */
        avgBalances.MA = avgBalances.MA / avgBalances.maCount;
        avgBalances.VT = avgBalances.VT / avgBalances.vtCount;

    }).fail( /* Unsuccessful method execution */
    function( jsdo, success, request ) {
        /* one or more resource operations invoked by saveChanges() failed */
        var lenErrors,
            errors,
            errorType;

        /* handle all operation errors */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) { /* Each error */
            switch(errors[idxError].type) {
                case progress.data.JSDO.DATA_ERROR:
                    errorType = "Server Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
```

```
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
                case default:
                    errorType = null; // Unexpected errorType value
                    break;
            }
            if (errorType) { /* log all error text
                console.log("ERROR: " + errorType + errors[idxError].error);
                if (errors[idxError].id) { /* error with record object */
                    console.log("RECORD ID: " + errors[idxError].id);
                    /* possibly log the data values for record with this ID */
                }
                if (errors[idxError].responseText) {
                    console.log("HTTP FULL TEXT: "
                                + errors[idxError].responseText);
                }
            }
            else { /* unexpected errorType */
                console.log("UNEXPECTED ERROR TYPE: "
                            + errors[idxError].type);
            }
        }
    });

function onAfterUpdate (jsdo , record , success , request ) {
    /* check for errors on any Update operation */
    if (success) {
        /* do the normal thing for a successful Update operation.
           for example, process updated record according to its Balance value  */
        if (record.data.Balance > $100000.00) {
            /* process a high balance condition . . . */
        }
        record.acceptRowChanges();
    }
    else {
        /* all error messages handled by the Promise.fail() callback */
        record.rejectRowChanges();
    }
};
```

This sample code:

- Subscribes a callback function, `onAfterUpdate`, to the `afterUpdate` event to enable error-checking and manipulation of the updated record after the Update operation completes on the server, which includes client processing of the record based on a high balance condition. This callback accepts or rejects changes to each record based on the success of the corresponding operation. Otherwise, it ignores and defers all error message handling to the `fail()` callback registered on the Promise returned by `saveChanges( )`. (Typically, you code similar `afterCreate` and `afterDelete` event callbacks for the app as required.)
- Finds a record in the `ttCustomer` table with the customer name, `'Lift Tours'`, which has an initial `State` field value of `'MA'`, and changes its state to `'VT'`. (**Note:** The table reference can be omitted if `ttCustomer` is the only temp-table in the `dsCustomer` ProDataSet.)
- Sets `autoApplyChanges` on the `dsCustomer` JSDO to `false`. Setting `autoApplyChanges` to `false` allows each Update or other record-change operation attempted by `saveChanges( )` to be identified and handled, either after each operation returns its response to the client using an event callback (as in the example) or after all operations have returned their responses using a Promise callback (also, as in the example). However, you must manually invoke the appropriate JSDO method to accept or reject the changes

in JSDO memory, depending on the operation success and its requirements. In this example, the operation event callback accepts or rejects the changes for each record, based on the operation success.

- Calls `saveChanges(false)` (without Submit) to invoke all pending Data Object operations one record at a time on the server, including update of the existing record with the customer name, `'Lift Tours'`. It thereby synchronizes the content of JSDO memory with the server database for one record-change operation at a time across the network. A returned Promise object handles the overall success or failure of `saveChanges( )` after all pending Create, Update, and Delete (CUD) operations complete.

  For a successful `saveChanges( )` invocation without Submit, the `done()` callback averages the balances of all client records for selected `State` field values. For an unsuccessful `saveChanges( )` invocation, the `fail()` callback handles all error messages returned in response to invoking each CUD operation across the network.

  Note: For unsuccessful operation results in this example, record changes are rejected before any associated error information is processed. However, the error messages for all operations remain available to the `getErrors( )` method until the next call to `fill( )` or `saveChanges( )`.

Note: Any event handler callback functions (such as `onAfterUpdate`) always execute **before** any registered Promise callback methods (such as `done()`). For an invocation of `saveChanges( )` without Submit, the event callbacks execute after each CUD operation completes and returns its results from the server, and the appropriate registered Promise callbacks execute only after **all** results from these CUD operations have been returned from the server. This allows the results of all CUD operations to be processed together on the client. Although this example uses Promise callbacks in addition to operation event callbacks, all accepting and rejecting of JSDO memory changes is done in the operation event callbacks.

Delete operation example

To delete an existing record from JSDO memory on the client, you call the `remove( )` method on a JSDO table reference.

You can call the `remove( )` method on:

- A table reference on the JSDO that has a working record
- A specific `progress.data.JSRecord` object reference, as in the example below

When the `remove( )` method completes, no working record is set for the associated table or any of its child tables.

After either a successful call to the `remove( )` method, the JSDO marks the affected record for pending deletion on the server.

To synchronize the server database with existing records you have deleted in JSDO memory **without** using a Submit operation, you call `saveChanges( )` by passing either an empty parameter list to the method or a single parameter value of `false`. This executes the OpenEdge ABL or Rollbase server routine that implements the Data Object Update operation once for each deleted record in JSDO memory. In addition, for any other changed records in JSDO memory, this call also executes the server routine that implements the associated Create or Update operation once for each associated record change.

Note: When multiple pending record changes exist in JSDO memory, the `saveChanges( )` method invoked without Submit groups invocation of the associated resource operations in the order of 1) all Delete operations, 2) all Create operations, and 3) all Update operations, and invokes each such operation one record at a time over the network.

When you call the saveChanges( ) method, if jQuery Promises are supported in your environment, it returns a Promise object on which you can register callbacks to handle completion of all record-change operations that it has invoked. Otherwise, you can subscribe event callbacks to handle the results of these operations.

Operation results are thus returned as follows:

1. For each Delete operation that completes, the JSDO fires an afterDelete event to execute any callbacks you have subscribed to handle that event. The JSDO also fires any afterCreate and afterUpdate events to execute callbacks you have subscribed to handle these events.
2. After all record-change operations complete, the JSDO fires an afterSaveChanges event to execute any callbacks you have subscribed to handle that event. In addition, any returned Promise object executes the Promise callbacks that you have registered, depending on the combined operation results. (Note that the signatures of all Promise callbacks are identical to the signature of the afterSaveChanges event callback.)

After the saveChanges( ) method and all resource operations that it has invoked successfully complete, no working records are set for the tables in the JSDO.

Note: If an error occurs on the server, and the autoApplyChanges property has the default value of true, any deleted record is restored to JSDO memory on the client.

For more information on handling saveChanges( ) method results for individual record-change operations, see the description of the saveChanges( ) method on page 314.

Following is an example showing an OpenEdge ABL implementation of the Delete operation and its invocation on a JSDO.

Note: The **bold** code in the JavaScript examples primarily trace the path of referenced JSDO method and callback parameters, as well as key code elements in the example, including those elements that are directly or indirectly referenced in the example description and notes.

An OpenEdge ABL implementation for Delete

For an OpenEdge Data Object resource, the signature of the ABL routine that implements a Delete operation must have a single INPUT-OUTPUT parameter for a DATASET, DATASET-HANDLE, TABLE or TABLE-HANDLE. If the resource supports before-image data, the parameter can only be for a DATASET or DATASET-HANDLE.

The following example shows a DeletedsCustomer( ) method that might implement the Data Object Delete operation for the ProDataSet resource, dsCustomer:

```
METHOD PUBLIC VOID DeletedsCustomer(INPUT-OUTPUT DATASET dsCustomer):

  DEFINE VAR hDataSet AS HANDLE NO-UNDO.
  hDataSet = DATASET dsCustomer:HANDLE.

  SUPER:DeleteData(DATASET-HANDLE hDataSet BY-REFERENCE).

END METHOD.
```

This implementation simply calls an overload of the DeleteData( ) method defined in the OpenEdge.BusinessLogic.BusinessEntity super class, which OpenEdge provides as an aid for implementing Data Object resources that support before-imaging. The invoked DeleteData( ) method provides a default implementation to delete an existing record in the server database identified from the single temp-table record deleted in the ProDataSet input to DeletedsCustomer( ).

Client JavaScript code: Delete

The following examples illustrate calling `remove( )` on a table reference in the JSDO to delete a record in JSDO memory, then calling `saveChanges( )` on the `dsCustomer` JSDO without Submit to delete the corresponding record in the server database (along with any other pending record-change operations), and using both an `afterDelete` event handler and a returned Promise object to handle the results.

This example uses simple error handling to log the contents of each error object returned by the JSDO `getErrors( )` method:

```
/* subscribe to event */
dsCustomer.ttCustomer.subscribe('afterDelete', onAfterDelete);

/* some code that deletes a record and sends it to the server */
var jsrecord = dsCustomer.ttCustomer.find(function(jsrecord) {
  return (jsrecord.data.Name === 'Lift Tours');
});
if (jsrecord) {jsrecord.remove();}

/* some other JSDO memory record changes . . . */

dsCustomer.autoApplyChanges = false;
dsCustomer.saveChanges(false).done( /* Successful method execution */
    function( jsdo, success, request ) {
        /* all resource operations invoked by saveChanges() succeeded */
        /* for example, track the average customer balance in certain states */
        var avgBalance = { MA : 0.0, maCount : 0,
                           VT : 0.0, vtCount : 0 };
        jsdo.ttCustomer.foreach( function(jsrecord) {
            if (jsrecord.data.State === 'MA') {
                /* sum balances for customers in MA . . . */
                avgBalance.MA += jsrecord.data.Balance;
                avgBalance.maCount += 1;
            }
            if (jsrecord.data.State === 'VT') {
                /* sum balances for customers in VT . . . */
                avgBalance.VT += jsrecord.data.Balance;
                avgBalance.vtCount += 1;
            }
        });
        /* compute averages and process further . . . */
        avgBalances.MA = avgBalances.MA / avgBalances.maCount;
        avgBalances.VT = avgBalances.VT / avgBalances.vtCount;

    }).fail( /* Unsuccessful method execution */
    function( jsdo, success, request ) {
        /* one or more resource operations invoked by saveChanges() failed */
        var lenErrors,
            errors,
            errorType;

        /* handle all operation errors */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) { /* Each error */
            console.log(JSON.stringify(errors[idxError]));
        }
    });

function onAfterDelete (jsdo , record , success , request ) {
    /* check for errors on any Delete operation */
    if (success) {
        /* do the normal thing for a successful Delete operation.
           for example, log the deleted record to the console  */
           console.log("Deleted Customer record for: " + record.data.Name);
```

```
        }
        record.acceptRowChanges();
    }
    else {
        /* all error messages handled by the Promise.fail() callback */
        record.rejectRowChanges();
    }
};
```

This example is identical to the previous one but with more complex error handling to log the contents of each error object using more readable output:

```
/* subscribe to event */
dsCustomer.ttCustomer.subscribe('afterDelete', onAfterDelete);

/* some code that deletes a record and sends it to the server */
var jsrecord = dsCustomer.ttCustomer.find(function(jsrecord) {
  return (jsrecord.data.Name === 'Lift Tours');
});
if (jsrecord) {jsrecord.remove();};

/* some other JSDO memory record changes . . . */

dsCustomer.autoApplyChanges = false;
dsCustomer.saveChanges(false).done( /* Successful method execution */
    function( jsdo, success, request ) {
        /* all resource operations invoked by saveChanges() succeeded */
        /* for example, track the average customer balance in certain states */
        var avgBalance = { MA : 0.0, maCount : 0,
                           VT : 0.0, vtCount : 0 };
        jsdo.ttCustomer.foreach( function(jsrecord) {
            if (jsrecord.data.State === 'MA') {
                /* sum balances for customers in MA . . . */
                avgBalance.MA += jsrecord.data.Balance;
                avgBalance.maCount += 1;
            }
            if (jsrecord.data.State === 'VT') {
                /* sum balances for customers in VT . . . */
                avgBalance.VT += jsrecord.data.Balance;
                avgBalance.vtCount += 1;
            }
        });
        /* compute averages and process further . . . */
        avgBalances.MA = avgBalances.MA / avgBalances.maCount;
        avgBalances.VT = avgBalances.VT / avgBalances.vtCount;

    }).fail( /* Unsuccessful method execution */
    function( jsdo, success, request ) {
        /* one or more resource operations invoked by saveChanges() failed */
        var lenErrors,
            errors,
            errorType;

        /* handle all operation errors */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) { /* Each error */
            switch(errors[idxError].type) {
                case progress.data.JSDO.DATA_ERROR:
                    errorType = "Server Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
```

```
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
                case default:
                    errorType = null; // Unexpected errorType value
                    break;
            }
            if (errorType) { /* log all error text
                console.log("ERROR: " + errorType + errors[idxError].error);
                if (errors[idxError].id) { /* error with record object */
                    console.log("RECORD ID: " + errors[idxError].id);
                    /* possibly log the data values for record with this ID */
                }
                if (errors[idxError].responseText) {
                    console.log("HTTP FULL TEXT: "
                                + errors[idxError].responseText);
                }
            }
            else { /* unexpected errorType */
                console.log("UNEXPECTED ERROR TYPE: "
                            + errors[idxError].type);
            }
        }
    });

function onAfterDelete (jsdo , record , success , request ) {
    /* check for errors on any Delete operation */
    if (success) {
        /* do the normal thing for a successful Delete operation.
           for example, log the deleted record to the console  */
           console.log("Deleted Customer record for: " + record.data.Name);
        }
        record.acceptRowChanges();
    }
    else {
        /* all error messages handled by the Promise.fail() callback */
        record.rejectRowChanges();
    }
};
```

This sample code:

- Subscribes a callback function, `onAfterDelete`, to the `afterDelete` event to enable error-checking and manipulation of the record after the Delete operation completes on the server, which includes the client logging the deleted record. This callback accepts or rejects changes to each record based on the success of the corresponding operation. Otherwise, it ignores and defers all error message handling to the `fail()` callback registered on the Promise returned by `saveChanges( )`. (Typically, you code similar `afterCreate` and `afterUpdate` event callbacks for the app as required.)
- Finds a record in the `ttCustomer` table with the customer name, `'Lift Tours'` and deletes the record from JSDO memory.
- Sets `autoApplyChanges` on the `dsCustomer` JSDO to `false`. Setting `autoApplyChanges` to `false` allows each Delete or other record-change operation attempted by `saveChanges( )` to be identified and handled, either after each operation returns its response to the client using an event callback (as in the example) or after all operations have returned their responses using a Promise callback (also as in the example). However, you must manually invoke the appropriate JSDO method to accept or reject the changes in JSDO memory, depending on the operation success and its requirements. In this example, the operation event callback accepts or rejects the changes for each record, based on the operation success.

- Calls `saveChanges(false)` (without Submit) to invoke all pending Data Object operations one record at a time on the server, including removal of the existing record with the customer name, `'Lift Tours'`. It thereby synchronizes the content of JSDO memory with the server database for one record-change operation at a time across the network. A returned Promise object handles the overall success or failure of `saveChanges( )` after all pending Create, Update, and Delete (CUD) operations complete.

  For a successful `saveChanges( )` invocation without Submit, the `done()` callback averages the balances of all client records for selected `State` field values. For an unsuccessful `saveChanges( )` invocation, the `fail()` callback handles all error messages returned in response to invoking each CUD operation across the network.

  Note: For unsuccessful operation results in this example, record changes are rejected before any associated error information is processed. However, the error messages for all operations remain available to the `getErrors( )` method until the next call to `fill( )` or `saveChanges( )`.

Note: Any event handler callback functions (such as `onAfterDelete`) always execute **before** any registered Promise callback methods (such as `done()`). For an invocation of `saveChanges( )` without Submit, the event callbacks execute after each CUD operation completes and returns its results from the server, and the appropriate registered Promise callbacks execute only after **all** results from these CUD operations have been returned from the server. This allows the results of all CUD operations to be processed together on the client. Although this example uses Promise callbacks in addition to operation event callbacks, all accepting and rejecting of JSDO memory changes is done in the operation event callbacks.

Submit operation example

To send multiple record changes over the network in a single request, you:

1. Use the same mechanisms to create, update, and delete records in JSDO memory as is described for each respective Create operation example on page 68, Update operation example on page 72, and Delete operation example on page 78.
2. Call the `saveChanges( )` method using the Submit operation by passing it a parameter value of `true`.

When you call `saveChanges(true)` to synchronize pending changes with the server database, this executes the OpenEdge ABL server routine that implements the Data Object Submit operation. This operation processes all pending record changes sent from the client in a single network request and returns the results of all these changes from the server in a single network response.

Note that this Submit operation is supported **only** for OpenEdge Data Object Services, where the Data Object is implemented for a ProDataSet resource with one or more temp-tables that supports before-imaging. This method call relies on client before-image data to identify all pending record changes for the network request since the last invocation of the `fill( )` or `saveChanges( )` method on the JSDO. The server then relies on the before-image data to identify and process each record change according to its type. If a ProDataSet resource and the JSDO that accesses it are not defined to support before-image data, making this call to `saveChanges( )` raises an exception.

When you call the `saveChanges( )` method to invoke a Submit operation on a supported resource, if jQuery Promises are supported in your environment, it returns a Promise object on which you can register callbacks to handle the results for all record changes in the request. Otherwise, you can subscribe event callbacks to handle the results of the request.

Operation results are thus returned as follows:

1. For each record change that the Submit operation completed, the JSDO fires an `afterDelete`, `afterCreate`, and `afterUpdate` event for each record change of the corresponding type in order to execute any callbacks you have subscribed to handle these events.
2. The JSDO fires an `afterSaveChanges` event to execute any callbacks you have subscribed to handle that event. In addition, any returned Promise object executes the Promise callbacks that you have registered, depending on the overall Submit operation results. (Note that the signatures of all Promise callbacks are identical to the signature of the `afterSaveChanges` event callback.)

After the `saveChanges( )` method completes a Submit operation successfully, no working records are set for the tables in the JSDO.

Note: If a Submit operation results in successful record create or update that includes changes to the record on the server that was sent from the client (for example, an update to a sequence value), JSDO memory is automatically synchronized with these server changes when the request object with the operation results is returned to the client.

Note: If an error occurs on the server for a Submit operation, and the `autoApplyChanges` property has the default value of `true`, each record change is automatically accepted or rejected in JSDO memory based on the presence of an error string that can be returned by calling the JSDO `getErrorString( )` method on the record. If you want to accept and reject record changes to synchronize with any server transaction that handles for the Submit operation, you must set `autoApplyChanges` to `false` and call the appropriate `accept*Changes( )` or `reject*Changes( )` method to handle the results.

For more information on handling `saveChanges( )` method results for a Submit operation, see the description of the saveChanges( ) method on page 314.

Following is an example showing an OpenEdge ABL implementation of the Submit operation and its invocation on a JSDO.

Note: The **`bold`** code in the JavaScript examples primarily trace the path of referenced JSDO method and callback parameters, as well as key code elements in the example, including those elements that are directly or indirectly referenced in the example description and notes.

An OpenEdge ABL implementation for Submit

The signature of the ABL routine associated with a submit operation must have a single `INPUT-OUTPUT` parameter for a `DATASET` or `DATASET-HANDLE`.

For an OpenEdge Data Object resource, the signature of the ABL routine that implements a Submit operation must have a single `INPUT-OUTPUT` parameter for a `DATASET` or `DATASET-HANDLE`.

The following example shows a `SubmitOrderEntry( )` method that might be associated with Data Object Submit operations:

The following example shows a `SubmitdsCustomer( )` method that might implement the Data Object Submit operation for the ProDataSet resource, `dsCustomer`:

```
METHOD PUBLIC VOID SubmitdsCustomer(INPUT-OUTPUT DATASET dsCustomer):

  DEFINE VAR hDataSet AS HANDLE NO-UNDO.
  hDataSet = DATASET dsCustomer:HANDLE.

  SUPER:SubmitData(DATASET-HANDLE hDataSet BY-REFERENCE).

END METHOD.
```

This implementation simply calls an overload of the `SubmitData( )` method defined in the `OpenEdge.BusinessLogic.BusinessEntity` super class, which OpenEdge provides as an aid for implementing Data Object resources that support before-imaging. The invoked `SubmitData( )` method provides a default implementation to create, update, or delete one or more records in the server database from the changed temp-table records in the ProDataSet input to `SubmitdsCustomer( )`.

Client JavaScript code: Submit

The following examples illustrate calling `add( )`, `assign( )`, and `remove( )` on a table reference in the JSDO to create, update, and delete records in JSDO memory, then calling `saveChanges(true)` (**with** Submit) on a `dsCustomer` JSDO to apply all the corresponding record changes (along with any other pending record changes) to the server database in a single request, and using both an `afterUpdate` event handler and a returned Promise object to handle the results.

This example uses simple error handling to log the contents of each error object returned by the JSDO `getErrors( )` method:

```
/* subscribe to event */
dsCustomer.ttCustomer.subscribe('afterUpdate', onAfterUpdate);

/* some code that adds a record to JSDO memory */
var jsrecord = dsCustomer.ttCustomer.add( {State : 'MA'} );

/* some code that updates a record in JSDO memory */
jsrecord = dsCustomer.ttCustomer.find(function(jsrecord) {
  return (jsrecord.data.Name === 'Lift Tours');
});
if (jsrecord) {jsrecord.assign( {State: 'VT'} );}

/* some code that deletes a record from JSDO memory */
jsrecord = dsCustomer.ttCustomer.find(function(jsrecord) {
  return (jsrecord.data.Name === 'Burrows Sport Shop');
});
if (jsrecord) {jsrecord.remove();}

/* some other JSDO memory record changes . . . */

dsCustomer.autoApplyChanges = false;
dsCustomer.saveChanges(true).done( /* Successful Submit operation */
    function( jsdo, success, request ) {
        /* all record changes processed by the Submit succeeded */
        /* for example, track the average customer balance in certain states */
        var avgBalance = { MA : 0.0, maCount : 0,
                           VT : 0.0, vtCount : 0 };
        jsdo.ttCustomer.foreach( function(jsrecord) {
            if (jsrecord.data.State === 'MA') {
                /* sum balances for customers in MA . . . */
                avgBalance.MA += jsrecord.data.Balance;
```

```
                avgBalance.maCount += 1;
            }
            if (jsrecord.data.State === 'VT') {
                /* sum balances for customers in VT . . . */
                avgBalance.VT += jsrecord.data.Balance;
                avgBalance.vtCount += 1;
            }
        });
        /* compute averages and process further . . . */
        avgBalances.MA = avgBalances.MA / avgBalances.maCount;
        avgBalances.VT = avgBalances.VT / avgBalances.vtCount;
        jsdo.acceptChanges(); /* Accept all record changes */
    }).fail( /* Unsuccessful Submit operation */
    function( jsdo, success, request ) {
        /* one or more record changes processed by the Submit failed */
        var lenErrors,
            errors,
            errorType;

        /* handle Submit operation errors */
        console.log("Operation: Submit");
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) { /* Each error */
            console.log(JSON.stringify(errors[idxError]));
        }
        jsdo.rejectChanges(); /* Reject all record changes */
        /* NOTE: Where an error occurred on the Submit invocation, thus
           preventing the back end from processing the changes, you might
           want to retry the Submit after the error conditions have been
           cleared.
    });

function onAfterUpdate (jsdo , record , success , request ) {
    /* check for errors on any record update in the Submit operation */
    if (success) {
        /* do the normal thing for a successful record update.
           for example, process updated record according to its Balance value  */
        if (record.data.Balance > $100000.00) {
            /* process a high balance condition . . . */
        }
    }
    else {
        /* all error messages handled by the Promise.fail() callback */
    }
};
```

This example is identical to the previous one but with more complex error handling to log the contents of each error object using more readable output:

```
/* subscribe to event */
dsCustomer.ttCustomer.subscribe('afterUpdate', onAfterUpdate);

/* some code that adds a record to JSDO memory */
var jsrecord = dsCustomer.ttCustomer.add( {State : 'MA'} );

/* some code that updates a record in JSDO memory */
jsrecord = dsCustomer.ttCustomer.find(function(jsrecord) {
  return (jsrecord.data.Name === 'Lift Tours');
});
if (jsrecord) {jsrecord.assign( {State: 'VT'} );};

/* some code that deletes a record from JSDO memory */
jsrecord = dsCustomer.ttCustomer.find(function(jsrecord) {
  return (jsrecord.data.Name === 'Burrows Sport Shop');
```

```
});
if (jsrecord) {jsrecord.remove();};

/* some other JSDO memory record changes . . . */

dsCustomer.autoApplyChanges = false;
dsCustomer.saveChanges(true).done( /* Successful Submit operation */
    function( jsdo, success, request ) {
        /* all record changes processed by the Submit succeeded */
        /* for example, track the average customer balance in certain states */
        var avgBalance = { MA : 0.0, maCount : 0,
                           VT : 0.0, vtCount : 0 };
        jsdo.ttCustomer.foreach( function(jsrecord) {
            if (jsrecord.data.State === 'MA') {
                /* sum balances for customers in MA . . . */
                avgBalance.MA += jsrecord.data.Balance;
                avgBalance.maCount += 1;
            }
            if (jsrecord.data.State === 'VT') {
                /* sum balances for customers in VT . . . */
                avgBalance.VT += jsrecord.data.Balance;
                avgBalance.vtCount += 1;
            }
        });
        /* compute averages and process further . . . */
        avgBalances.MA = avgBalances.MA / avgBalances.maCount;
        avgBalances.VT = avgBalances.VT / avgBalances.vtCount;
        jsdo.acceptChanges(); /* Accept all record changes */
    }).fail( /* Unsuccessful Submit operation */
    function( jsdo, success, request ) {
        /* one or more record changes processed by the Submit failed */
        var lenErrors,
            errors,
            errorType;

        /* handle Submit operation errors */
        console.log("Operation: Submit");
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) { /* Each error */
            switch(errors[idxError].type) {
                case progress.data.JSDO.DATA_ERROR:
                    errorType = "Server Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
                case default:
                    errorType = null; // Unexpected errorType value
                    break;
            }
            if (errorType) {
                console.log("ERROR: " + errorType + errors[idxError].error);
                if (errors[idxError].id) { /* error with record object */
                    console.log("RECORD ID: " + errors[idxError].id);
                    /* possibly log the data values for record with this ID */
                }
                if (errors[idxError].responseText) {
                    console.log("HTTP FULL TEXT: "
                                + errors[idxError].responseText);
```

```
                }
            }
            else { /* unexpected errorType */
                console.log("UNEXPECTED ERROR TYPE: "
                            + errors[idxError].type);
            }
        }
        jsdo.rejectChanges(); /* Reject all record changes */
        /* NOTE: Where an error occurred on the Submit invocation, thus
           preventing the back end from processing the changes, you might
           want to retry the Submit after the error conditions have been
           cleared.
    });

function onAfterUpdate (jsdo , record , success , request ) {
    /* check for errors on any record update in the Submit operation */
    if (success) {
        /* do the normal thing for a successful record update.
           for example, process updated record according to its Balance value  */
        if (record.data.Balance > $100000.00) {
            /* process a high balance condition . . . */
        }
    }
    else {
        /* all error messages handled by the Promise.fail() callback */
    }
};
```

This sample code:

- Subscribes a single callback function, `onAfterUpdate`, to the `afterUpdate` event to enable specific data manipulation for updates to an existing record that is returned from the server. The manipulation on a successful record update includes processing the record based on a high balance condition. This callback ignores and defers all error handling to the `fail()` callback registered on the Promise returned by `saveChanges( )`. (You might also code similar `afterCreate` and `afterDelete` event callbacks for the app as required.)
- Performs a variety of record changes in JSDO memory as part of the Submit operation (**Note:** The table reference can be omitted if `ttCustomer` is the only temp-table in the `dsCustomer` ProDataSet.)
- Sets `autoApplyChanges` on the `dsCustomer` JSDO to `false`. Setting `autoApplyChanges` to `false` allows each Update or other record-change operation attempted by `saveChanges( )` to be identified and handled, either after each operation returns its response to the client using an event callback (as in the example) or after all operations have returned their responses using a Promise callback (also, as in the example). However, you must manually invoke the appropriate JSDO method to accept or reject the changes in JSDO memory, depending on the operation success and its requirements. In this example, the Promise callbacks accept or reject the changes for all records, based on the success of the overall Submit operation.
- Calls `saveChanges(true)` (**with** Submit) to send all pending record changes to the server in a single request, and thereby synchronizes the content of JSDO memory with the server database in a single Data Object operation across the network. A returned Promise object handles the overall success or failure of the Submit operation after all pending record changes complete.

  For a successful Submit invocation, the `done()` callback averages the balances of all client records for selected `State` field values and accepts all record changes. For an unsuccessful Submit invocation, the `fail()` callback handles all types of errors returned for the Submit operation and rejects all record changes.

Note: Where a network or server error has prevented the Submit operation from executing on the server, you might want to retry the Submit after the associated network or server error has cleared.

Note: Any event handler callback functions (such as `onAfterUpdate`) always execute **before** any registered Promise callback methods (such as `done()`), and for a Submit operation, all such callbacks execute only after **all** results for the Submit have been returned from the server. This allows the results of all record changes to be processed together on the client. Although this example uses Promise callbacks in addition to operation event callbacks, all accepting and rejecting of JSDO memory changes is done in the Promise callbacks.

Accessing custom Invoke operations

In addition to the standard CRUD methods, you can call custom *invocation methods* on a JSDO that supports them. Each invocation method corresponds to a single routine on the server that is mapped depending on the type of Progress Data Object Service.

For an OpenEdge Data Object Service, each invocation method maps to a specific ABL routine that is implemented and defined in the Data Object resource to be called using an Invoke operation. For a Rollbase Data Object Service, each invocation method maps to a server-generated method that returns records from a Rollbase object that is related to the Data Object resource.

The Data Service Catalog identifies the available Invoke operations, with their corresponding custom JSDO method and server routine mappings. Calling an invocation method on the JSDO thereby causes the corresponding routine to execute on the server.

The signature of an OpenEdge ABL routine defined to implement an Invoke operation is largely unrestricted. All input and output parameters can use any ABL data types supported by OpenEdge Data Object Services. For more information, see OpenEdge ABL to JavaScript data type mappings on page 363.

The signatures of Rollbase server methods that are defined to implement Invoke operations conform to a common pattern used to return data with supported Rollbase types from a related object. For more information on the supported types, see Rollbase object to JavaScript data type mappings on page 365.

For an ABL implementation, the invocation method name can be the same as that of the ABL routine, or it can be an alias, as defined by the resource. The invocation method passes any ABL input parameters as properties of a single object parameter. The method returns results from the ABL routine, including any return value and output parameters, in the `response` property of a request object.

This `response` property references an object containing properties whose names match the names of output parameters defined in the ABL routine. Since JavaScript is case-sensitive, code that accesses the value of an output parameter must exactly match the name defined in the ABL routine. For user-defined functions and non-void ABL methods, the return value is available in the `_retVal` property of the `response` property object. This `_retVal` property also contains any error information returned by the server If the request fails.

For a Rollbase implementation, the Rollbase server determines the required mappings between invocation methods and their corresponding server implementaitons, and generates a Data Service Catalog accordingly. In this case, the `_retVal` property of the `response` property object always contains the related data that the method returns.

Note: For an OpenEdge implementation, JSDO memory is not automatically updated with the results of an Invoke operation. To add records returned by the Invoke operation to JSDO memory, call `addRecords( )` on the appropriate table. For a Rollbase implementation, JSDO memory is only updated automatically with the related data returned by an Invoke operation when that Invoke operation functions as the Read operation for a given resource; othewise, you must use `addRecords( )` on the JSDO created for the related resource to merge the returned data into its JSDO memory.

Asynchronous vs. synchronous method execution

You can execute invocation methods either asynchronously (the default behavior) or synchronously, depending on how you call them. A JSDO provides two basic mechanisms for calling an invocation method:

- **Calling the method by name** — Invoking the method on the JSDO instance like any other JSDO method. This allows you to call the method asynchronously or synchronously according to a parameter that you pass, with all asynchronous results handled by event callbacks.
- **Passing the method as a parameter to the JSDO `invoke( )` method** — Invoking the `invoke( )` method on the JSDO instance, passing as parameters the name of the method and an object containing properties whose names match the names of the input parameters defined for the server method that implements the Invoke operation. This allows you to call the invocation method asynchronously only, with results handled either (or both) using event callbacks or callbacks that you register using a Promise object returned as the value of the `invoke( )` method.

If you call the invocation method by name, along with an optional object parameter containing properties that match any input parameters defined for the implementing server method, you can pass a second optional `boolean` parameter that specifies the execution mode. A value of `true` (the default) for this second parameter specifies asynchronous execution; for synchronous execution, set the parameter to `false`.

To process the results of an asynchronous method call, you can subscribe an event-handler callback function to the `afterInvoke` event, or if you use the `invoke( )` method, register callbacks on the returned Promise object by calling any of the available Promise methods (see Asynchronous execution using Promises on page 52). All Invoke operation callbacks have the same signature, which includes a returned request object from which you can return results by accessing its `response` property.

For a synchronous call, you do not use callbacks, but access results through the `response` property of the request object that is returned as the value of the invocation method.

Note: The recommended execution mode is asynchronous, because every invocation method call sends a request across the network. There are some cases where you might want to execute the method synchronously, for example, if you want to use the return value in an expression and the request response time is likely to be relatively short. However, please note that some web browsers are deprecating synchronous network requests in some or all situations.

Invoke operation example

To illustrate different ways that an Invoke operation can return results, the following examples show different ways of calling an Invoke operation on the client that is implemented by an OpenEdge ABL class method that returns both a `DECIMAL` value and an output parameter value of type `TABLE`.

Note: The **`bold`** code in the JavaScript examples primarily trace the path of referenced JSDO method and callback parameters, as well as key code elements in the example, including those elements that are directly or indirectly referenced in the example description and notes.

An OpenEdge ABL implementation for an Invoke operation

The following example shows an ABL method that might implement an Invoke operation in any Business Entity that accesses the `sports2000` database, such as `Customer` (see Accessing standard CRUD and Submit operations on page 61):

```
DEFINE PRIVATE TEMP-TABLE poCustomer
  FIELD CustNum LIKE Customer.CustNum
  FIELD Name LIKE Customer.Name
  FIELD CreditLimit LIKE Customer.CreditLimit
  INDEX idxCust UNIQUE PRIMARY poCustNum ASCENDING
  .

METHOD PUBLIC DECIMAL GetCreditInfo ( INPUT piCustNum AS INT, OUTPUT
    TABLE poCustomer):

  EMPTY TEMP-TABLE poCustomer NO-ERROR.
  FOR EACH Customer WHERE CustNum = piCustNum:
    CREATE poCustomer.
    ASSIGN
      poCustomer.CustNum = CustNum
      poCustomer.Name = Name
      poCustomer.CreditLimit = CreditLimit
      .
  END.

  RETURN poCustomer.CreditLimit.

END METHOD.
```

This method loops through `Customer` records, and returns, as an output parameter, a temp-table with a single record containing three fields (`CustNum`, `Name`, and `CreditLimit`) that are set from three corresponding fields of the unique `Customer` record with its `CustNum` field matching the value of the input parameter, `piCustNum`. The method also returns, as its value, the `CreditLimit` field value from the same `Customer` record. This allows `CreditLimit` to be more easily referenced on the client as the method return value in an expression if desired.

Client JavaScript code: An Invoke

The following JavaScript illustrates an asynchronous call to the preceding method using an `afterInvoke` event callback (`onAfterInvokeGetCreditInfo`) to handle the results:

```
dsCustomer.subscribe('afterInvoke', 'GetCreditInfo',
                     onAfterInvokeGetCreditInfo);
var currentCust = { piCustNum : 10 };

dsCustomer.GetCreditInfo ( currentCust );

function onAfterInvokeGetCreditInfo (jsdo , success , request ) {
    /* check for errors on this Invoke operation */
    if (success) {
        /* do the normal thing for a successful Invoke operation. */
        /* for example, evaluate the CreditLimit for Customer currentCust
           and if their Balance is greater, display a message . . . */
        if (request.response._retVal && request.response._retVal < 1000.00) {
            var poCustomer = request.response.poCustomer.poCustomer;
            if (dsCustomer.find(function(jsrecord) {
                    return (jsrecord.data.CustNum == poCustomer[0].data.CustNum);
                }) {
                if (dsCustomer.Balance > request.response._retVal) {
                    console.log("Customer "
                        + dsCustomer.CustNum + " " + dsCustomer.Name
                        + " has a Balance higher than their CreditLimit"
                    );
                }
            }
        }
    }
    else {
        /* return errors from this Invoke operation */
        if (request.response && request.response._errors &&
            request.response._errors.length > 0){

            var lenErrors = request.response._errors.length;
            for (var idxError=0; idxError < lenErrors; idxError++) {
                var errorEntry = request.response._errors[idxError];
                var errorMsg = errorEntry._errorMsg;
                var errorNum = errorEntry._errorNum;
                /* handle Invoke operation error . . . */

            }
        }
    }
};
```

This sample code:

- Subscribes an event-handler callback, `onAfterInvokeGetCreditInfo`, to the `afterInvoke` event to enable error-checking and processing of the results.
- Defines the object variable, `currentCust`, to specify the value of the `piCustNum` input parameter passed to `GetCreditInfo( )`.
- Calls `GetCreditInfo( )` on the `dsCustomer` JSDO asynchronously.
- The callback function tests for the success of the Invoke operation and accesses both the operation return value (`_retVal`) and its one output parameter (`poCustomer`) to do the normal thing for a successful execution, and accesses any returned errors for an unsuccessful execution.

- In the assignment of `request.poCustomer.poCustomer` to the `poCustomer` variable, the parameter name (`poCustomer`) must be specified twice. The first instance refers to the parameter name as defined in the ABL routine; the second instance is the name of the JavaScript object containing the table data that the server serializes as JSON to send to the client.

The following JavaScript illustrates an asynchronous call to the same method using Promise callbacks to handle the results:

```
var currentCust = { piCustNum : 10 };

dsCustomer.invoke ( "GetCreditInfo", currentCust ).done(
    function( jsdo, success, request ) {
        /* do the normal thing for a successful Invoke operation. */
        /* for example, evaluate the CreditLimit for Customer currentCust
           and if their Balance is greater, display a message . . . */
        if (request.response._retVal && request.response._retVal < 1000.00) {
            var poCustomer = request.response.poCustomer.poCustomer;
            if (dsCustomer.find(function(jsrecord) {
                    return (jsrecord.data.CustNum == poCustomer[0].data.CustNum);
                }) {
                if (dsCustomer.Balance > request.response._retVal) {
                    console.log("Customer "
                        + dsCustomer.CustNum + " " + dsCustomer.Name
                        + " has a Balance higher than their CreditLimit"
                    );
                }
            }
        }
    }).fail(
    function( jsdo, success, request ) {
        /* return errors from this Invoke operation */
        if (request.response && request.response._errors &&
            request.response._errors.length > 0) {

            var lenErrors = request.response._errors.length;
            for (var idxError=0; idxError < lenErrors; idxError++) {

                var errorEntry = request.response._errors[idxError];
                var errorMsg = errorEntry._errorMsg;
                var errorNum = errorEntry._errorNum;
                /* handle Invoke operation error . . . */

            }
        }
    });
```

This sample code:

- Defines the object variable, `currentCust`, to specify the value of the `piCustNum` input parameter passed to `GetCreditInfo( )`.
- Calls `GetCreditInfo( )` on the `dsCustomer` JSDO asynchronously using the JSDO `invoke( )` API to return a Promise object.
- Chains access to the `done( )` method on the returned Promise object to register a callback function that handles successful operation execution. This callback accesses both the Invoke operation return value (`_retVal`) and its one output parameter (`poCustomer`) to do the normal thing for a successful execution.
- Chains access to the `fail( )` method on the returned Promise object to register a callback function that handles unsuccessful operation execution. This callback accesses and handles any returned errors.
- In the assignment of `request.poCustomer.poCustomer` to the `poCustomer` variable, the parameter name (`poCustomer`) must be specified twice. The first instance refers to the parameter name as defined

in the ABL routine; the second instance is the name of the JavaScript object containing the table data that the server serializes as JSON to send to the client.

The following JavaScript illustrates a synchronous call to the same method:

```
var currentCust = { piCustNum : 10 };
var request = null;

try {
    request = dsCustomer.GetCreditInfo ( currentCust, false ));
    if (!request.success) {
        throw "Invoke request to GetCreditInfo() not successful for CustNum == "
            + "for CustNum == " + currentCust.piCustNum.toString();
    }
    /* do the normal thing for a successful Invoke operation. */
    /* for example, evaluate the CreditLimit for Customer currentCust
       and if their Balance is greater, display a message . . . */
    else if (request.response._retVal && request.response._retVal < 1000.00) {
        var poCustomer = request.response.poCustomer.poCustomer;
        if (dsCustomer.find(function(jsrecord) {
                return (jsrecord.data.CustNum == poCustomer[0].data.CustNum);
            }) {
            if (dsCustomer.Balance > request.response._retVal) {
                console.log("Customer "
                    + dsCustomer.CustNum + " " + dsCustomer.Name
                    + " has a Balance higher than their CreditLimit"
                );
            }
        }
        else {
            /* No such record in dsCustomer */
            throw "No dsCustomer record with CustNum == "
                + currentCust.piCustNum.toString();
        }
    }
    else {
        throw "dsCustomer.GetCreditInfo() returns "
            + "out-of-range CreditLimit for CustNum == "
            + currentCust.piCustNum.toString();
    }
}
catch(err) {
    /* handle error with access to request . . . */
};
```

This sample code:

- Defines two object variables: `currentCust` to specify the value of the `piCustNum` input parameter passed to `GetCreditInfo( )`, and `request` to hold the request object reference returned as the value of `GetCreditInfo( )`.
- Uses `try`...`catch` blocks to handle certain errors.
- Calls `GetCreditInfo( )` on the `dsCustomer` JSDO to test the contents of its returned request object, and either do the normal thing for a successful execution or throw various error strings for an unsuccessful Invoke operation or other errors encountered.
- Might not be a good example of an Invoke operation for synchronous execution, because it is not executed to return a value within an expression, and in addition to a return value it also returns an output parameter (`poCustomer`) containing the corresponding Customer record with a small subset of fields. Searching a large result set for the specified record to return in this output parameter can block for an extended period of time.

Managing JSDO login sessions

The first task in accessing a Data Object resource with a JSDO is to create a JSDO login session for the web application that provides the Data Object Service and resource you need. You can create a JSDO login session by calling the `login( )` method on an instance of one of the following OpenEdge JavaScript classes:

- **progress.data.JSDOSession class on page 124** — (Recommended) Supports asynchronous access only to a web server using jQuery Promises that are returned by `JSDOSession` methods (such as `login( )`) for registering callbacks to handle the results. Each instance of this class enforces access to a single web application that supports Data Object Services using the application's configured web server authentication model, as specified by options passed to the class constructor. This means that once created, a single `JSDOSession` instance can be used to access only the web application for which it is created. Progress recommends that you use this class because it enforces the most typical web authentication pattern for a client app that accesses a web application. For client web apps, this includes the option to maintain an existing JSDO login session when the user initiates a page refresh on the browser page that is running the app (available in Progress Data Objects Version 4.3 or later).
- **progress.data.Session class on page 136** — Supports either synchronous or asynchronous access to a web server, with `Session` methods either returning synchronous results (the default), often including a method value, or optionally returning asynchronous results using callbacks that you subscribe as event handlers prior to calling the method. Each instance of this class can be used to access any web application that supports Data Object Services, and can be re-used to access any other such web application, using that application's configured authentication model, which you specify as part of calling the instance's `login( )` method. This means that once created, a single `Session` instance can be used to access a different web application once any current web application has been logged out using the `logout( )` method. Progress deprecates this class in favor of `progress.data.JSDOSession`, though you do need to use a `Session` instance if your client environment does not support jQuery Promises (or the exact equivalent). **Note:** A client environment based on Kendo UI supports jQuery Promises.

Using an instance of either class, the `login( )` method takes any required user credentials as input and once authenticated on the specified web application, establishes a JSDO login session for it.

The successful result of calling `login( )` is typically user authorization to access the resources of Data Object Services hosted by the web application. Once user authorization is obtained, you then call the `addCatalog( )` method on the session instance to load one or more Data Service Catalogs for the Data Object Service or Services you want to access. You can then create a JSDO for a Data Object resource defined in any of the loaded Catalogs.

However, before invoking this login sequence, you need to gather some information, depending on the session class instance, to determine how best to configure and code it. You also might want to manage client app execution based on whether the login session and its connected Data Object Services are online and available over the network (*session online status*). You can do this for a JSDO login session with the help of a particular set of session events, methods, and properties.

The following sections describe how to prepare for and manage these tasks:

- Requirements for creating a JSDO login session on page 96
- Using default web pages to support client app login on page 99
- Handling changes in session online status on page 99
- Using protected web resources on page 102

Requirements for creating a JSDO login session

Identify any requirements from the following information to configure and code a JSDO login session:

- Choosing the web server authentication model on page 96
- Whether to enable cookies on page 97
- Choosing appropriate URIs for the app type and its deployment location on page 97
- Supporting page refresh for web apps on page 98

Choosing the web server authentication model

Web servers support a number of authentication models to manage client access to resources provided by a web application. The JSDO supports the following authentication models, which correspond to the ways that a web application can be protected:

- **Anonymous authentication** — No authentication is required to access any of its Data Object Services. This is the default authentication model.

  Although no authentication is required, the client app must still call the `login( )` method on the session instance without user credentials in order to establish a JSDO login session for the specified web application. If you do pass credentials for this authentication model, the method ignores them.

- **HTTP Basic authentication** — The web application requires a valid username and password (credentials) to access its Data Object Services.

  To have a `JSDOSession` or `Session` instance manage access to the web application's resources for you, you need to pass these credentials in a call to the instance's `login( )` method. Typically, you would require the user to enter their credentials into a login dialog provided by your client app, either using a form of your own design or using a template provided by Progress Software (see Using default web pages to support client app login on page 99). Once authenticated, all further access to web application resources is provided according to the user's authorization settings, and the client app is ready to load a Data Service Catalog.

  Note: The option to manage page refresh that is provided for client web apps using the `JSDOSession` class is not supported for HTTP Basic. For more information, see Supporting page refresh for web apps on page 98 (available in Progress Data Objects Version 4.3 or later).

  Note: It is possible to configure a client app so that, prior to logging in, it opens a protected web page provided by the web application. However, this is an atypical client app configuration. For more information, see Using protected web resources on page 102.

- **HTTP Form-based authentication** — The web application requires a valid username and password (credentials) to access its Data Object Services.

  Like HTTP Basic, Form-based authentication requires user credentials for access to protected resources; the difference is that the web application itself sends a form to the client to get the credentials. However, when you have a `JSDOSession` or `Session` instance manage access to the web application's resources, you handle Form-based authentication in the same way that you handle Basic—get the user's credentials yourself and pass them to the `login( )` method. Internally, the JSDO session intercepts the form sent to the client by the web application and handles the authentication without the form being displayed. Once authenticated, all further access to web application resources is provided according to the user's authorization settings, and the client app is ready to load a Data Service Catalog.

Note: Similar to HTTP Basic, using HTTP Forms, it is possible to configure a client app so that, prior to logging in, it opens a protected web page provided by the web application. However, this is an atypical client app configuration. For more information, see Using protected web resources on page 102.

Caution: Progress recommends that you always use SSL (HTTPS) when logging into a web application, but especially for a web application configured for HTTP Basic authentication. If you do not, the `login( )` method sends its user credentials to the web server as clear text.

You must know the web server authentication model, the client app type and platform, and how web application resources are protected on the web server. You can then set the `authenticationModel` property in a `JSDOSession` object constructor or on an instantiated `Session` object accordingly. For more information on security considerations for Data Object Services, see the sections on REST application security in *OpenEdge Development: Web Services* and in the administration documentation for your particular OpenEdge application server.

As noted later in this section, OpenEdge, provides default web resources with every deployed web application that you can use to help implement a login sequence, or you can define similar web resources of your own. For more information, see Using default web pages to support client app login on page 99.

Whether to enable cookies

You need to decide whether the web browsers or mobile devices where the client app runs will have cookies enabled. If the web application you access uses HTTP Form-based authentication, the mobile devices and web browsers that access the web application **must** have cookies enabled. Otherwise, the client app cannot login and access Data Object Services. If there is any question about the availability of cookies on client platforms, you might consider configuring HTTP Basic authentication for the web application instead, and set the session instances `authenticationModel` property accordingly.

If the web application uses HTTP Basic authentication and the mobile devices and web browsers will not have cookies enabled, you **must** set a property in the single sign-on (SSO) configuration of the web application to allow session logins to work from the client app. For more information, see the sections on enabling SSO for a web application in the administration documentation for your OpenEdge application server.

Choosing appropriate URIs for the app type and its deployment location

For Data Object Services, in order to log into a web application and load any of its Data Service Catalogs, you need to provide appropriate URIs (relative or absolute) for both, depending on the client app type and where it is deployed.

If you are writing a web app, and it will be deployed to the same Apache Tomcat Web server as the web application it is accessing, all of these URIs can be relative to the web server root (domain or host and port). The web browser automatically appends these relative URIs to the web server root from which the web app is loaded. However, if a web app is deployed to a different web server from the one where the web application is deployed, all of these URIs must be absolute and include the web server root for the web application.

If you are writing a hybrid app, all of the URIs for the web application and its Data Service Catalogs must always be provided as absolute URIs, because a hybrid app is loaded and executed from the JSDO memory of the mobile device, which knows nothing about the web application its web app is going to access.

Note: In general, if a client app requires absolute URIs, you need to maintain separate JavaScript sources or configuration files for versions of the client app that you deploy for different web application environments, such as one for testing and another for production.

Supporting page refresh for web apps

Note: Applies to Progress Data Objects Version 4.3 or later.

When a user runs a web app in a browser, it is common, especially when the app seems unresponsive, for the user to click a browser control that causes the page to be reloaded (refreshed) and redisplayed in its document context, hopefully in a state that the user expects. In the process of reloading the page, the refresh clears memory so that all of the JavaScript objects and variables are created again as if for the first time.

If the app is accessing a JSDO when the user initiates this page refresh, by default, all information that a `JSDOSession` or `Session` object uses to manages its JSDO login session is also lost. As such, the refresh resets the state of the login session as if it was never logged in. This causes the app to prompt the user for their credentials, if required, and invoke the session object's `login( )` method in order to re-establish the JSDO login session's connection to its web application. However, it should not be necessary for the session to log in again, because the state of the web application on the server is not affected by the browser page refresh.

To reduce the need for an additional login when a page refresh occurs, both the `progress.data.JSDOSession` class and the `progress.data.getSession( )` stand-alone function provide the option for an instance to store some state from its login session once the session is first successfully established. The instance then attempts to re-establish the same login session state without requiring an additional session login when the web app page is refreshed.

To support page refresh in a web app, the `JSDOSession` class provides a constructor option (`name` property) to enable the page refresh feature itself and a method (`isAuthorized( )`) to test whether its `JSDOSession` instance already has an authorized login session established from a restored prior session state, or if a login session has yet to be established for a first-time execution. You enable page fresh support by passing the `name` property to the constructor with a value (an *operative* value) that the `JSDOSession` instance can use to store and restore its login session state. The `JSDOSession` class also provides a read-only `name` property that you can access to return the operative value, if any, that was set to enable its page refresh support.

Note: Page refresh supports **only** Anonymous and Form-based authentication. If you attempt to enable page refresh support using Basic authentication, any setting of the `name` property is ignored.

Use of this page refresh support typically follows this web app algorithm for as many `JSDOSession` instances as required:

1. At the start, create the `JSDOSession` object, enabling page refresh support by passing the `name` property with an operative value to the constructor.
2. Call `isAuthorized( )` on the `JSDOSession` to test if:
   - A login session for the `JSDOSession` instance already exists, having been restored from a page refresh, and is authorized to access its web application. App execution continues with the existing login session.
   - No authorized login session for the `JSDOSession` instance exists, either because it has yet to be established (first-time execution), it was previously logged out, it timed out, it otherwise lost its authorized connection to the web application. A new login session for the instance needs to be established before app execution continues.
   - Some other session failure, such as an offline status, has prevented the `JSDOSession` instance from connecting to its web application and must be handled by the app.

The `progress.data.getSession( )` stand-alone function, which creates and returns a `JSDOSession` instance already initialized with a valid JSDO login session and loaded with a Data Service Catalog, provides a similar option (`name` property) to enable the page refresh feature in the `JSDOSession` that it creates. However, with this stand-alone function, you have no need for the `isAuthorized( )` method, because the `getSession( )` function fully supports the page refresh feature itself without the need for any additional coding, such as invoking `isAuthorized( )` on an existing `JSDOSession`.

For more information on enabling page refresh support and its effects, see the description of the progress.data.JSDOSession class on page 124 and the getSession( ) stand-alone function on page 248. For more information on using the `isAuthorized( )` method, including example code, see the description of the isAuthorized( ) method on page 268.

Using default web pages to support client app login

When you deploy OpenEdge Data Object Services in a web application to a Tomcat web server, OpenEdge provides default web pages that you can use to configure web application authentication. The URIs for the following web pages (provided relative to the web application root) support the startup and user login of a client app in a way that authenticates the user prior to requesting access to protected Data Object resources, depending on the web server authentication model:

- **`/index.html`** — Default public welcome page for client app startup. This page is typically unprotected and provides, or redirects access to another unprotected page that provides, a login form for the user to enter login credentials.

  Caution: Protecting the welcome page, or any other page, that the client app accesses prior to user login is an atypical configuration that Progress does not recommend. For more information, see Using protected web resources on page 102.

- **`/static/home.html`** — Default protected login target page provided by the web application to support HTTP Basic and HTTP Form-based authentication. (For Anonymous authentication, this page is always unprotected.) Generally, this page is not designed to be displayed as part of client app UI, but to be used as a protected web resource against which the session instance's `login( )` method **must** authenticate prior to accessing protected Data Object resources. The client app typically provides a separate, unprotected login page for the user to enter their credentials, which are then passed to the `login( )` method. (For a `progress.data.Session` instance only, you can also pass the URI of a non-default protected web resource to `login( )`.)

  Note: OpenEdge protects this `/static/home.html` page in all web applications that it generates by configuring it for access only by users assigned the role, `ROLE_PSCUser`, which you can assign to users as part of configuring your web application security. For more information, see the sections on security in the administration documentation of your OpenEdge application server.

Thus, JSDO session management uses these default options to ensure that user authentication can occur prior to loading a Data Service Catalog (using `addCatalog( )`) and requesting access to a protected Data Object resource on behalf of a JSDO.

Handling changes in session online status

Both `progress.data.JSDOSession` and `progress.data.Session` instances provide events, methods, and properties that you can use to identify and respond to changes in a client app's session online status. This online status reflects two basic network conditions, where:

- The connection between the device (or JSDO login session) in which the client app is running and an available Internet access point (such as a Wi-Fi router) has either been broken or re-established, also referred to as the *device online status* (or the *JSDO login session online status*).
- The availability of a web application to which the client app was previously connected has either been interrupted or re-established on a device (or JSDO login session) that is otherwise connected to the Internet, also referred to as *web application online status* or *Data Object Service online status*.

The following table briefly describes session instance events, methods, and properties that you can use to track a client app's online status.

Table 9: Events, methods, and properties for tracking app online status

Member	Brief description (See also the reference entry)
connected property on page 216	Returns a `boolean` that indicates the most recent online status of the current `JSDOSession` or `Session` object when it last determined if the web application it manages was available.
offline event on page 292	Fires when the current `JSDOSession` or `Session` object detects that the device on which it is running has gone offline, or that the web application to which it has been connected is no longer available.
online event on page 294	Fires when the current `JSDOSession` or `Session` object detects that the device on which it is running has gone online after it was previously offline, or that the web application to which it is connected is now available after it was previously unavailable.
ping( ) method (JSDOSession class) on page 297	Determines the online state of the current `JSDOSession` object from its ability to access the web application that it manages, and for an OpenEdge web application, from detecting if its associated application server is running.
ping( ) method (Session class) on page 300	Determines the online state of the current `Session` object from its ability to access the web application that it manages, and for an OpenEdge web application, from detecting if its associated application server is running.
pingInterval property on page 303	A `number` that specifies the duration, in milliseconds, between one automatic execution of the current `JSDOSession` or `Session` object's `ping( )` method and the next.
subscribe( ) method (JSDOSession class) on page 348	Subscribes a given event callback function to an event of the current `JSDOSession` object.
subscribe( ) method (Session class) on page 349	Subscribes a given event callback function to an event of the current `Session` object.

Member	Brief description (See also the reference entry)
unsubscribe( ) method (JSDOSession class) on page 355	Unsubscribes a given event callback function from an event of the current `JSDOSession` object.
unsubscribe( ) method (Session class) on page 356	Unsubscribes a given event callback function from an event of the current `Session` object.
unsubscribeAll( ) method on page 356	Unsubscribes all event callback functions from a single named event of the current JSDO, `JSDOSession` or `Session` object, or unsubscribes all event callback functions from all events of the current JSDO, `JSDOSession`, or `Session` object.

You can identify a change in a client app's session online status by subscribing handlers to the `online` and `offline` events of a `JSDOSession` or `Session` instance using the object's `subscribe( )` method any time after you create the object. When the JSDO login session detects that its previously online device has gone offline or its previously connected web application is no longer available, the session fires its `offline` event. When the session detects that its previously offline device has come back online or its previously unavailable web application connection has, again, become available, the session fires its `online` event.

When a subscribed event fires, the handler signature includes:

- A reference to the session instance on which the event was fired
- A reference to a `request` object (if any) that was sent for a Data Object Service request that triggered the event
- For the `offline` event, a reason that an offline status was detected, such as its device going offline.

Note: If the `offline` or `online` event fires because of a change in device online status, no `request` object reference is returned and the handler's `request` object parameter is set to `null`.

A JSDO login session supports several ways to identify either the device online status or the web application online status for Data Object Services managed by a session instance:

- Subscribing handlers to the `online` and `offline` events of the session instance using the instance's `subscribe( )` method, as described above. Again, these events fire whenever there is a change in session online status for any reason.
- Invoking the `ping( )` method on the session instance. This method returns the current online status of the session instance. It sends a request to the web application that is managed by the session instance and attempts to determine if the Data Object Services provided by the web application are available.

  On a `JSDOSession` instance, this method immediately returns a Promise object on which you can register various callback functions using Promise object methods. These callbacks execute depending on the conditions of method completion. For example, if the callback executes that you register using the `done( )` method, the `JSDOSession` instance is online with access to its Data Object Services. If the callback executes that you register using the `fail( )` method, the `JSDOSession` instance is offline either because the device is offline or there is a problem with the web application or Data Object Service connections.

  On a `Session` instance, this method can execute either asynchronously, with results returned in a callback function that you specify, or synchronously with a `boolean` value to indicate the online status and properties of an object parameter that are set to provide more information.

In addition, if the current online status identified by `ping( )` has changed from the most recent status detection (either from a call to `ping( )` or a JSDO request for a Data Object Service), the session instance also fires its corresponding `offline` or `online` event, to which any subscribed event handler can respond. Note that `ping( )` returns online status results **only** between a successful execution of the `login( )` method and successful execution of the `logout( )` method on the same session instance.

- Setting the `pingInterval` property on the session instance to a value in milliseconds greater than zero (0). This setting causes the instance to automatically and periodically invoke its `ping( )` method once its `login( )` method has successfully executed. The minimum time between the completion of one execution of `ping( )` and the start of the next execution of `ping( )` is the number of milliseconds you assign. Each automatic execution of `ping( )` only fires an `offline` or `online` event on its session instance if the method detects a change in session online status. Automatic `ping( )` execution returns no other results.
- Testing the value of the `connected` property on the session instance. This `boolean` value indicates the most recent online status of the current session instance when it last determined if the web application it manages was available. If the property value is `true`, the instance most recently determined that the session is connected and logged in to its web application. If its value is `false`, the session was last found to be disconnected.

  This property is first set to `true` from a successful invocation of the `login( )` method on the session instance. However, to obtain the **most current** online status, if different, you must either invoke the instance's `ping( )` method or make a JSDO request to one of the instance's Data Object Services, which updates the value of its `connected` property if its online status has changed. This property is also set to `false` after you invoke the `logout( )` method on its session instance.

Note that from the point of successful invocation of `login( )` on a session instance, a JSDO login session is always initially online. So, the first **change** in device or Data Object Service online status is indicated by a firing of the `offline` event by the session instance in response to detecting that the device or its most recently connected web application has gone offline.

Note also that all attempts to detect the current availability of Data Object Services provided by a connected web application, such as executing `ping( )` or making JSDO requests, return results from the hosting web server **only** after successfully starting and maintaining a user login session on the web application from a device that is online (that is, by invoking `login( )` on a session instance for the web application and **not** yet invoking `logout( )`). However, you can always receive results from changes in the device's online status by subscribing to `offline` and `online` events from the first point that you create a session instance (and regardless if you have yet started a JSDO login session) until you either destroy the session instance or unsubscribe to these events using the `unsubscribe( )` or `unsubscribeAll( )` methods.

Using protected web resources

It is possible to configure a client app so that, prior to logging in, it opens a protected web resource. A protected web resource is typically a web page provided by a web application that is protected according to either the HTTP Basic or HTTP Form-based authentication model. If the client app attempts to open a protected web page **prior** to obtaining and authenticating the user's credentials with a call to the `login( )` method on a newly instantiated `JSDOSession` or `Session` object, either the browser running a web app, or the mobile device container running a hybrid app, automatically prompts the user for login credentials and sends them directly to the web application for authentication.

Note: For typical web applications configured to use HTTP Form-based authentication, when a client app tries to access a protected web resource without prior authorization, the web application sends a web page containing a login form to the web client for the user to enter login credentials for authentication. However, the JSDO handles HTTP Form-based authentication differently, and never uses the web page that is sent to a client app. Instead, the JSDO relies on the client app to provide the login form, exactly the same as for HTTP Basic authentication.

If authentication for access to this protected web resource succeeds, the web application then returns the protected web resource to the browser or device container for display to the user. At this point, the client app must **also** call the `login( )` method to initiate a login session in the web application in order to access any of its Data Object Services. However, in this case, because user authentication has already occurred, there is no need to pass user credentials, and if any are passed, the `login( )` method ignores them. In fact, the client app does not even know the username provided by the user who authenticated against the protected web page prior to this call to the `login( )` method.

In general, because of differences between typical app and JSDO support for HTTP Form-based authentication, with its separation between user authentication and the need to call the `login( )` method to start a JSDO login session, configuring a client app to access a protected web page prior to invoking the `login( )` method is not a typical client app configuration when using a JSDO. Instead, Progress recommends that a client app **only** access unprotected web and Data Object resources **until** the app explicitly prompts for user credentials using an unprotected login page that the client app provides, then calls the `login( )` method to both authenticate the user and start the required JSDO login session for the web application.

That being said, Progress **also** recommends, and helps to ensure, that the `login( )` method **itself** accesses a protected web resource as a login target when starting the JSDO login session. In general, you need to be sure that security is configured to complete authentication and load the Data Service Catalog **before** a client app requests resources in the Catalog. Although it is **possible** to configure web application security so that Data Object resources in the Data Service Catalog are the only resources that require web server authentication as part of the call to the `addCatalog( )` method, doing so can cause problems when a client app that otherwise authenticates successfully tries to access the Data Object Services that the web application provides.

In other words, in certain situations where a Data Object resource is the first web application resource to require authentication, a client app can be prevented from accessing the resource even after an authenticated JSDO login session starts successfully. Therefore, Progress provides defaults so the `login( )` method always accesses a protected non-UI web resource before the client app attempts to access a protected Data Object resource defined in the Data Service Catalog. Once the user is authenticated against this protected web resource, the web server can then reliably provide access to similarly protected resources of the web application—including Data Object resources—according to the user's authorization settings (roles). For more information on the default protected web resource for the `login( )` method, see Using default web pages to support client app login on page 99.

Dynamic data binding for alternate client platforms

If you do not need the UI support provided by Kendo UI, or have other client requirements, you can build client apps using the basic HTML and JavaScript coding tools of Progress Developer Studio for OpenEdge. For this purpose, the JSDO provides a progress.ui.UIHelper class on page 142 to map JSDO data to HTML elements using supported UI frameworks, such as jQuery or jQuery Mobile.

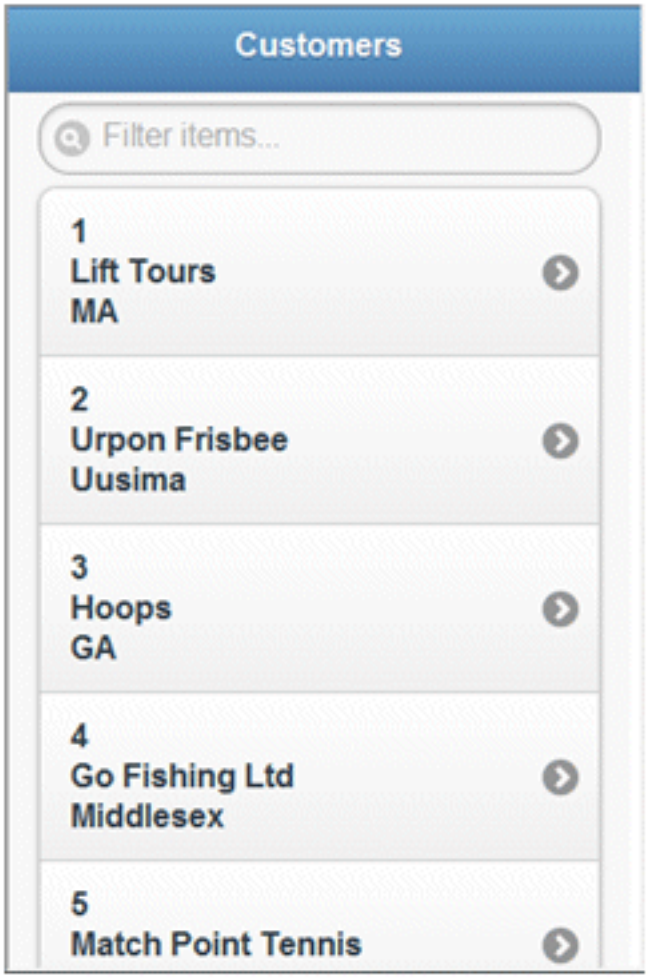
The `UIHelper` class assists in building mobile or web apps with a list view and a detail page. It can be used in the following scenarios and JavaScript frameworks:

- jQuery Mobile
- iUI

- User interface based on HTML and JavaScript where list views are built directly using the `<li>` element

The following figure shows a sample screen generated using the `UIHelper` class

Figure 1: Sample display using the UIHelper class



Using the UIHelper class

The `UIHelper` is instantiated to work with a JSDO instance. The `setListView( )` method is used to specify the HTML element for the list (`<ul>` element). The `clearItems( )` and `addItem( )` methods are used to build the list view. The `showListView( )` method is used to show the list view on the screen.

Note: For complete information on the methods of the `UIHelper` class, see JSDO properties, methods, and events reference on page 149.

The following is a sample HTML file using JQuery Mobile (`index.html`):

Table 10: Sample HTML file using JQuery Mobile (index.html)

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1" />
        <title>Customers</title>
        <link rel="stylesheet"
href="http://code.jquery.com/mobile/1.1.0/jquery.mobile-1.1.0.min.css" />
        <style>
            /* App custom styles */
        </style>
        <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js">
        </script>
        <script src="http://code.jquery.com/mobile/1.1.0/jquery.mobile-1.1.0.min.js">
        </script>
        <script src="progress.session.js"></script>
        <script src="progress.js"></script>
        <script src="customers.js"></script>
    </head>
    <body>
        <div data-role="page" id="custlist">
            <div data-theme="b" data-role="header">
                <h3>Customers</h3>
            </div>
            <div data-role="content">
                <ul id="listview" data-role="listview" data-divider-theme="b"
data-inset="true" data-filter="true">
                </ul>
            </div>
        </div>
        <div data-role="page" id="custdetail" data-add-back-btn="true" data-theme="b">
        <div data-theme="b" data-role="header"><h1>Customer</h1></div>
        <div data-role="content">
            <form action="" id="customerform">
            </form>
        </div>
        </div>
    </body>
</html>
```

The following is the `customers.js` script used in the sample HTML.

Table 11: Sample JavaScript file using JQuery Mobile (customers.js)

```
var customers;
var uihelper;
var forminitialized = false;

$(document).ready(function() {
    var session = new progress.data.Session();

    session.login("http://localhost:8980/SportsApp", "", "");
    session.addCatalog(
      'http://localhost:8980/SportsApp/static/mobile/CustomerOrderSvc.json'
    );
    customers = new progress.data.JSDO({ name: 'CustomerOrder' });
    customers.subscribe('AfterFill', onAfterFillCustomers, this);
    uihelper = new progress.ui.UIHelper({ jsdo: customers });
    uihelper.eCustomer.setDetailPage({ name: 'custdetail' });
    uihelper.eCustomer.setListView({
        name:'listview',
        format: '{CustNum}<br>{Name}<br>{State}',
        autoLink: true
    });
    $('#customerform').html(
       uihelper.eCustomer.getFormFields()
            + '<input type="button" value="Add" id="btnAdd" />'
            + '<input type="button" value="Save" id="btnSave" />'
            + '<input type="button" value="Delete" id="btnDelete" />'
    );
    customers.fill();
});

function onAfterFillCustomers() {
    uihelper.eCustomer.clearItems();
    customers.eCustomer.foreach(function(customer) {
        uihelper.eCustomer.addItem();
    });
    uihelper.eCustomer.showListView();
}
```

The `setDetailPage( )` method is used to specify the name of the HTML element, generally a `<div>`, that represents the detail page.

The `getFormFields( )` method can be used to obtain the HTML text for the fields of the specified table reference. This HTML text is generally added to the HTML element representing the form in the detail page. This element does not need to be a `<form>` element; it can be a `<div>` element.

The format property of the initialization object for `setListView( )` contains substitution parameters that are replaced when `addItem( )` is called: `{CustNum}`, `{Name}`, and `{State}`. The working record of the specified table reference is used to determine the values of the fields. In the sample `customer.js` script, the call to `addItem( )` queries the working record to obtain the values of the fields `CustNum`, `Name`, and `State`. The `addItem( )` method then builds a list item using a default template for list items.

Using a custom template

You can specify a different template for the list items using one of the following methods:

- Using the `itemTemplate` property in the initialization object for `setListView( )`.
- Calling the class-level method, `progress.ui.UIHelper.setItemTemplate( )`.
- By not specifying the `format` property. In this case, the `UIHelper` uses the first item element in the list view as the template for the items.

You can also specify a different template for the fields returned when calling `getFormFields( )`:

- Specify the `fieldTemplate` property in the initialization object for `setDetailPage( )`.
- Call the class-level method, `progress.ui.UIHelper.setFieldTemplate( )`.

Alternatively, you can define the layout for the detail page using HTML instead of calling `getFormFields( )`.

The default item template looks like this:

```
<li data-theme="c" data-id="{__id__}">
    <a href="#{__page__}" class="ui-link" data-transition="slide">{__format__}</a>
</li>
```

The default field template looks like this:

```
<div data-role="fieldcontain">
    <label for="{__name__}">{__label__}</label>
    <input id="{__name__}" name="{__name__}" placeholder="" value="" type="text" />
</div>
```

The templates use the following substitution parameters:

- Used when building the list items by calling `addItem( )`:
  - `{__id__}` — The internal ID of the record
  - `{__page__}` — The `name` attribute of the object passed as a parameter to `setDetailPage( )`, which defines the detail page
  - `{__format__}` — The `format` attribute of the object passed as a parameter to `setListView( )` or (optionally) to `addItem( )`, which identifies the fields to be included for each item in the list view
- Used for the HTML text returned when calling `getFormFields( )`:
  - `{__name__}` — The field name in a table as defined by the `name` property in the catalog
  - `{__label__}` — The label of the field in a table as defined by the `title` property in the catalog

The properties `itemTemplate` and the `fieldTemplate` can be used to change the template for a specific `UIHelper` instance.

For example:

```
uiHelper.eCustomer.setListView({
    name: 'cust-listview',
    format: '{CustNum} {Name}<br>{Address}',
    autoLink: true,
   itemTemplate: '<li data-id="{__id__}"><a href="#{__page__}" >{__format__}</a></li>'
});
```

```
uiHelper.eCustomer.setDetailPage({
    name: 'cust-detail-page',
    fieldTemplate: '<input id="{__name__}"></input>'
});
```

The class-level methods `progress.ui.UIHelper.setItemTemplate( )` and `progress.ui.UIHelper.setFieldTemplate( )` can be used to change the template to be used by the `UIHelper` for a JavaScript session.

For example:

```
progress.ui.UIHelper.setItemTemplate('<li data-id="{__id__}"><a href="#{__page__}"
>{__format__}</a></li>');
```

```
progress.ui.UIHelper.setFieldTemplate('<input id="{__name__}"></input>');
```

6

JSDO class and object reference

The classes and objects described in this reference to support Progress Data Objects comprise the entire implementation for the client-side JavaScript Data Object (JSDO). Each class or object description lists its members by category: properties, methods, and events, with a short description of each member. For a detailed description of each member, see JSDO properties, methods, and events reference on page 149.

Note: JavaScript is a case-sensitive language. So, class type, property, method, and event names, as well as other defined language elements (such as data types) must have the specified letter case.

Note: In addition to the documented members, these classes and objects might also contain undocumented members that are reserved for internal use by OpenEdge.

For details, see the following topics:

- progress.data.JSDO class
- progress.data.JSDOSession class
- progress.data.JSRecord class
- progress.data.PluginManager class
- progress.data.Session class
- progress.ui.UIHelper class
- request object

progress.data.JSDO class

The `progress.data.JSDO` is a JavaScript class that provides access to resources (Data Objects) of a Progress Data Object Service. A single `progress.data.JSDO` object (JSDO instance) provides access to a single resource supported by a given Data Object Service. This resource implements a given Data Object depending on the type of Data Object Service that provides it. For an OpenEdge Data Object Service, the resource is implemented using an ABL Business Entity, which is a singleton class or persistent procedure running on an OpenEdge application server (Progress Application Server for OpenEdge or OpenEdge AppServer). For a Rollbase Data Object Service, the resource is implemented by a Rollbase object provided by a Rollbase server running in a public or private cloud.

Once created for a given *Data Object resource*, a JSDO provides application-oriented, JavaScript methods that invoke *Data Object operations* supported by its resource implementation. These can include standard operations for reading and writing data, and can also include custom operations for invoking business logic and more complex data operations, depending on the implementation. An OpenEdge resource maps specified ABL routines to any of several supported operation types, each of which corresponds to a particular method in the JSDO. A Rollbase resource maps a corresponding set of server-implemented APIs to these same JSDO methods.

For an OpenEdge Data Object Service, you (or another OpenEdge user) identify how a JSDO maps JavaScript methods to operations of a given resource by adding annotations to the singleton procedure or class source code. These annotations map the ABL routines that implement a given resource operation to the corresponding JavaScript method in the JSDO that invokes the operation. You specify these annotations using features of Progress Developer Studio for OpenEdge (Developer Studio) in two ways:

1. When initially creating an ABL Business Entity to implement a Data Object that supports the business operations you want on a specified data model
2. When modifying any existing ABL singleton class or procedure that you want to function as a Business Entity that implements a Data Object resource

You can also use Developer Studio to add a given resource to an existing OpenEdge Data Object Service, as part of generating the Service along with the client-side artifacts required to create a corresponding JSDO for the resource on the client. These client-side artifacts include the Data Service Catalog that defines the schema and operations for each resource you can access in the Service using a corresponding JSDO.

For a Rollbase Data Object Service, the available resources and their supported operations are automatically created from the existing definitions of Rollbase objects that you (or another Rollbase user) select to create its Rollbase Data Service Catalog. The resulting Catalog defines the mappings between JSDO methods and the Rollbase APIs used to implement the operations on each resource (Rollbase object) supported by the Data Object Service.

At run time, the JSDO maintains an internal data store (*JSDO memory*) for managing table records that are exchanged between a given Data Object resource and the JSDO, and it provides methods to read and write data in JSDO memory as individual JavaScript record objects (see the progress.data.JSRecord class on page 133 description). To support this data exchange between a JSDO and its Data Object resource, a given resource can support standard operation types that include *Create*, *Read*, *Update*, *Delete* (CRUD) operations, and *Submit* operations. The Read operation of a resource returns a specified result set from the resource to the client; the CUD operations create, update, or delete (respectively) one record at a time in the resource; and a Submit operation creates, updates, and deletes multiple records at a time in the resource. In addition to these standard operations, a resource can also support custom *Invoke* operations to access ABL business logic in a given OpenEdge resource or to read data from Rollbase objects that are related to a given Rollbase object resource.

The standard CRUD operations can operate on any OpenEdge or Rollbase single-table resource, or on an OpenEdge multi-table resource (ProDataSet) containing one or more temp-tables. Standard Submit operations are currently available **only** on an OpenEdge ProDataSet resource that supports before-imaging (with potentially complex server transaction support).

Each standard Data Object operation type (CRUD or Submit) maps to a corresponding standard method of the JSDO on the client. The records of each resource table are managed in the JSDO as an array of record objects, which the standard JSDO methods use to exchange data with the server as supported by the type of Data Object Service. The standard JSDO methods mapped to standard operation types support client access to the common business operations that can be generated for either an OpenEdge Business Entity or a Rollbase object. Other standard methods of the JSDO provide internal access to individual record objects in JSDO memory. Depending on the results of its standard methods, the JSDO also maintains a working record for each table in its memory that you can access directly using table and field references (see the notes in this class description). Thus, using the standard methods of a JSDO and its table references, you can interact with a corresponding Data Object resource in a consistent manner from one JSDO-resource mapping to another.

The JSDO methods to invoke the standard resource operations include `fill( )` to Read resource data and `saveChanges( )` to either Create, Update, or Delete individual records in the resource, or Submit to create, update, and delete multiple records in the resource. Both of these JSDO methods execute asynchronously. Any results, including errors, that are returned from both the resource operations and the JSDO method invocations themselves can be managed using two mechanisms:

1. **Named events** — To which you can subscribe event handlers (callback functions) to process the results when a given event is fired. The available events fire both before and after each resource operation executes, and both before and after execution of the JSDO method that invokes the operation, or operations. The callback for each event has a particular signature that is appropriate for the results returned for the event. See the descriptions of both the JSDO `fill( )` and `saveChanges( )` methods and the associated event descriptions for more information.
2. **jQuery Promises** — Deferred Promise objects, an instance of which can be returned as the method value when results are available. A Promise object provides methods that you can use to register callback functions that, in turn, execute for different conditions associated with the JSDO method results. In order to use jQuery Promises, your development environment must support them (or the exact equivalent). For more information on this mechanism, and how it works with the `fill( )` and `saveChanges( )` methods, see the notes for this class description.

If your environment supports jQuery Promises, you can use either or both of these mechanisms to return results for the standard resource operations.

Like the standard CRUD and Submit operations, a resource can define custom Invoke operations, depending on the type of resource (OpenEdge or Rollbase). For OpenEdge resources, an Invoke operation can call and return results for any ABL routine on the server that is defined by the resource. To call an Invoke operation for a specific ABL routine, an OpenEdge resource defines a corresponding *invocation method* that you can call on the JSDO to send the request the server. Rollbase resources can define similar invocation methods that you call on the JSDO to invoke server APIs that return records from related Rollbase objects.

Note that data exchanged between server and client using Invoke operations is not automatically stored in JSDO memory. It is initially accessible only from the return values and other results of the invocation methods provided by the JSDO. You can subsequently use other standard JSDO methods to exchange data between the invocation methods and JSDO memory. For example, you can use the JSDO `addRecords( )` method to merge appropriate data returned by an invocation method directly into JSDO memory. The result in JSDO memory can then continue to be maintained and synchronized with the server using the JSDO `fill( )` and `saveChanges( )` methods to invoke the standard CRUD and Submit operations.

Unlike the standard `fill( )` and `saveChanges( )` methods, you can execute custom JSDO invocation methods either synchronously or asynchronously, depending on parameter settings that you use to call them. If you execute an invocation method asynchronously, the JSDO also supports events that fire before and after the method (and its Invoke operation) execute in the resource, returning the results in any event handler callbacks that you subscribe to these events. In addition, you can use the standard JSDO `invoke( )` method as an API to asynchronously call custom invocation methods that return their results in jQuery Promise callbacks, similar to the `fill( )` and `saveChanges( )` methods, as long as your environment supports jQuery Promises.

Note that all JSDO methods that invoke either standard or custom resource operations return a request object to any subscribed event callbacks. This request object supports a number of properties containing the results of the resource operation, some of which appear in the request object only for specific events. Similarly, any Promise object returned by a JSDO method also returns an appropriate request object to each callback that you register for that object. For custom invocation methods that you execute synchronously, this request object is returned as the value of the invocation method. For more information on this request object, see the request object on page 146 description.

When you instantiate a JSDO, it relies on a prior JSDO login session that you can establish using an instance of either the `progress.data.JSDOSession` class or the `progress.data.Session` class. This login session enables optionally secure communications between the client JSDO and the web server, its supported Data Object Services, and ultimately the application server that implements the resource accessed by the JSDO.

A `JSDOSession` insatnce manages its JSDO login session asynchronously using only jQuery Promises to return all session-related results, where as a `Session` instance can manage its JSDO login session either synchronously or asynchronously using several `Session` object events. Note that, similar to managing resource operations using jQuery Promises returned from JSDO methods, you can use the `JSDOSession` class to manage JSDO login sessions only if your environment supports jQuery Promises. Otherwise, you must use the `Session` class to manage the login session for a JSDO. For more information, see the progress.data.JSDOSession class on page 124 and progress.data.Session class on page 136 descriptions.

Constructors

Two constructors are available for the JSDO. The first constructor takes the name of the corresponding Data Object resource as a parameter; the second constructor takes an initialization object as a parameter.

The resource name specified for the constructor must match the name of a resource provided by a Data Object Service for which a login session has already been established. After the JSDO is created, it uses the information stored in the Data Service Catalog to communicate with the specified resource.

Syntax

```
progress.data.JSDO ( resource-name )
progress.data.JSDO ( init-object )
```

resource-name

A `string` expression set to the name of a resource that the JSDO will access. This resource must be provided by a Data Object Service for which a login session has already been established.

init-object

An object that can contain any writable JSDO properties. It **must** contain the required JSDO `name` property, which specifies the Data Object resource that the JSDO will access. This resource must be provided by a Data Object Service for which a login session has already been established.

This *init-object* can also contain either or both of the following optional initialization properties:

- **autoFill** — A `boolean` that specifies whether the the JSDO invokes its `fill( )` method upon instantiation to initialize JSDO memory with data from its resource. The default value is `false`.
- **events** — An object that specifies one or more JSDO event subscriptions, each with its properties represented as an array, with the following syntax:

  Syntax

```
events : {
     'event' : [ {
          [ scope : object-ref , ]
          fn : function-ref
     } ] [ ,
     'event' : [ {
          [ scope : object-ref , ]
          fn : function-ref
     } ] ] ...
}
```

  event

  The name of an event to which the JSDO instance subscribes callbacks. See Events on page 119 for a list of available JSDO events.

  object-ref

  An optional object reference that defines the execution scope of the callback function invoked when the event fires. If the `scope` property is omitted, the execution scope is the global object (usually the browser or device window).

  function-ref

  A reference to the callback function that is invoked when the event fires.

  Each event passes a fixed set of parameters to its event callback, as described for the event. A reference to the description of each JSDO event is provided later in this class description.

Example

The following example illustrates the use of an initialization object to instantiate a JSDO:

```
dsItems = new progress.data.JSDO({
  name : 'Item',
  autoFill : false,
  events : {
    'afterFill' : [ {
                      scope : this,
                      fn : function (jsdo, success, request) {
                        // afterFill event handler statements ...
                      }
                    } ]
  }
});
```

Properties

Table 12: progress.data.JSDO properties

Member	Brief description (See also the reference entry)
autoApplyChanges property on page 202	A `boolean` on a JSDO that indicates if the JSDO automatically accepts or rejects changes to JSDO memory when you call the `saveChanges( )` method.
autoSort property on page 203	A `boolean` on a JSDO and its table references that indicates if record objects are sorted automatically on the affected table references in JSDO memory at the completion of a supported JSDO operation.
caseSensitive property on page 213	A `boolean` on a JSDO and its table references that indicates if `string` field comparisons performed by supported JSDO operations are case sensitive or case-insensitive for the affected table references in JSDO memory.
name property (JSDO class) on page 290	Returns the name of the Data Object resource for which the current JSDO was created.
record property on page 305	A property on a JSDO table reference that references a `JSRecord` object with the data for the working record of a table referenced in JSDO memory.
table reference property (JSDO class) on page 351	An object reference property on a JSDO that has the name of a corresponding table in the Data Object resource for which the current JSDO was created.
useRelationships property on page 357	A `boolean` that specifies whether JSDO methods that operate on table references in JSDO memory recognize and honor *data-relations* defined in the schema (that is, work only on records of a child table that are related to the working record of a parent table).

Methods

Certain methods of the `progress.data.JSDO` class are called on the JSDO object itself, without regard to a table reference, whether that reference is explicit (specified in the method signature) or implicit (in the case of a JSDO containing only one table that is not explicitly specified). Other methods can be called on a specific table reference that is mapped to one of possibly multiple tables managed by the resource for which the current JSDO is created.

Table 13: progress.data.JSDO class-instance methods

Member	Brief description (See also the reference entry)
acceptChanges( ) method on page 153	Accepts changes to the data in JSDO memory for the specified table reference or for all table references of the specified JSDO.
acceptRowChanges( ) method on page 156	Accepts changes to the data in JSDO memory for a specified record object.
addLocalRecords( ) method on page 169	Reads the record objects stored in the specified local storage area and updates JSDO memory based on these record objects, including any pending changes and before-image data, if they exist.
addRecords( ) method on page 175	Updates JSDO memory with one or more record objects read from an array, single-table, or multi-table resource that are passed in an object parameter, including any pending changes and before-image data, if they exist.
deleteLocal( ) method on page 218	Clears out all data and changes stored in a specified local storage area, and removes the cleared storage area.
fill( ) method on page 220 **Alias:** `read( )`	Initializes JSDO memory with record objects from the data records in a single-table resource, or in one or more tables of a multi-table resource, as returned by the Read operation of the Data Object resource for which the JSDO is created.
getProperties( ) method on page 245	Returns an object containing the names and values of the user-defined properties defined in the current JSDO instance. **Note:** Applies to Progress Data Objects Version 4.3 or later.
getProperty( ) method on page 246	Returns the value of the specified JSDO user-defined property. **Note:** Applies to Progress Data Objects Version 4.3 or later.
hasData( ) method on page 255	Returns `true` if record objects can be found in any of the tables referenced in JSDO memory (with or without pending changes), or in only the single table referenced on the JSDO, depending on how the method is called; and returns `false` if no record objects are found in either case.
hasChanges( ) method on page 253	Returns `true` if JSDO memory contains any pending changes (with or without before-image data), and returns `false` if JSDO memory has no pending changes.

Member	Brief description (See also the reference entry)
invocation method on page 262	A custom method on the JSDO that executes an Invoke operation defined for a Data Object resource.
invoke( ) method on page 264	Asynchronously calls a custom invocation method on the JSDO to execute an Invoke operation defined by a Data Object resource.
readLocal( ) method on page 304	Clears out the data in JSDO memory and replaces it with all the data stored in a specified local storage area, including any pending changes and before-image data, if they exist.
rejectChanges( ) method on page 305	Rejects changes to the data in JSDO memory for the specified table reference or for all table references of the specified JSDO.
rejectRowChanges( ) method on page 308	Rejects changes to the data in JSDO memory for a specified record object.
saveChanges( ) method on page 314	Synchronizes to the server all record changes (creates, updates, and deletes) pending in JSDO memory for the current Data Object resource since the last call to the `fill( )` or `saveChanges( )` method, or since any prior changes have been otherwise accepted or rejected.
saveLocal( ) method on page 330	Saves JSDO memory to a specified local storage area, including pending changes and any before-image data, according to a specified data mode.
setProperties( ) method on page 337	Replaces all user-defined properties in the current JSDO instance with the user-defined properties defined in the specified object. **Note:** Applies to Progress Data Objects Version 4.3 or later.
setProperty( ) method on page 338	Sets the value of the specified JSDO user-defined property. **Note:** Applies to Progress Data Objects Version 4.3 or later.
subscribe( ) method (JSDO class) on page 347	Subscribes a given event callback function to a named event of the current JSDO or table reference.

Member	Brief description (See also the reference entry)
unsubscribe( ) method (JSDO class) on page 353	Unsubscribes a given event callback function from a named event of the current JSDO or table reference.
unsubscribeAll( ) method on page 356	Unsubscribes all event callback functions from a single named event of the current JSDO, `JSDOSession` or `Session` object, or unsubscribes all event callback functions from all events of the current JSDO, `JSDOSession`, or `Session` object.

Table 14: progress.data.JSDO table-reference methods

Member	Brief description (See also the reference entry)
acceptChanges( ) method on page 153	Accepts changes to the data in JSDO memory for the specified table reference or for all table references of the specified JSDO.
acceptRowChanges( ) method on page 156	Accepts changes to the data in JSDO memory for a specified record object.
add( ) method on page 158 **Alias:** `create( )`	Creates a new record object for a table referenced in JSDO memory and returns a reference to the new record.
addRecords( ) method on page 175	Updates JSDO memory with one or more record objects read from an array, single-table, or multi-table resource that are passed in an object parameter, including any pending changes and before-image data, if they exist.
assign( ) method (JSDO class) on page 198 **Alias:** `update( )`	Updates field values for the specified `JSRecord` object in JSDO memory.
find( ) method on page 227	Searches for a record in a table referenced in JSDO memory and returns a reference to that record if found. If no record is found, it returns `null`.
findById( ) method on page 229	Locates and returns the record in JSDO memory with the internal ID you specify.
foreach( ) method on page 231	Loops through the records of a table referenced in JSDO memory and invokes a user-defined callback function as a parameter on each iteration.
getData( ) method on page 233	Returns an array of record objects for a table referenced in JSDO memory.

Member	Brief description (See also the reference entry)
getErrors( ) method on page 233	Returns an array of errors from the most recent invocation of Create, Read, Update, Delete, or Submit operations (CRUD or Submit) that you have invoked by calling the JSDO `fill( )` or `saveChanges( )` method on a Data Object resource. **Note:** Updated for Progress Data Objects Version 4.3 or later.
getErrorString( ) method on page 239	Returns any before-image error string in the data of a record object referenced in JSDO memory that was set as the result of a resource Create, Update, Delete, or Submit operation.
getId( ) method on page 242	Returns the unique internal ID for the record object referenced in JSDO memory.
getSchema( ) method on page 247	Returns an array of objects, one for each field that defines the schema of a table referenced in JSDO memory.
hasData( ) method on page 255	Returns `true` if record objects can be found in any of the tables referenced in JSDO memory (with or without pending changes), or in only the single table referenced on the JSDO, depending on how the method is called; and returns `false` if no record objects are found in either case.
rejectChanges( ) method on page 305	Rejects changes to the data in JSDO memory for the specified table reference or for all table references of the specified JSDO.
rejectRowChanges( ) method on page 308	Rejects changes to the data in JSDO memory for a specified record object.
remove( ) method on page 311	Deletes the specified table record referenced in JSDO memory.
setSortFields( ) method on page 339	Specifies or clears the record fields on which to automatically sort the record objects for a table reference after you have set its `autoSort` property to `true` (the default).
setSortFn( ) method on page 341	Specifies or clears a user-defined sort function with which to automatically sort the record objects for a table reference after you have set its `autoSort` property to `true` (the default).
sort( ) method on page 344	Sorts the existing record objects for a table reference in JSDO memory using either specified sort fields or a specified user-defined sort function.

Member	Brief description (See also the reference entry)
subscribe( ) method (JSDO class) on page 347	Subscribes a given event callback function to a named event of the current JSDO or table reference.
unsubscribe( ) method (JSDO class) on page 353	Unsubscribes a given event callback function from a named event of the current JSDO or table reference.

Events

Table 15: progress.data.JSDO events

Member	Brief description (See also the reference entry)
afterCreate event on page 181	Fires after the JSDO, by means of a `saveChanges( )` call following an `add( )` call, sends a request to create a record in the Data Object resource and receives a response to this request from the server.
afterDelete event on page 183	Fires after the JSDO, by means of a `saveChanges( )` call following a `remove( )` call, sends a request to delete a record in the Data Object resource and receives a response to this request from the server.
afterFill event on page 185 **Alias:** `afterRead`	Fires on the JSDO after a call to the `fill( )` method executes and returns the results from the server to JSDO memory for a Read operation request on its Data Object resource.
afterInvoke event on page 186	Fires after a custom invocation method is called asynchronously on a JSDO and a response to the Invoke operation request is received from the server.
afterSaveChanges event on page 192	Fires once for each call to the `saveChanges( )` method on a JSDO, after responses to all create, update, and delete record requests sent to a Data Object resource have been received from the server.
afterUpdate event on page 195	Fires after the JSDO, by means of a `saveChanges( )` call following an `assign( )` call, sends a request to update a record in the Data Object resource and receives a response to this request from the server.
beforeCreate event on page 206	Fires before the JSDO, by means of a `saveChanges( )` call following an `add( )` call, sends a request to create a record in the Data Object resource on the server.

Member	Brief description (See also the reference entry)
beforeDelete event on page 207	Fires before the JSDO, by means of a `saveChanges( )` call following a `remove( )` call, sends a request to delete a record in the Data Object resource on the server.
beforeFill event on page 208 **Alias:** `beforeRead`	Fires on the JSDO before a call to the `fill( )` method executes and sends a Read operation request to its Data Object resource.
beforeInvoke event on page 209	Fires when a custom invocation method is called asynchronously on a JSDO before the request for the Invoke operation is sent to the server.
beforeSaveChanges event on page 210	Fires once on the JSDO before a call to the `saveChanges( )` method sends the first request to create, update, or delete a record in its Data Object resource on the server.
beforeUpdate event on page 211	Fires before the JSDO, by means of a `saveChanges( )` call following an `assign( )` call, sends a request to update a record in the Data Object resource on the server.

The JSDO can subscribe to the events listed in the previous table in either of two ways:

- **Using the JSDO constructor** — In the *`init-object`* parameter of the constructor, list each subscribed event with an optional scope object and an event handler method to be executed when the event fires. See the constructors description for this class.
- **Using the `subscribe( )` method** — See the subscribe( ) method (JSDO class) on page 347.

Note: JSDO events do not fire for custom invocation methods if the method call is synchronous.

Example

The following code fragment shows a simple JavaScript coding sequence for creating and using a JSDO referenced by the variable `dsCustomer` to read resource data, starting with the attempted JSDO login using a `progress.data.JSDOSession` instance:

Table 16: Example — Using a JSDO

```
// create a JSDOSession for a specified web application
// that requires Form authentication
var pdsession = new progress.data.JSDOSession( {
  serviceURI : 'https://testmach:8943/SportsMobile',
  authenticationModel : progress.data.Session.AUTH_TYPE_FORM
});

// log in by authenticating to the specified web application
pdsession.login(username, password).done(
  function( session, result, info ) {
    // login().done handler statements
    // load Catalog for a service that is hosted by the web application
   session.addCatalog('https://testmach:8943/SportsMobile/static/SportsMobileSvc.json'

      ).done(
      function( session, result, details ) {
        // addCatalog().done() handler statements
        // create JSDO for dsCustomer resource
        var dsCustomer = new progress.data.JSDO('dsCustomer');
        // load JSDO memory with resource data
        dsCustomer.fill().done(
          function (jsdo, success, request) {
            // fill().done() handler statements ...
          }).fail(
          function (jsdo, success, request) {
            // fill().fail() handler statements ...
          });
      }).fail(
      function( session, result, details ) {
        // addCatalog().fail() handler statements ...
      });
  }).fail(
  function( session, result, info ) {
    // login().fail() handler statements ...
  });
```

Notes

- Three web-communicating methods of this class, `fill( )`, `saveChanges( )`, and `invoke( )`, always execute asynchronously and return a reference to a jQuery Promise object as the return value when jQuery Promises are supported in the development environment. A Promise is a deferred object that defines methods which, in turn, register callbacks that execute when results are available from the method that returns the Promise. You provide your own code in each callback to handle the results returned for a given Promise method, much like the code you provide in an event callback.

  The primary Promise methods for use with a JSDO object include `done( )`, `fail( )`, and `always( )`, which allow you to separately handle the successful, unsuccessful, and all results (respectively) of a JSDO method execution. Note that for any given JSDO method, the parameter lists for all Promise method callbacks have the same signature. Note also that the callback for the `always( )` method executes with the same values in its parameter list as the callback for the `done( )` or `fail( )` method, whichever one executes with results.

  For more information on the jQuery implementation of Promises and the additional Promise methods that are available to register callbacks, see the following jQuery topics at the specified web locations:

  - **Category: Deferred Object** — http://api.jquery.com/category/deferred-object/
  - **Promise Object** — http://api.jquery.com/Types/#Promise
  - **.promise** — http://api.jquery.com/promise/

- For more information on defining a Data Object resource, including the standard CRUD, Submit, and any custom Invoke operations, for:
  - **OpenEdge Data Object Services** — See the topics on creating Business Entities to implement Data Object resources and updating Business Entities for access by mobile and web apps built using Kendo UI in *OpenEdge Development: Web Services*.
  - **Rollbase Data Object Services** — See Creating Progress Data Catalogs for use with AppBuilder in the Rollbase documentation. Note that Rollbase automatically defines CRUD and Invoke operations only that are supported by each single-table resource according to the definition of the Rollbase object from which the resource is created.

- The JSDO supports a *working record* for each table referenced in JSDO memory. Certain JSDO methods set a specific record as the working record. After other methods execute, there is either no working record set or existing working records remain unchanged. When there is a working record, you can access the fields of the record using one of the following mechanisms:

  Syntax

```
jsdo-ref.table-ref.field-ref
jsdo-ref.field-ref
jsdo-ref.record.data.field-ref // Read-only;
                               // For a jsdo-ref with only one table-ref
jsdo-ref.table-ref.record.data.field-ref // Read-only
jsrecord-ref.data.field-ref // Read-only
```

  jsdo-ref

  The reference to a JSDO, and if the JSDO contains only a single table, an implied reference to any working record that is set for that table.

  table-ref

  A table reference property with the name of a table in *jsdo-ref* memory and a reference to the table working record. There is one table reference in a JSDO for each table referenced by the JSDO. For more information on table reference properties, see the table reference property (JSDO class) on page 351 description.

  field-ref

  A field reference property on a *table-ref* or on the `data` property object, with the name and value of a field in the working record of the referenced table. There is one such field reference and `data` object property for each field defined in the table schema. If *field-ref* is an array field, which is supported in OpenEdge resource tables for example, you can also access individual array elements using either standard JavaScript subscripts or JSDO array-element references in the form, *field-ref_integer*, where *integer* is a one (1)-based integer that qualifies the name of the field reference property that corresponds to the array field element. For more information on field reference properties, see the table reference property (JSDO class) on page 351 description.

  `record`

  A property of type `JSRecord` on a table reference, which references the working record of a referenced table specified by:

  - *jsdo-ref.table-ref*

- *jsdo-ref* if the JSDO references only one table

If the JSDO references more than one table, the `record` property is `null` at the JSDO level and is available only on a *table-ref*.

`data`

A property on a `JSRecord` object with the field values for the working record specified by:

- *jsdo-ref.table-ref*
- *jsdo-ref* if the JSDO references only one table
- A *jsrecord-ref* returned for an associated JSDO table reference

---

Note: If a *field-ref* has the same name as a built-in property or method of the JSDO, you **must** use the `data` property to reference its value in the working record.

Caution: **Never write** directly to a *field-ref* using this `data` property; in this case, use *field-ref* **only to read** the data. Writing field values using the `data` property does **not** mark the record for update when calling the `saveChanges( )` method, nor does it re-sort the record in JSDO memory according to any order you have established using the `autoSort` property. To mark a record for update and automatically re-sort the record according to the `autoSort` property, you must assign a field value either by setting a *jsdo-ref.table-ref.field-ref* for a working record or by calling the `assign( )` method on a valid *table-ref* or `JSRecord` object reference.

---

jsrecord-ref

A reference to a `JSRecord` object for a table referenced in JSDO memory. You can return a *jsrecord-ref* for a working record as the value of the `record` property or as a value returned by supported JSDO built-in methods that return a working record, such as `add( )` and `find( )`.

For more information on properties available to reference working record fields using this syntax, see the properties listed in this class description and in the description of the progress.data.JSRecord class on page 133. For more information on the methods for setting the working record for referenced tables, see the methods listed in this class description and in the description of the progress.data.JSRecord class on page 133.

- Many standard JSDO methods are actually invoked on a JSDO table reference, and can only be invoked on the JSDO itself when its JSDO memory is initialized with a single table.
- For a multi-table resource, such as an OpenEdge ProDataSet, the JSDO accesses the data for all unrelated tables in JSDO memory as top-level tables of the JSDO. Access to data for all related child tables depends on the working record of the parent table in the JSDO and the setting of the `useRelationships` property.

See also

progress.data.JSRecord class on page 133, progress.data.Session class on page 136, record property on page 305, table reference property (JSDO class) on page 351, table reference property (UIHelper class) on page 353

progress.data.JSDOSession class

The `progress.data.JSDOSession` is a JavaScript class that provides methods, properties, and events to create and manage a JSDO login session. A *JSDO login session* includes a single end point (web application) and a single authentication model (Anonymous, HTTP Basic, or HTTP Form), and manages user access to Progress Data Object resources using instances of the `progress.data.JSDO` class (JSDO). This includes loading the definitions for the Data Object resources that JSDOs can access and starting and managing JSDO login sessions on web servers that provide access to these resources.

This `JSDOSession` class supports similar features to the `progress.data.Session` class, except that `JSDOSession` methods for loading Data Service Object resource definitions and managing login sessions **only** run asynchronously and return jQuery Promises to handle the results in callbacks registered using Promise methods, whereas the equivalent `progress.data.Session` methods can either run synchronously or fire asynchronous events to allow results to be handled in separately registered callbacks. In addition, the information required to instantiate and start JSDO login sessions with `JSDOSession` objects is handled somewhat differently than with `Session` objects, including optional support for web app page refresh (available in Progress Data Objects Version 4.3 or later).

Note: If your development environment does not support jQuery Promises (or the exact equivalent), you must use the `progress.data.Session` class to manage login sessions for JSDOs. For more information, see the description of the progress.data.Session class on page 136.

Like a `Session` object, a `JSDOSession` object manages user authentication and session identification information in HTTP/S messages sent between JSDOs running in a mobile app and Data Object Services running on a web server, each of which provide access to a set of Data Object resources. The authentication information includes a username and password (*user credentials*), if necessary, to authenticate a JSDO login session in a web application, which provides a REST transport between a set of Data Object Services and any client mobile app that accesses them. The session identification information includes the URI for the web application and might include a session ID that identifies the JSDO login session and helps to manage interactions between the client and the web application.

To start a JSDO login session for a `JSDOSession` object to manage, first instantiate the `JSDOSession` for a particular web application URI and authentication model as required by the web application's security configuration. Then invoke the object's `login( )` method. Pass the `login( )` method any required user credentials to authenticate `JSDOSession` access to the web application according to its authentication model. Once started, a login session for a web application supports all Data Object Services that the application provides, each of which can define one or more resources for access by JSDOs.

Each Data Object Service provides a separate JSON file (Data Service Catalog) that defines the schema for its set of resources and the operations to communicate between these resources and the JSDO instances that access them from a mobile app. To create a JSDO instance for a Data Object resource, a `JSDOSession` object must first load the Catalog that defines the resource using its `addCatalog( )` method. This method can load this Catalog from the web application, from some other web location, or from a location on the client where the mobile app runs, and the method can also accept credentials to access a Data Service Catalog that is protected separately from the web application itself.

Note: The standard mechanism supported by the JSDO framework to invoke most of the features of instantiating a `JSDOSession` object, establishing its JSDO login session, and loading a Data Service Catalog, is to invoke a single call to the `progress.data.getSession( )` stand-alone function. For more information, see the getSession( ) stand-alone function on page 248 description. (Available in Progress Data Objects Version 4.3 or later.) Also, see the Notes on page 131 on this class.

Once a Data Service Catalog is loaded into a `JSDOSession` object, you can instantiate a JSDO to access a particular resource defined in that Catalog. Once the `JSDOSession` is logged into its web application, the JSDO can then access its resource, relying on any authentication information for the login session (if necessary) to authorize this access.

Multiple JSDOs can thus rely on a single `JSDOSession` object to manage session access to all Data Object Services and their resources defined by the same web application.

Constructor

Instantiates a `JSDOSession` object that you can use to start and manage a JSDO login session in a web application and load the Data Service Catalog for each supported Data Object Service whose resources are accessed using JSDOs.

Syntax

```
progress.data.JSDOSession ( options )
```

options

An object that contains the following configuration properties:

- **`serviceURI`** — (Required) A `string` expression containing the URI of the web application for which to start a JSDO login session. This web application must support one or more Data Object Services in order to create JSDOs for the resources provided by the application. This URI is appended with a string that identifies a resource to access as part of the login process.

  If the mobile app from which you are logging in is a hybrid app that will be installed to run directly in a native device container, or if it is a web app deployed to a different web server from the web application that hosts the JSDO login session, you must specify `serviceURI` as an absolute URI to the Tomcat server domain or host and port, for example, `http://www.progress.com/SportsMobileApp`, or perhaps for testing, `http://testmach:8980/SportsMobileApp`.

  If the mobile app from which you are logging in is a web app deployed to the same Apache Tomcat server as the web application that hosts the JSDO login session, you can specify `serviceURI` as a relative URI, for example, `/SportsMobileApp`, which is relative to the deployment end point (Tomcat server domain or host and port).

  Note: Once the `login( )` method executes, the value you pass for `serviceURI` also sets the value of the `sessionURI` property on the current `JSDOSession` object, whether or not JSDO login completes successfully.

- **`authenticationModel`** — (Optional) A `string` constant that specifies one of the three authentication models that the `JSDOSession` object supports, depending on the web application configuration:

  - **`progress.data.Session.AUTH_TYPE_ANON`** — (Default) The web application supports Anonymous access. No authentication is required.
  - **`progress.data.Session.AUTH_TYPE_BASIC`** — The web application supports HTTP Basic authentication and requires a valid username and password. To have the `JSDOSession` object manage access to the web application's resources for you, you need to pass these credentials in a call to the `JSDOSession` object's `login( )` method. Typically, you would require the user to enter their credentials into a login dialog provided by your mobile app,

either using a form of your own design or using a template provided by Progress Software Corp.

- **progress.data.Session.AUTH_TYPE_FORM** — The web application uses Form-based authentication. Like HTTP Basic, Form-based authentication requires user credentials for access to protected resources; the difference is that the web application itself sends a form to the client to get the credentials. However, when you have the JSDOSession object manage access to the web application's resources, you handle Form-based authentication the same way that you handle Basic—get the user's credentials yourself and pass them to the login( ) method. The JSDOSession intercepts the form sent by the web application and handles the authentication without that form being displayed.

If the web application requires authentication, you must set this value correctly to ensure that users can log in.

For more information on these authentication models and how to configure them for a web application, see the sections on configuring web server authentication models in the administration documentation for the server hosting your Data Object Service.

- **name** — (Optional; available in Progress Data Objects Version 4.3 or later) A string with an operative value that you define to enable page refresh support for a JSDOSession object when it is instantiated for access by a client web app. The operative value can be any value other than the empty string (""), null, or undefined.

  If this page refresh support is enabled, when the web app successfully logs the newly instantiated JSDOSession into its server web application, the object stores the state of its JSDO login session using this name property value as a key. Then, at any time after log in, if the user initiates a browser refresh on the web app page, the JSDOSession object automatically identifies and restores its login session using this value. This helps to ensure, after a page refresh, that the web app does not need to prompt the user again for credentials in order to re-establish its connection to the web application for this JSDOSession.

  If you do not specify an operative value for name (the default), or you specify a non-operative value, such as the empty string (""), null, or undefined, the JSDOSession is instantiated without this page refresh support.

  For more information on how a login session is restored for a JSDOSession object with page refresh support, see the Notes on page 131 for this class description.

Note: If you pass the JSDOSession constructor the same value for name as an existing JSDOSession, it will return a JSDOSession using that same key. Both of them have the potential to overwrite each other. You must ensure that you pass a unique name value for each call to the JSDOSession constructor (or the getSession( ) stand-alone function).

Note: Page refresh supports **only** Anonymous and Form-based authentication. You cannot enable page refresh support when you set authenticationModel for a JSDOSession to progress.data.Session.AUTH_TYPE_BASIC; in this case, any setting of the name property is ignored. To enable page refresh support, you must set the authenticationModel property to either progress.data.Session.AUTH_TYPE_ANON or progress.data.Session.AUTH_TYPE_FORM.

Note: To help manage the results of a page refresh for any web app that accesses a JSDO, the JSDOSession class provides an isAuthorized( ) method that you can call to test that an authorized login session is established for a given JSDOSession object. For more information and an example, see the description of the isAuthorized( ) method on page 268.

Note: The standard mechanism supported by the JSDO framework to invoke most of the features of instantiating a `JSDOSession` object, establishing its JSDO login session, and loading a Data Service Catalog, is to invoke a single call to the `progress.data.getSession( )` stand-alone function. For more information, see the getSession( ) stand-alone function on page 248 description. (Available in Progress Data Objects Version 4.3 or later.) Also, see the Notes on page 131 on this class.

Properties

Table 17: progress.data.JSDOSession properties

Member	Brief description (See also the reference entry)
authenticationModel property (JSDOSession class) on page 201	Returns a `string` constant that was passed as an option to the object's class constructor, and specifies the authentication model that the current `JSDOSession` object requires to start a JSDO login session in the web application for which the `JSDOSession` object was also created.
catalogURIs property on page 215	Returns the list of URIs successfully used to load Data Service Catalogs into the current `JSDOSession` or `Session` object.
clientContextId property on page 216	The value of the most recent client context identifier (CCID) that the current `JSDOSession` or `Session` object has found in the `X-CLIENT-CONTEXT-ID` HTTP header of a server response message.
connected property on page 216	Returns a `boolean` that indicates the most recent online status of the current `JSDOSession` or `Session` object when it last determined if the web application it manages was available.
JSDOs property on page 272	Returns an array of JSDOs that use the current `JSDOSession` or `Session` object to communicate with their Data Object Services.
loginHttpStatus property on page 284	Returns the specific HTTP status code returned in the response from the most recent login attempt on the current `JSDOSession` or `Session` object.
loginResult property on page 285	Returns the return value of the `login( )` method, which is the basic result code for the most recent login attempt on the current `JSDOSession` or `Session` object.
name property (JSDOSession class) on page 291	Returns the value of any `name` property to enable page refresh support that was passed to the constructor of the current `JSDOSession` object. **Note:** Applies to Progress Data Objects Version 4.3 or later.

Member	Brief description (See also the reference entry)
onOpenRequest property on page 295	Returns the reference to a user-defined callback function that the `JSDOSession` or `Session` object executes to modify a request object before sending the request object to the server.
pingInterval property on page 303	A `number` that specifies the duration, in milliseconds, between one automatic execution of the current `JSDOSession` or `Session` object's `ping( )` method and the next.
services property on page 331	Returns an array of objects that identifies the Data Object Services that have been loaded for the current `JSDOSession` or `Session` object and its web application.
serviceURI property on page 333	Returns the URI to the web application that has been passed as an option to the class constructor for the current `JSDOSession` object or that has been passed as a parameter to the most recent call to the `login( )` method on the current `Session` object, whether or not the most recent call to `login( )` succeeded.
userName property on page 358	Returns the username passed as a parameter to the most recent call to the `login( )` method on the current `JSDOSession` or `Session` object.

Methods

Table 18: progress.data.JSDOSession class-instance methods

Member	Brief description (See also the reference entry)
addCatalog( ) method (JSDOSession class) on page 159	Loads one or more local or remote Data Service Catalogs into the current `JSDOSession` object.
invalidate( ) method on page 256	Terminates the login session managed by the current `JSDOSession` object and permanently disables the object, rendering it unable to start a new login session. **Note:** Applies to Progress Data Objects Version 4.4.1 or later.
isAuthorized( ) method on page 268	Determines if the current `JSDOSession` object has authorized access to the web application specified by its `serviceURI` property setting. **Note:** Applies to Progress Data Objects Version 4.3 or later.

Member	Brief description (See also the reference entry)
login( ) method (JSDOSession class) on page 275	Starts a JSDO login session in a web application for the current `JSDOSession` object by sending an HTTP request with specified user credentials to the web application URI specified in the object's constructor.
logout( ) method (JSDOSession class) on page 286	Terminates the login session on the web application managed by the current `JSDOSession` object and leaves the object available to start a new login session with a call to its `login( )` method.
ping( ) method (JSDOSession class) on page 297	Determines the online state of the current `JSDOSession` object from its ability to access the web application that it manages, and for an OpenEdge web application, from detecting if its associated application server is running.
subscribe( ) method (JSDOSession class) on page 348	Subscribes a given event callback function to an event of the current `JSDOSession` object.
unsubscribe( ) method (JSDOSession class) on page 355	Unsubscribes a given event callback function from an event of the current `JSDOSession` object.
unsubscribeAll( ) method on page 356	Unsubscribes all event callback functions from a single named event of the current JSDO, `JSDOSession` or `Session` object, or unsubscribes all event callback functions from all events of the current JSDO, `JSDOSession`, or `Session` object.

Events

Table 19: progress.data.JSDOSession events

Member	Brief description (See also the reference entry)
offline event on page 292	Fires when the current `JSDOSession` or `Session` object detects that the device on which it is running has gone offline, or that the web application to which it has been connected is no longer available.
online event on page 294	Fires when the current `JSDOSession` or `Session` object detects that the device on which it is running has gone online after it was previously offline, or that the web application to which it is connected is now available after it was previously unavailable.

Example — Using the JSDOSession class

This example shows how you might create a `JSDOSession` object using the URI and authentication model of a web application (`CustService`), log into the web application,and load the Data Service Catalog for a Data Object Service provided by that web application. It creates this login session for a JSDO that is created and accessed by an instance of the JSDO dialect of the Kendo UI DataSource (`dataSource`), which provides a `Customer` resource from the Data Object Service for access by a Kendo UI Grid:

```
var session;

/* create jsdoSession   */
session = new progress.data.JSDOSession(
        { serviceURI: "http://localhost:8980/CustService",
          authenticationModel: progress.data.Session.AUTH_TYPE_FORM
        };

/* create login screen (using UI defined in HTML file) */
window.loginView = kendo.observable( {
    submit: function() {
        var loginParams = {
                 username: this.username,
                 password: this.password) };

        /* log in (also loads CustService.json if it succeeds) */
        session.login(loginParams
            ).done( // Logged in
            function(session, result, info) {
                session.addCatalog("http://localhost/.../CustService.json"
                    ).done( // Data Service Catalog loaded
                    function(session, result, details){
                        /* success function - create grid and datasource */
                        $("#grid").kendoGrid( {
                            dataSource: {
                                type: "jsdo",
                                transport: {
                                    jsdo: { resourceName: "Customer" }
                                },
                                error: function(e) {
                                    console.log("Error: ", e);
                                }
                            },
                            /* etc., other kendoGrid properties */
                        };

                        /* switch UI to show grid */
                        window.location.href = "#grid";
                    }).fail( // Data Service Catalog not loaded
                    function(session, result, details){
                        alert("Data Service Catalog failed to load");
                }); //JSDOSession addCatalog()

            }).fail( // Not logged in
            function(session, result, info) {
                /* display error message, stay on login screen */
                alert("Login failed");
        }); // JSDOSession login()
    } // observable submit method
} ); // kendo.observable
```

The JSDO automatically finds and uses the `JSDOSession` object where its `Customer` resource is loaded from the Data Service Catalog.

For more information on using the JSDO dialect of the Kendo UI DataSource, see Using the JSDO dialect of the Kendo UI DataSource on page 17.

Notes

- The six web-communicating methods of this class, `invalidate( )`, `isAuthorized( )`, `login( )`, `addCatalog( )`, `logout( )`, and `ping( )`, as well as the stand-alone functions, `progress.data.getSession( )` and `progress.data.invalidateAllSessions( )`, all execute asynchronously and return a reference to a jQuery Promise object as the return value when jQuery Promises are supported in the development environment. A Promise is a deferred object that defines methods which, in turn, register callbacks that execute when results are available from the method that returns the Promise. You provide your own code in each callback to handle the results returned for a given Promise method, much like the code you provide in an event callback.

  The primary Promise methods for use with a `JSDOSession` object include `done( )`, `fail( )`, and `always( )`, which allow you to separately handle the successful, unsuccessful, and all results (respectively) of a `JSDOSession` method execution. Note that for any given `JSDOSession` method, the parameter lists for all Promise method callbacks have the same signature. Note also that the callback for the `always( )` method executes with the same values in its parameter list as the callback for the `done( )` or `fail( )` method, whichever one executes with results.

  For more information on the jQuery implementation of Promises and the additional Promise methods that are available to register callbacks, see the following jQuery topics at the specified web locations:

  - **Category: Deferred Object** — http://api.jquery.com/category/deferred-object/
  - **Promise Object** — http://api.jquery.com/Types/#Promise
  - **.promise** — http://api.jquery.com/promise/

- When you have the credentials from a user login, you have two options to create and initialize a JSDO login session:

  1. As shown in the Example — Using the JSDOSession class on page 130, manually instantiate the `JSDOSession` object for a specified network server (web application) and if successful, invoke its `login( )` method to start the associated JSDO login session and if successful, invoke its `addCatalog( )` method to load one or more Data Service Catalogs and if successful, create one or more JSDO instances to access the resources provided by the initialized Data Object Services.
  2. Make one call to the `progress.data.getSession( )` stand-alone function (the standard mechanism provided by the JSDO framework to start a login session), which automatically instantiates the `JSDOSession` for access to a specified web application, and if successful, invokes its `login( )` method to start the login session, and if successful, invokes its `addCatalog( )` method to load a single Data Service Catalog. If the `getSession( )` invocation is successful, you can then create one or more JSDO instances to access resources provided by the single specified Data Object Service. If you require access to additional Data Object Services, you can also make additional calls to `addCatalog( )` to load the Catalogs. If you require access to additional web applications on this network server (or on other network servers, with CORS support), you can make additional calls to `getSession( )` to create as many login sessions as necessary.

  At any point during an established login session, you can call the `ping( )` method to determine if its `JSDOSession` object is in an online state and able to access both its web application and associated application server.

Caution: Under certain conditions when a server error occurs, either or both the asynchronous `login( )` and `addCatalog( )` methods fail to return a result and notify the app that the error has occurred. However, both of these methods provide an option (which is set by default) that can force these methods to return a result after a specified timeout threshold.

Following is a known set of inclusive conditions that, if all are true, prevent the asynchronous `login( )` and `addCatalog( )` methods from returning an error result to the app:

- The app is running on an iOS device.
- The failure is caused by invalid credentials.
- The authentication model used to authenticate the credentials is HTTP Basic.

The option provided to force these methods to return a result is an `iOSBasicAuthTimeout` property that you can pass to each method in an object parameter. This property specifies a timeout threshold value that when reached forces the method to return an error result to the app in any registered Promise method callback that executes in response to an error result, such as the `fail( )` callback. The session itself enforces a default timeout threshold for these methods of four (4) seconds, which you can change by setting this property to a different value. A value of zero (0) cancels the timeout threshold entirely, which prevents the method from returning under the specified conditions. Note that any non-zero timeout threshold is **only** operative for these methods when **all** of the following conditions are true:

- The JSDO session uses Basic authentication.
- The app is running on an iOS device (iPhone, iPad, or iPod), either as a hybrid app or as a web app in a browser (such as Safari).

Otherwise, no timeout is in effect for either of these methods (and should not be necessary), even if you explicitly set this property to a threshold value in its object argument before invoking the method.

- When your mobile or web app user has signaled their intention to logout, and if necessary for your application, you have three options to terminate one or more current JSDO login sessions, with the following trade-offs:

  1. Invoke the `logout( )` method on each `JSDOSession` object. This both terminates the login session and allows the `JSDOSession` to start another login session by calling `login( )` on it again. Note that this option can pose a security risk and is not recommended, especially for secure logins (other than Anonymous).
  2. Invoke the `invalidate( )` method on each `JSDOSession` object. This both terminates the login session and disables the `JSDOSession` from any further access, preventing any attempt to start another login session with it.
  3. Invoke the `progress.data.invalidateAllSessions( )` stand-alone function. This invokes `invalidate( )` on every `JSDOSession` object that you created using the `progress.data.getSession( )` stand-alone function and is currently managing a login session for the app. This is a convenient way to terminate multiple login sessions unless you have a reason to retain one or more of them for further use, such as Anonymous login sessions you might be using to look-up read-only information. In that case, you must call `invalidate( )` or `logout( )` only for each session you want to terminate.

  Note: If you did not use `getSession( )` to create a `JSDOSession` object, but instantiated the class manually, you can only log out its session or invalidate the object permanently by calling `logout( )` or `invalidate( )` on the `JSDOSession` instance manually.

- The behavior of a login session using this class depends on the authentication model of the web server and how its resources are protected and accessed. For more information, see the description of the `login( )` method.
- If you have special requirements for sending `JSDOSession` requests to the web server, you can modify the XMLHttpRequest object that is sent by the `JSDOSession` object. To do so, assign a callback function as the value of `onOpenRequest` property.
- You can enable page refresh support accessed by a client web app (available in Progress Data Objects Version 4.3 or later) when you create a `JSDOSession` instance using either the class constructor or the `progress.data.getSession( )` stand-alone function. If you enable page refresh support, when the user initiates a browser page refresh on the web app page and you instantiate the corresponding

JSDOSession object with the same name property value, the following JSDOSession properties will have the same values as they did for the corresponding JSDOSession instance before the page refresh was initiated (with one exception):

- loginResult
- clientContextId
- connected
- loginHttpStatus
- userName
- pingInterval
- authenticationModel (passed to the constructor anyway)
- serviceURI (passed to the constructor anyway)

The one exception is if the values of the authenticationModel and serviceURI properties passed to the JSDOSession constructor are different from existing saved values. In this case, the saved data is ignored and cleared out, and a subsequent invocation of the isAuthorized( ) method returns progress.data.Session.LOGIN_AUTHENTICATION_REQUIRED as the *result* parameter value to any callback registered for the Promise that is returned by the method.

See also

addCatalog( ) method (JSDOSession class) on page 159, getSession( ) stand-alone function on page 248, invalidate( ) method on page 256, invalidateAllSessions( ) stand-alone function on page 258, isAuthorized( ) method on page 268, login( ) method (JSDOSession class) on page 275, logout( ) method (JSDOSession class) on page 286, onOpenRequest property on page 295, ping( ) method (JSDOSession class) on page 297, progress.data.JSDO class on page 110

progress.data.JSRecord class

The progress.data.JSRecord is a JavaScript class that represents a record instance for any table stored in the JSDO memory of an associated progress.data.JSDO class instance (JSDO). The constructor for progress.data.JSRecord is protected. JSRecord instances are created internally by JSDO methods.

Properties

Table 20: progress.data.JSRecord class properties

Member	Brief description (See also the reference entry)
data property on page 217	The data (field) and state values for a record associated with a JSRecord object.

Methods

Table 21: progress.data.JSRecord class methods

Member	Brief description (See also the reference entry)
acceptRowChanges( ) method on page 156	Accepts changes to the data in JSDO memory for a specified record object.
assign( ) method (JSDO class) on page 198 **Alias:** `update( )`	Updates field values for the specified `JSRecord` object in JSDO memory.
getErrorString( ) method on page 239	Returns any before-image error string in the data of a record object referenced in JSDO memory that was set as the result of a resource Create, Update, Delete, or Submit operation.
getId( ) method on page 242	Returns the unique internal ID for the record object referenced in JSDO memory.
remove( ) method on page 311	Deletes the specified table record referenced in JSDO memory.
rejectRowChanges( ) method on page 308	Rejects changes to the data in JSDO memory for a specified record object.

Events

This class has no events.

Example

The following example assumes that a JSDO is referenced by the `jsdo` variable, and that a `UIHelper` instance associated with that JSDO is referenced by the `uihelper` variable. The example creates a new record object and displays it, along with a message with credit information using properties of the record object:

```
function addRecord() {
  var jsrecord = jsdo.add( {Balance: 10000, State: 'MA'} );
  uihelper.display( );
  alert( 'Record ID: ' + jsrecord.getId( ) + ' CreditLimit: ' +
         jsrecord.data.CreditLimit );
}
```

Notes

Using the `add( )`, `find( )`, `findById( )`, or `foreach( )` method, or the `record` property, on a given JSDO and table reference, a `JSRecord` instance returns a working record for the table referenced in JSDO memory. You can then use properties and methods of the `JSRecord` to update, delete, or display the specified record from the JSDO.

See also

add( ) method on page 158, find( ) method on page 227, findById( ) method on page 229, foreach( ) method on page 231, progress.data.JSDO class on page 110, record property on page 305, table reference property (JSDO class) on page 351

progress.data.PluginManager class

Note: Applies to Progress Data Objects Version 4.3 or later.

The `progress.data.PluginManager` is a JavaScript class that provides class-level methods to create and manage custom JSDO plugins. A *JSDO plugin* is an object with functions that execute for any supported Data Object operation that you invoke on a client JSDO instance. You can define these functions for a given plugin to customize the default behavior of the associated operation.

To use a JSDO plugin for a Data Object operation you must complete the following tasks:

- For an OpenEdge Data Object, annotate the ABL Business Entity that implements the operation to identify the name of the JSDO plugin with the operation in the Data Service Catalog. Rollbase internal Data Objects provide server support for identifying a built-in `"JFP"` JSDO plugin with the operation.
- Use the (class-level) methods of this class to add (*register*) a new JSDO plugin or modify an existing plugin for use by any client JSDO instance that invokes the operation on the server.

For more information on completing these tasks, see the descriptions of the methods of this class.

Currently, the only Data Object operation that supports JSDO plugins is the Read operation (`fill( )` method), and the only supported plugin type is a *MappingType* plugin that allows you to customize both the operation request and the response returned from executing the operation on the server.

The JSDO also defines a single built-in MappingType plugin with the reserved name, `"JFP"`. This built-in plugin supports client handling of requests for Read operations that recognize JSON Filter Pattern (JFP) input. For more information on JFP input and this built-in MappingType plugin, see the description of the fill( ) method on page 220. You can customize the default request handling in this built-in `"JFP"` plugin, as well as add a custom response handler to it. You can also register your own plugins that are based on this built-in plugin (or that are based on any custom plugin previously registered in the document context).

This class has no constructor, but provides class-level methods to manage plugin registration.

Methods

Table 22: progress.data.PluginManager class-level methods

Member	Brief description (See also the reference entry)
addPlugin( ) method on page 171	Adds (registers) a JSDO plugin object with a specified name in the current document context that can be used with a supported Data Object operation invoked on a JSDO instance.
getPlugin( ) method on page 244	Returns a reference to the registered JSDO plugin object with the specified name.

Note

For more information on registering a new custom JSDO plugin or on customizing a previously registered JSDO plugin, as well as on specifying a JSDO plugin for use by a Data Object operation on the server, see the description of the addPlugin( ) method on page 171.

See also

fill( ) method on page 220

progress.data.Session class

Note: Deprecated in Progress Data Objects Version 4.4 and later. Please use the progress.data.JSDOSession class on page 124 instead.

The `progress.data.Session` is a JavaScript class that provides methods, properties, and events to create and manage a JSDO login session. A *JSDO login session* includes a single end point (web application) and a single authentication model (Anonymous, HTTP Basic, or HTTP Form), and manages user access to Progress Data Object resources using instances of the `progress.data.JSDO` class (JSDO). This includes loading the definitions for the Data Object resources that JSDOs can access and starting and managing JSDO login sessions on web servers that provide access to these resources.

This `Session` class supports similar features to the `progress.data.JSDOSession` class, except that `Session` methods for loading Data Service Object resource definitions and managing login sessions can either run synchronously or fire asynchronous events to allow results to be handled in separately registered callbacks, whereas the equivalent `progress.data.JSDOSession` methods run asynchronously **only** and return jQuery Promises to handle the results in callbacks registered using Promise methods. In addition, the information required to instantiate and start JSDO login sessions with `Session` objects is handled somewhat differently than with `JSDOSession` objects.

Note: If you are creating JSDO login sessions to work with the JSDO dialect of the Kendo UI DataSource, Progress recommends using the `progress.data.JSDOSession` class to manage login sessions for JSDOs. Note that to use `JSDOSession` objects, your development environment must also support jQuery Promises (or the exact equivalent), similar to the Telerik Platform. For more information, see the description of the progress.data.JSDOSession class on page 124. For more information on the JSDO dialect of the Kendo UI DataSource, see Using the JSDO dialect of the Kendo UI DataSource on page 17.

Like a `JSDOSession` object, a `Session` object manages user authentication and session identification information in HTTP/S messages sent between JSDOs running in a mobile app and Data Object Services running on a web server, each of which provide access to a set of Data Object resources. The authentication information includes a username and password (*user credentials*), if necessary, to authenticate a JSDO login session in a web application, which provides a REST transport between a set of Data Object Services and any client mobile app that accesses them. The session identification information includes the URI for the web application and might include a session ID that identifies the JSDO login session and helps to manage interactions between the client and the web application.

To start a JSDO login session for a `Session` object to manage, first instantiate the `Session` object and ensure that you have set the object's `authenticationModel` property to the correct authentication model for the session as required by the web application's security configuration. Then invoke the `login( )` method on the `Session` object, passing as parameters the web application URI, any required user credentials to authenticate `Session` access to the web application according to its authentication model, and an optional specified web application resource (such as a static HTML page) used to authenticate access. Once started, a login session for a web application supports all Data Object Services that the application provides, each of which can define one or more resources for access by JSDOs.

Each Data Object Service provides a separate JSON file (Data Service Catalog) that defines the schema for its set of resources and the operations to communicate between these resources and the JSDO instances that access them from a mobile app. To create a JSDO instance for a Data Object resource, a `Session` object must first load the Catalog that defines the resource using its `addCatalog( )` method. This method can load this Catalog from the web application, from some other web location, or from a location on the client where the mobile app runs, and the method can also accept credentials to access a Data Service Catalog that is protected separately from the web application itself.

Once a Data Service Catalog is loaded into a `Session` object, you can instantiate a JSDO to access a particular resource defined in that Catalog. Once the `Session` is logged into its web application, the JSDO can then access its resource, relying on any authentication information for the login session (if necessary) to authorize this access.

Multiple JSDOs can thus rely on a single `Session` object to manage session access to all Data Object Services and their resources defined by the same web application.

Constructor

Instantiates a `Session` object that you can use to start a JSDO login session for a web application and load the Data Service Catalog for each supported Data Object Service whose resources are accessed using JSDOs.

Syntax

```
progress.data.Session ( )
```

Properties

Table 23: progress.data.Session properties

Member	Brief description (See also the reference entry)
authenticationModel property (Session class) on page 201	A `string` constant that you can set to specify the authentication model that a given web application requires for a mobile app to start a JSDO login session for this `Session` object.
catalogURIs property on page 215	Returns the list of URIs successfully used to load Data Service Catalogs into the current `JSDOSession` or `Session` object.
clientContextId property on page 216	The value of the most recent client context identifier (CCID) that the current `JSDOSession` or `Session` object has found in the `X-CLIENT-CONTEXT-ID` HTTP header of a server response message.

Member	Brief description (See also the reference entry)
connected property on page 216	Returns a `boolean` that indicates the most recent online status of the current `JSDOSession` or `Session` object when it last determined if the web application it manages was available.
JSDOs property on page 272	Returns an array of JSDOs that use the current `JSDOSession` or `Session` object to communicate with their Data Object Services.
lastSessionXHR property on page 274	Returns an object reference to the XMLHttpRequest object (XHR) that was most recently used by the `progress.data.Session` object to execute a `Session` object method.
loginHttpStatus property on page 284	Returns the specific HTTP status code returned in the response from the most recent login attempt on the current `JSDOSession` or `Session` object.
loginResult property on page 285	Returns the return value of the `login( )` method, which is the basic result code for the most recent login attempt on the current `JSDOSession` or `Session` object.
loginTarget property on page 285	Returns the string appended to the web application URI passed to the `login( )` method to form the URI of an application resource against which the user has been authenticated for the current login session.
onOpenRequest property on page 295	Returns the reference to a user-defined callback function that the `JSDOSession` or `Session` object executes to modify a request object before sending the request object to the server.
pingInterval property on page 303	A `number` that specifies the duration, in milliseconds, between one automatic execution of the current `JSDOSession` or `Session` object's `ping( )` method and the next.
services property on page 331	Returns an array of objects that identifies the Data Object Services that have been loaded for the current `JSDOSession` or `Session` object and its web application.

Member	Brief description (See also the reference entry)
serviceURI property on page 333	Returns the URI to the web application that has been passed as an option to the class constructor for the current `JSDOSession` object or that has been passed as a parameter to the most recent call to the `login( )` method on the current `Session` object, whether or not the most recent call to `login( )` succeeded.
userName property on page 358	Returns the username passed as a parameter to the most recent call to the `login( )` method on the current `JSDOSession` or `Session` object.

Methods

Table 24: progress.data.Session class-instance methods

Member	Brief description (See also the reference entry)
addCatalog( ) method (Session class) on page 164	Loads a local or remote Data Service Catalog into the current `Session` object.
login( ) method (Session class) on page 280	Starts a JSDO login session on the current `Session` object by sending an HTTP request with specified user credentials to the URI of a specified web application.
logout( ) method (Session class) on page 288	Terminates the login session on the web application managed by the current `Session` object, and reinitializes most of the state information maintained by the object.
ping( ) method (Session class) on page 300	Determines the online state of the current `Session` object from its ability to access the web application that it manages, and for an OpenEdge web application, from detecting if its associated application server is running.
subscribe( ) method (Session class) on page 349	Subscribes a given event callback function to an event of the current `Session` object.
unsubscribe( ) method (Session class) on page 356	Unsubscribes a given event callback function from an event of the current `Session` object.
unsubscribeAll( ) method on page 356	Unsubscribes all event callback functions from a single named event of the current JSDO, `JSDOSession` or `Session` object, or unsubscribes all event callback functions from all events of the current JSDO, `JSDOSession`, or `Session` object.

Events

Table 25: progress.data.Session events

Member	Brief description (See also the reference entry)
afterAddCatalog event on page 179	Fires when the `addCatalog( )` method on the current `Session` object completes execution after it was called asynchronously.
afterLogin event on page 188	Fires when the `login( )` method on the current `Session` object completes execution after it was called asynchronously.
afterLogout event on page 190	Fires when the `logout( )` method on the current `Session` object completes execution after it was called asynchronously.
offline event on page 292	Fires when the current `JSDOSession` or `Session` object detects that the device on which it is running has gone offline, or that the web application to which it has been connected is no longer available.
online event on page 294	Fires when the current `JSDOSession` or `Session` object detects that the device on which it is running has gone online after it was previously offline, or that the web application to which it is connected is now available after it was previously unavailable.

Example — Using the Session class

This is an example of how you might create a `Session` object and use the URI to a web application to log into the application, load the Data Service Catalog for a Data Object Service provided by that application, and create a JSDO for a `Customer` resource defined by that Service in the Catalog:

```
// create Session
pdsession = new progress.data.Session();

// log in, i.e., authenticate to the web application
pdsession.login('https://BestSports.com:443/SportsApp', username, password);

// load Catalog for a service that's part of the web application
pdsession.addCatalog('https://BestSports.com:443/SportsApp/static/OrderEntrySvc.json');

// create JSDO
customers = new progress.data.JSDO( { name: 'Customer' } );

/* etc. - additional code to fill and use the JSDO */
```

The JSDO automatically finds and uses the `Session` object on which a Catalog that defines the `Customer` resource is loaded.

Notes

- Use an instance of this class to call the `login( )` method to start a JSDO login session, call the `addCatalog( )` method to load one or more Data Service Catalogs for the session, and possibly call the

`logout( )` method to terminate the session. To use the same `Session` instance to start a new login session, you must call the `logout( )` method first.

Caution: To help ensure that HTTP Forms access to web applications works in certain web browsers, such as Firefox, when the web application is configured for Cross-Origin Resource Sharing (CORS), always call the `login( )`, `addCatalog( )`, and `logout( )` methods asynchronously.

Caution: Under certain conditions when a server error occurs, either or both the `login( )` and `addCatalog( )` methods fail to return a result and notify the app that the error has occurred. However, both of these methods provide an option (which is set by default) that can force these methods to return a result after a specified timeout threshold.

Following is a known set of inclusive conditions that, if all are true, prevent the `login( )` and `addCatalog( )` methods from returning an error result to the app:

- The app is running on an iOS device.
- The failure is caused by invalid credentials.
- The authentication model used to authenticate the credentials is HTTP Basic.
- The method is executed asynchronously.

The option provided to force these methods to return a result is an `iOSBasicAuthTimeout` property that you can pass to each method in an object parameter. This property specifies a timeout threshold value that when reached forces the method to return an error result to the app in an any registered `afterLogin` or `afterAddCatalog` event callback. The session itself enforces a default timeout threshold for these methods of four (4) seconds, which you can change by setting this property to a different value. A value of zero (0) cancels the timeout threshold entirely, which prevents the method from returning under the specified conditions. Note that any non-zero timeout threshold is **only** operative for these methods when **all** of the following conditions are true:

- The app is running on an iOS device (iPhone, iPad, or iPod), either as a hybrid app or as a web app in a browser (such as Safari).
- The JSDO session uses Basic authentication.
- The `login( )` or `addCatalog( )` method is invoked asynchronously.

Otherwise, no timeout is in effect for either of these methods (and should not be necessary), even if you explicitly set this property to a threshold value in its object argument before invoking the method.

- The behavior of a login session using this class depends on the authentication model of the web server and how its resources are protected and accessed. For more information, see the description of the `login( )` method.
- If you have special requirements for sending Data Object Service requests to the web server, you can modify the XMLHttpRequest object that is sent by the `Session` object. To do so, assign a callback function as the value of `Session.onOpenRequest`.

See also

addCatalog( ) method (Session class) on page 164, login( ) method (Session class) on page 280, logout( ) method (Session class) on page 288, progress.data.JSDO class on page 110

progress.ui.UIHelper class

The `progress.ui.UIHelper` class is a JSDO class that provides methods for managing the user interface of a mobile app that accesses a JSDO. This class is intended for use by:

- Developers using JavaScript libraries, such as jQuery Mobile, to build user interfaces by creating HTML-based lists and fields.
- Developers using tools other than the Telerik Platform. The Telerik Platform and its Kendo UI DataSource eliminate the need for `UIHelper` functionality.

Each instance of `UIHelper` supports the display of data for a specific JSDO, and typically controls the format and content of a list view (showing items representing table records) and a detail page (showing a form with input fields for the list item clicked by the user).

Constructor

Instantiates a `UIHelper` object for use in managing the UI for a specified JSDO.

Syntax

```
progress.ui.UIHelper( JSDO-object )
```

JSDO-object

An object reference to the JSDO associated with the `UIHelper` instance.

Properties

Table 26: progress.ui.UIHelper properties

Member	Brief description (See also the reference entry)
table reference property (UIHelper class) on page 353	An object reference property on a `UIHelper` instance that corresponds to a table of the JSDO with which the `UIHelper` instance is associated.

Methods

Table 27: progress.ui.UIHelper class-level methods

Member	Brief description (See also the reference entry)
setFieldTemplate( ) method on page 334	A class method that specifies a new default field format for all detail forms created with `UIHelper` instances in the current JavaScript session.
setItemTemplate( ) method on page 335	A class method that specifies a new default item format for all list views created with `UIHelper` instances in the current JavaScript session.

Table 28: progress.ui.UIHelper table-reference methods

Member	Brief description (See also the reference entry)
addItem( ) method on page 168	Adds an item to a list view based on the working record of a specified table reference.
assign( ) method (UIHelper class) on page 199	Updates a specified working record in JSDO memory with the values currently displayed on the detail page form.
clearItems( ) method on page 215	Clears the items from a list view associated with a single table reference.
display( ) method on page 219	Copies the field values of a given record to corresponding fields in the current HTML document for display in the form on the detail page.
getFormFields( ) method on page 240	Reads the schema of the referenced table and returns a `string` containing the HTML text of field definitions created by processing a `UIHelper` form field template against fields that you specify.
getFormRecord( ) method on page 241	Returns a `JSRecord` object for the record shown in the form on the detail page.
getListViewRecord( ) method on page 243	Returns a `JSRecord` object for a specified item in a list view.
setDetailPage( ) method on page 333	Specifies the HTML page that contains the form in which item details are displayed.
setListView( ) method on page 336	Defines a list view for a referenced JSDO table.
showListView( ) method on page 343	Displays the referenced table's list view on the mobile app device or browser page.

Events

This class has no events.

Example

The sample mobile app shown below is followed by the code for its index page (`index.html`) and client logic (`customers.js`):

Figure 2: Sample mobile screen using the `progress.ui.UIHelper` class

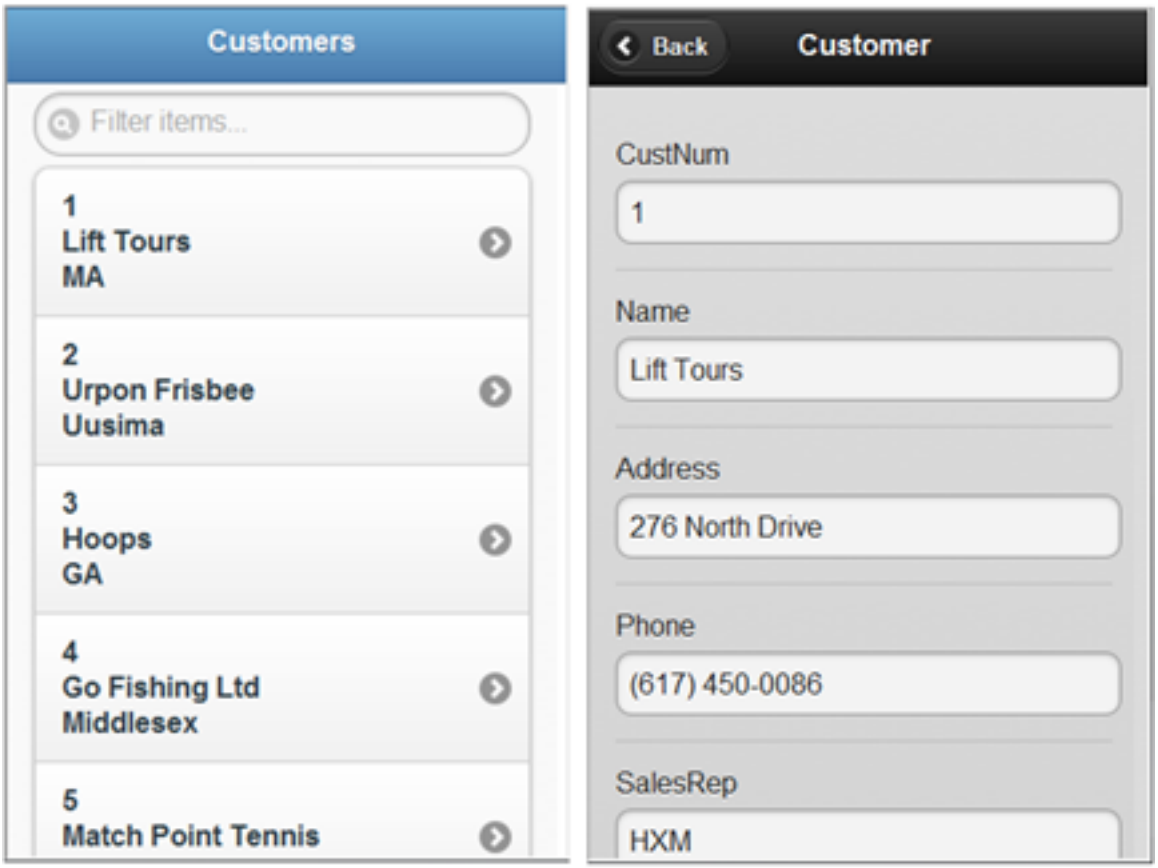


Table 29: Example — index.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1" />
        <title>Customers</title>
        <link rel="stylesheet"
href="http://code.jquery.com/mobile/1.1.0/jquery.mobile-1.1.0.min.css" />
        <style>
            /* App custom styles */
        </style>
        <script
src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
        <script
src="http://code.jquery.com/mobile/1.1.0/jquery.mobile-1.1.0.min.js"></script>
        <script src="../progress.session.js"></script>
        <script src="../progress.js"></script>
        <script src="customers.js"></script>
    </head>
    <body>
        <div data-role="page" id="custlist">
            <div data-theme="b" data-role="header">
                <h3>Customers</h3>
            </div>
            <div data-role="content">
                <ul id="listview" data-role="listview" data-divider-theme="b"
data-inset="true" data-filter="true">
                </ul>
            </div>
        </div>
        <div data-role="page" id="custdetail" data-add-back-btn="true" data-theme="b">
        <div data-role="header"><h1>Customer</h1></div>
        <div data-role="content">
                <form action="" id="customerform">
                </form>
        </div>
        </div>
    </body>
</html>
```

Table 30: Example — customers.js

```
var customers;
var uihelper;
var forminitialized = false;

$(document).ready(function() {
    var session = new progress.data.Session();

        session.login("http://localhost:8980/SportsApp", "", "");
        session.addCatalog(
'http://localhost:8980/SportsApp/static/CustomerOrderSvc.json' );

        customers = new progress.data.JSDO({ name: 'CustomerOrder' });
        customers.subscribe('AfterFill', onAfterFillCustomers, this);

        uihelper = new progress.ui.UIHelper({ jsdo: customers });
        uihelper.eCustomer.setDetailPage({
            name: 'custdetail'
        });

        uihelper.eCustomer.setListView({
            name:'listview',
            format: '{CustNum}<br>{Name}<br>{State}',
            autoLink: true
        });

        $('#customerform').html(
            uihelper.eCustomer.getFormFields()
               + '<input type="button" value="Add" id="btnAdd" />'
               + '<input type="button" value="Save" id="btnSave" />'
               + '<input type="button" value="Delete" id="btnDelete"
/>'
        );

    customers.fill();
});
function onAfterFillCustomers() {
    uihelper.eCustomer.clearItems();
    customers.eCustomer.foreach(function(customer) {
        uihelper.eCustomer.addItem();
    });
    uihelper.eCustomer.showListView();
}
```

See also

progress.data.JSDO class on page 110

request object

An object containing data and status information returned from a call to one of the methods of an associated `progress.data.JSDO` (JSDO) instance that executes a CRUD, Submit, or Invoke operation on a resource of a Data Object Service. This `request` object is returned by the associated JSDO method call: `fill( )`, `saveChanges( )`, or a given invocation method.

In the case of an asynchronous call (any resource operation that you execute asynchronously), the `request` object is passed as a parameter to any event callback functions that you subscribe to associated JSDO events, or to any callback functions that you register using a returned jQuery Promise object. For Invoke operations that you execute synchronously, the object is available as the return value of the corresponding JSDO invocation method. The object is also passed as a parameter to any event callback functions that you subscribe to the `online` and `offline` events of the `JSDOSession` or `Session` object that manages Data Object Services for the JSDO.

Properties

Table 31: Request object properties

Member	Brief description (See also the reference entry)
async property on page 200	A `boolean` that indicates, if set to `true`, that the Data Object resource operation was executed asynchronously in the mobile app.
batch property on page 205	A reference to an object with a property named `operations`, which is an array containing the request objects for each of the one or more record-change operations on a resource performed in response to calling the JSDO `saveChanges( )` method without using Submit (either with an empty parameter list or with the single parameter value of `false`).
fnName property on page 230	For an Invoke operation, the name of the custom JSDO invocation method that executed the operation.
jsdo property on page 272	An object reference to the JSDO that performed the operation returning the request object.
jsrecord property on page 272	A reference to the record object that was created, updated, or deleted by the current record-change operation.
jsrecords property on page 273	An array of references to record objects that are created, updated, or deleted on the server for a Submit operation (invoking `saveChanges(true)`) on a Data Object resource that supports before-imaging.
objParam property on page 291	A reference to the object, if any, that was passed as an input parameter to the JSDO method that has returned the current request object.
response property on page 312	Returns an object or `string` containing data and status information from an operation invoked on a Data Object resource.

Member	Brief description (See also the reference entry)
success property on page 350	A `boolean` that when set to `true` indicates that the Data Object resource operation was successfully executed.
xhr property on page 358	A reference to the XMLHttpRequest object used to make an operation request on a resource of a Data Object Service.

Methods

This object has no methods.

Events

This object has no events.

See also

fill( ) method on page 220, invocation method on page 262, progress.data.JSDO class on page 110, progress.data.JSDOSession class on page 124, progress.data.Session class on page 136, saveChanges( ) method on page 314

7

JSDO properties, methods, and events reference

This reference describes the properties, methods, and events of the JavaScript classes and objects described in JSDO class and object reference on page 109. Where a given member can be accessed on different object types, the member description indicates this in an **Applies to** list.

If multiple object types support access to the same member, the member description indicates any differences in behavior. For example, for methods, the method syntax shows how to call the method on each object type, and the description indicates when to call the method, and to what effect, for each object type where you can call it.

If the member is significantly different in form and function when used with different object types, there is a separate description of the member for each object type, with its title qualified with the object type, for example, one description for authenticationModel property (JSDOSession class) on page 201 and one for authenticationModel property (Session class) on page 201.

For details, see the following topics:

- acceptChanges( ) method
- acceptRowChanges( ) method
- add( ) method
- addCatalog( ) method (JSDOSession class)
- addCatalog( ) method (Session class)
- addItem( ) method
- addLocalRecords( ) method

- addPlugin( ) method
- addRecords( ) method
- afterAddCatalog event
- afterCreate event
- afterDelete event
- afterFill event
- afterInvoke event
- afterLogin event
- afterLogout event
- afterSaveChanges event
- afterUpdate event
- assign( ) method (JSDO class)
- assign( ) method (UIHelper class)
- async property
- authenticationModel property (JSDOSession class)
- authenticationModel property (Session class)
- autoApplyChanges property
- autoSort property
- batch property
- beforeCreate event
- beforeDelete event
- beforeFill event
- beforeInvoke event
- beforeSaveChanges event
- beforeUpdate event
- caseSensitive property
- catalogURIs property
- clearItems( ) method
- clientContextId property
- connected property
- data property
- deleteLocal( ) method
- display( ) method

- fill( ) method
- find( ) method
- findById( ) method
- fnName property
- foreach( ) method
- getData( ) method
- getErrors( ) method
- getErrorString( ) method
- getFormFields( ) method
- getFormRecord( ) method
- getId( ) method
- getListViewRecord( ) method
- getPlugin( ) method
- getProperties( ) method
- getProperty( ) method
- getSchema( ) method
- getSession( ) stand-alone function
- hasChanges( ) method
- hasData( ) method
- invalidate( ) method
- invalidateAllSessions( ) stand-alone function
- invocation method
- invoke( ) method
- isAuthorized( ) method
- jsdo property
- JSDOs property
- jsrecord property
- jsrecords property
- lastSessionXHR property
- login( ) method (JSDOSession class)
- login( ) method (Session class)
- loginHttpStatus property
- loginResult property

- loginTarget property
- logout( ) method (JSDOSession class)
- logout( ) method (Session class)
- name property (JSDO class)
- name property (JSDOSession class)
- objParam property
- offline event
- online event
- onOpenRequest property
- ping( ) method (JSDOSession class)
- ping( ) method (Session class)
- pingInterval property
- readLocal( ) method
- record property
- rejectChanges( ) method
- rejectRowChanges( ) method
- remove( ) method
- response property
- saveChanges( ) method
- saveLocal( ) method
- services property
- serviceURI property
- setDetailPage( ) method
- setFieldTemplate( ) method
- setItemTemplate( ) method
- setListView( ) method
- setProperties( ) method
- setProperty( ) method
- setSortFields( ) method
- setSortFn( ) method
- showListView( ) method
- sort( ) method
- subscribe( ) method (JSDO class)

- subscribe( ) method (JSDOSession class)
- subscribe( ) method (Session class)
- success property
- table reference property (JSDO class)
- table reference property (UIHelper class)
- unsubscribe( ) method (JSDO class)
- unsubscribe( ) method (JSDOSession class)
- unsubscribe( ) method (Session class)
- unsubscribeAll( ) method
- useRelationships property
- userName property
- xhr property

acceptChanges( ) method

Accepts changes to the data in JSDO memory for the specified table reference or for all table references of the specified JSDO.

If the method succeeds, it returns `true`. Otherwise, it returns `false`.

Note: This method is most useful when the JSDO `autoApplyChanges` property is set to `false`. In this case, you typically invoke this method **after** calling the `saveChanges( )` method in order to accept a series of changes after they have been successfully applied to the server. If the `autoApplyChanges` property is `true`, the JSDO automatically accepts or rejects changes for the specified table reference, or for all table references of the specified JSDO, based on the success of the corresponding record-change operations on the server resource.

Note: Accepting all pending changes in JSDO memory—or even pending changes for a single table reference—because none raised an error from the server might be too broad an action for your application. If so, consider using `acceptRowChanges( )` to accept changes a single table record at a time. For more information, see the description of `acceptRowChanges( )` method.

Return type: `boolean`

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Working record: The working record is set depending on the changes accepted.

Syntax

```
jsdo-ref.acceptChanges ( )
jsdo-ref.table-ref.acceptChanges ( )
```

jsdo-ref

A reference to the JSDO. If you call the method on *jsdo-ref*, the method accepts changes for all table references in the JSDO.

table-ref

A table reference on the JSDO. If you call the method on *table-ref*, the method accepts changes for the specified table reference.

When you accept changes on a table reference, this method makes the record objects in the specified table reflect all pending changes to them in JSDO memory. When you accept changes on the JSDO reference, the method makes the record objects in all tables of the JSDO reflect all pending changes to them in JSDO memory. As the specified changes are accepted, the method also empties any associated before-image data, clears all associated error message settings, and removes the associated record change indications from JSDO memory.

Note: Regardless if you call `acceptChanges( )`, any error message settings that result from Data Object resource operations invoked by the most recent execution of the `fill( )` or `saveChanges( )` method remain available for return by the `getErrors( )` method until the next execution of either `fill( )` or `saveChanges( )`. For more information, see the getErrors( ) method on page 233 description.

This method is especially useful for a resource that supports before-imaging (such as an OpenEdge ProDataSet) and handles the results of all record changes that are sent using a Submit operation as part of a single server transaction that successfully commits all the changes. When the Submit operation returns successfully, calling this method ensures that JSDO memory is synchronized with all the record changes that were committed in the server transaction.

Note: After this method accepts changes, and if you have set up automatic sorting using the `autoSort` property, all the record objects for affected table references are sorted accordingly. If the sorting is done using sort fields, any `string` fields are compared according to the value of the `caseSensitive` property.

Caution: If you have pending JSDO changes that you need to apply to the server, be sure **not** to invoke this method **before** you invoke the `saveChanges( )` method to successfully apply these changes to the server. Otherwise, the affected client data will be inconsistent with the corresponding data on the server.

Example

The following code fragment shows a JSDO created so it **does not** automatically accept or reject changes to data in JSDO memory after a call to the `saveChanges( )` method. Instead, it subscribes a handler for the JSDO `afterSaveChanges` event to determine if all changes to the `eCustomer` table in JSDO memory should be accepted or rejected based on the success of all resource Create, Update, and Delete operations on the server. To change the data for a record, a jQuery event is also defined on an update button to update the corresponding `eCustomer` record in JSDO memory with the current field values entered in a customer detail form (`#custdetail`):

```
dataSet = new progress.data.JSDO( { name: 'dsCustomerOrder',
                                    autoApplyChanges : false } );
dataSet.eCustomer.subscribe('afterSaveChanges', onAfterSaveCustomers, this);

$('#btnUpdate').bind('click', function(event) {
  var jsrecord = dataSet.eCustomer.findById($('#custdetail #id').val());
  if (jsrecord) {jsrecord.assign();};
});

// Similar controls might be defined to delete and create eCustomer records...

$('#btnSave').bind('click', function(event) {
  dataSet.saveChanges();
});

function onAfterSaveCustomers(jsdo, success, request) {
  if (success)
  {
    jsdo.eCustomer.acceptChanges();
    // Additional actions associated with accepting the pending changes...
  }
  else
  {
    jsdo.eCustomer.rejectChanges();
    // Additional actions associated with rejecting the pending changes...
  }
}
```

When the update button is clicked, the event handler uses the `findById( )` method to find the original record (`jsrecord`) with the matching internal record ID (`#id`) and invokes the `assign( )` method on `jsrecord` with an empty parameter list to update its fields in `eCustomer` with any new values entered into the form. You might define similar events and controls to delete `eCustomer` records and add new `eCustomer` records.

A jQuery event also defines a save button that when clicked invokes the `saveChanges( )` method to apply all pending changes in JSDO memory to the server. After the method completes, and all results have been returned to the client from the server, the JSDO `afterSaveChanges` event fires, and if all resource operations on the server were successful, the handler calls `acceptChanges( )` to accept the pending changes to `eCustomer` in JSDO memory. For more information on how this same example determines when and how to reject changes, see the description of the `rejectChanges( )` method.

Note: This example shows the default invocation of `saveChanges( )`, which invokes each resource record-change operation, one record at a time, across the network. For a resource that supports before-imaging (such as an OpenEdge ProDataSet), you can also have `saveChanges( )` send all pending record change operations across the network in a single Submit operation. For more information and an example, see the description of the `saveChanges( )` method.

See also:

acceptRowChanges( ) method on page 156, autoApplyChanges property on page 202, autoSort property on page 203, caseSensitive property on page 213, rejectChanges( ) method on page 305, saveChanges( ) method on page 314

acceptRowChanges( ) method

Accepts changes to the data in JSDO memory for a specified record object.

This can be the working record of a table reference or the record specified by a `JSRecord` object reference. If the method succeeds, it returns `true`. Otherwise, it returns `false`.

Note: This method is most useful when the JSDO `autoApplyChanges` property is set to `false`. In this case, you might invoke this method in the callback for the corresponding JSDO event fired in response to a successful record-change operation on the server resource that was invoked by executing the `saveChanges( )` method. If the `autoApplyChanges` property is `true`, the JSDO automatically accepts or rejects changes to the record object based on the success of the corresponding resource operation on the server.

Return type: `boolean`

Applies to: progress.data.JSRecord class on page 133, progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Working record: The working record is set depending on the changes accepted.

Syntax

```
jsrecord-ref.acceptRowChanges ( )
jsdo-ref.acceptRowChanges ( )
jsdo-ref.table-ref.acceptRowChanges ( )
```

jsrecord-ref

A reference to a `JSRecord` object for a table reference in JSDO memory.

You can obtain a `JSRecord` object by:

- Invoking a JSDO method that returns record objects from a JSDO table reference (`find( )`, `findById( )`, or `foreach( )`)
- Accessing the `record` property on a JSDO table reference that already has a working record.
- Accessing the *`record`* parameter passed to the callback of a JSDO `afterCreate`, `afterDelete`, or `afterDelete` event.

jsdo-ref

A reference to the JSDO. You can call the method on *`jsdo-ref`* if the JSDO has only a single table reference, and that table reference has a working record.

table-ref

A table reference on the JSDO that has a working record.

When you accept changes on a specified record object, this method makes the record reflect all pending changes to it in JSDO memory. As the specified changes are accepted, the method also empties any associated before-image data, clears any associated error message setting, and removes the associated pending change indications from JSDO memory.

Note: Regardless if you call `acceptRowChanges( )`, any error message settings that result from Data Object resource operations invoked by the most recent execution of the `fill( )` or `saveChanges( )` method remain available for return by the `getErrors( )` method until the next execution of either `fill( )` or `saveChanges( )`. For more information, see the getErrors( ) method on page 233 description.

Note: After this method accepts changes on a record, and if you have set up automatic sorting using the `autoSort` property, all the record objects for the affected table reference are sorted accordingly. If the sorting is done using sort fields, any `string` fields are compared according to the value of the `caseSensitive` property.

Caution: If you have pending JSDO changes that you need to apply to the server, be sure **not** to invoke this method **before** you invoke the `saveChanges( )` method. Otherwise, the affected client data will be inconsistent with the corresponding data on the server.

Example

The following code fragment shows a JSDO created so it **does not** automatically accept or reject changes to data in JSDO memory after a call to the `saveChanges( )` method. Instead, it subscribes a single handler for each of the `afterDelete`, `afterCreate`, and `afterUpdate`, events to determine if changes to any `eCustomer` table record in JSDO memory should be accepted or rejected based on the success of the corresponding resource operation on the server. To change the data for a record, a jQuery event is also defined on a save button to update the corresponding `eCustomer` record in JSDO memory with the current field values entered in a customer detail form (`#custdetail`):

```
dataSet = new progress.data.JSDO( { name: 'dsCustomerOrder',
                                    autoApplyChanges : false } );
dataSet.eCustomer.subscribe('afterDelete', onAfterCustomerChange, this);
dataSet.eCustomer.subscribe('afterCreate', onAfterCustomerChange, this);
dataSet.eCustomer.subscribe('afterUpdate', onAfterCustomerChange, this);

$('#btnSave').bind('click', function(event) {
  var jsrecord = dataSet.eCustomer.findById($('#custdetail #id').val());
  if (jsrecord) {jsrecord.assign();};
  dataSet.saveChanges();
});

// Similar controls might be defined to delete and create eCustomer records...

function onAfterCustomerChange(jsdo, record, success, request) {
  if (success)  {
    record.acceptRowChanges();
    // Perform other actions associated with accepting this record change
  }
  else
  {
    record.rejectRowChanges();
  }
}
```

When the button is clicked, the event handler uses the `findById( )` method to find the original record with the matching internal record ID (`#id`) and invokes the `assign( )` method on `jsrecord` with an empty parameter list to update its fields in `eCustomer` with any new values entered into the form. It then calls the `saveChanges( )` method to invoke the resource Update operation to apply these record changes to the server. You might define similar events and controls to delete the `eCustomer` record or add a new `eCustomer` record.

After each resource operation for a changed `eCustomer` record completes, results of the operation are returned to the client from the server, and the appropriate event fires. If the operation was successful, the handler calls `acceptRowChanges( )` to accept the record change associated with the event in JSDO memory. An advantage of using an event to manually accept a record change is that you can perform other actions associated with accepting this particular change, such as creating a local log that describes the change.

Note: This example shows the default invocation of `saveChanges( )`, which invokes each resource record-change operation, one record at a time, across the network. For a resource that supports before-imaging (such as an OpenEdge ProDataSet), you can also have `saveChanges( )` send all pending record change operations across the network in a single Submit operation. For more information and an example, see the description of the `saveChanges( )` method.

See also:

acceptChanges( ) method on page 153, autoApplyChanges property on page 202, autoSort property on page 203, caseSensitive property on page 213, rejectRowChanges( ) method on page 308, saveChanges( ) method on page 314

add( ) method

Creates a new record object for a table referenced in JSDO memory and returns a reference to the new record.

To synchronize the change on the server, call the `saveChanges( )` method.

Alias: `create( )`

Return type: progress.data.JSRecord class on page 133

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Working record: After completing execution, the new record becomes the working record for the associated table. If the table has child tables, the working record for these child tables is not set.

Syntax

```
jsdo-ref.add ( [new-record-object] )
jsdo-ref.table-ref.add ( [new-record-object] )
```

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference.

table-ref

A table reference on the JSDO.

new-record-object

If specified as a non-`null` object, passes in the data to create the record for the `JSRecord` instance. The data to create the record is identified by one or more properties, each of which has the name of a corresponding field in the table schema and has the value to set that field in the new table record.

If you omit or set the parameter to `null`, or you do not include properties of *new-record-object* for all fields in the new record, the method uses the default values from the table schema stored in the catalog to set the unspecified record fields.

Note: After this method adds the new record object, and if you have set up automatic sorting using the `autoSort` property, all the record objects for the affected table reference are sorted accordingly. If the sorting is done using sort fields, any `string` fields are compared according to the value of the `caseSensitive` property.

If the specified table reference is for a child table in a multi-table resource, when the `useRelationships` property is `true`, `add( )` uses the relationship to set related field values of the new child record from the working record of the parent table. However, if the working record of the parent is not set, `add( )` throws an error. If `useRelationships` is `false`, the fields for the new record are set as specified by *new-record-object* and no error is thrown.

Example

Assuming `useRelationships` is `true`, given a JSDO created for a multi-table resource with a `customer` and related child `order` table, the `add( )` method in the following code fragment uses this relationship to automatically set the `CustNum` field in a new record added to the `order` table:

```
var dataSet = new Progress.data.JSDO( 'CustomerOrderDS' );
dataSet.customer.add( { CustNum: 1000, Balance: 10000, State: 'MA' } );

// CustNum is set automatically by using the relationship
dataSet.order.add( { OrderNum: 1000 } );
```

See also:

autoSort property on page 203, caseSensitive property on page 213, fill( ) method on page 220, getId( ) method on page 242, getSchema( ) method on page 247, data property on page 217, saveChanges( ) method on page 314

addCatalog( ) method (JSDOSession class)

Note: Updated for Progress Data Objects Version 4.4.1 and later.

Loads one or more local or remote Data Service Catalogs into the current `JSDOSession` object.

The appropriate Catalog must be loaded before creating a JSDO for any resource defined in the Catalog.

If a Catalog is on a web server (remote), this method throws an exception if it is not possible to send a request to the specified web application.

This method is always executed asynchronously and returns results in callbacks that you register using methods of a Promise object returned as the method value.

Note: In order to invoke this method successfully, jQuery Promises must be supported in your development environment. Otherwise, the method throws an exception.

Note: You can call this method either before or after the `JSDOSession` object has successfully established a JSDO login session using its `login( )` method. However, if the Data Service Catalog is stored remotely in the web application that hosts the Data Object Service, Progress recommends that you log into the web application **before** calling this method.

Note: As a recommended alternative to using this method on an existing `JSDOSession` object, you can create a new, initialized `JSDOSession` using the `progress.data.getSession( )` stand-alone function. In one call, this function instantiates the object, invokes its `login( )` method with specified credentials, then invokes its `addCatalog( )` method to load a specified Data Service Catalog. For more information, see the getSession( ) stand-alone function on page 248 description.

Return type: jQuery Promise

Applies to: progress.data.JSDOSession class on page 124

Syntax

```
addCatalog ( catalog-uri [ , cat-user-name , cat-password ]
             [ , parameter-object ] )
addCatalog ( catalog-uris [ , cat-user-name , cat-password ]
             [ , parameter-object ] )
```

catalog-uri

A `string` expression that specifies the URI of a Data Service Catalog file. This URI can specify either a location (remote) in a web application running on a web server or a location (local) that is relative to the device where the app is currently running. If the URI specifies a remote location, it is typically the location of the web application that hosts the Data Object Service and where the `JSDOSession` object has a JSDO login session. If the URI is a relative path (e.g., `catalogs/OrderEntrySvc.json`), the *catalog-uri* is assumed to be relative to the location from which the app was loaded.

If the mobile app from which you are logging in is a hybrid app that will be installed to run directly in a native device container, or if it is a web app deployed to a different web server from the web application to which the `JSDOSession` object has already logged in, you must specify an absolute URI for the Catalog that includes the Tomcat server domain or host and port, for example, `http://www.progress.com:8980/SportsMobileApp/static/OrderEntrySvc.json`, or perhaps for testing, `http://localhost:8980/SportsMobileApp/static/OrderEntrySvc.json`.

If the JSDOSession object has already logged in and the mobile app is a web app deployed to the same Apache Tomcat server as the web application that hosts the Data Object Service, you can specify a URI that is relative to the deployment end point (Tomcat server domain or host and port), for example: /SportsMobileApp/static/OrderEntrySvc.json, where /SportsMobileApp is the location of the web application.

Note: The default Catalog URI for a Catalog created for an OpenEdge Data Object Service, relative to the Apache Tomcat server domain or host and port where the session is logged in, is the following: */WebApplicationName*/static/*ServiceName*.json, where *WebApplicationName* is the name of the web application and *ServiceName* is the name of the Data Object Service for which the Data Service Catalog is created.

catalog-uris

An array of string expressions, each of which is the URI for a Catalog as defined for the *catalogURI* parameter. If any of the Catalogs in the array are protected separately from the web application where the JSDOSession object is logged in, you can specify the additional credentials using *cat-user-name* and *cat-password*.

Note: You can read the catalogURIs property to return the URIs for all Catalogs previously loaded into the JSDOSession object.

cat-user-name

A string expression containing a user ID to authenticate access to a protected Catalog. If you do not specify *cat-user-name*, Catalog access is authorized using existing user credentials (if any).

cat-password

A string expression containing a password (if required) to authenticate the user specified by *cat-user-name*.

parameter-object

An object that contains the following optional property:

- **iOSBasicAuthTimeout** — A number that specifies the time, in milliseconds, that the addCatalog( ) method waits for a response before returning an error. This error might mean that the user entered invalid credentials to access a protected Catalog URI. If you set this value to zero (0), no timeout is set, and addCatalog( ) can wait indefinitely before returning an error. If you do not set the iOSBasicAuthTimeout property, addCatalog( ) uses 4000 (4 seconds) as the default value.

  Note: Any non-zero timeout value (default or otherwise) for this *parameter-object* property operates **only** under certain conditions. Otherwise, any setting of this property has no effect. For more information, see the notes of the progress.data.JSDOSession class on page 124 description.

Note: If the JSDOSession object is already logged in, you typically do not need to specify *cat-user-name* and *cat-password*. These optional parameters are available primarily if the object is not yet logged in, or if you store the Catalog someplace on the web other than the web application that hosts the Data Object Service (where the JSDOSession object has a JSDO login session).

Caution: Although you can use a single `JSDOSession` object to load a Catalog that is stored somewhere on the web other than the web application that hosts the Data Object Service, this typically works only if the Catalog is protected with Anonymous or HTTP Basic authentication.

Promise method signatures

jQuery Promise objects define methods that register a callback function with a specific signature. The callback signatures depend on the method that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that `addCatalog( )` returns:

Syntax:

```
promise.done( function ( session , result , details ) )
promise.fail( function ( session , result , details ) )
promise.always( function ( session , result , details ) )
```

promise

A reference to the Promise object that is returned as the value of the `addCatalog( )` method. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDOSession class on page 124.

session

A reference to the `JSDOSession` object on which `addCatalog( )` was called.

result

A constant indicating the overall result of the call that can have one of the following values:

- **`progress.data.Session.SUCCESS`** — Each Catalog specified by the *`catalogURI`* or *`catalogURIs`* parameter of `addCatalog( )` has been loaded successfully or has already been loaded.
- **`progress.data.Session.GENERAL_FAILURE`** — One or more of the specified Catalogs has failed to load successfully.

Note: It is not always necessary to test the value of *`result`* in a Promise method callback for the `addCatalog( )` method, especially if the callback is registered using *`promise`*`.done( )` and *`promise`*`.fail( )`, where each callback always executes with the same value for success (`done( )`) or failure (`fail( )`).

details

An array of JavaScript objects that contain information on the Data Service Catalogs that `addCatalog( )` attempted to load. Each object has the following properties:

- **`catalogURI`** — The URI of a specified Catalog.
- **`result`** — A constant indicating the result of loading the Catalog that can have one of the following values:
  - **`progress.data.Session.SUCCESS`** — The specified Catalog loaded successfully, or has previously been loaded.

- **progress.data.Session.CATALOG_ALREADY_LOADED** — The specified Catalog was previously loaded.

  Note: When `addCatalog( )` is requested to load a Data Service Catalog that is already loaded, it does not load the Catalog again, but is considered a successful execution. That is, the *promise*`.done( )` callback is called, although the `result` value in the *details* object for the specified Catalog is `progress.data.Session.CATALOG_ALREADY_LOADED`.

- **progress.data.Session.AUTHENTICATION_FAILURE** — The specified Catalog failed to load because of an authentication error.
- **progress.data.Session.GENERAL_FAILURE** — The specified Catalog failed to load because of an error other than an authentication failure.

  Note: This value can also be returned if invalid user credentials triggered a Catalog access timeout according to the value of the *parameter-object*`.iOSBasicAuthTimeout` property.

- **errorObject** — Any error object thrown while attempting to load the Catalog.

  Note: If this object is thrown because of a Catalog access timeout triggered according to the value of the *parameter-object*`.iOSBasicAuthTimeout` property, the `message` property of the error object is set to `"addCatalog timeout expired"`.

- **xhr** — A reference to the XMLHttpRequest object used to load the Catalog from a web server.

Example

The following code fragment calls the `addCatalog(myCatalogURIs)` method on the `JSDOSession` object, `mySession`, to load multiple Data Service Catalogs specified in `myCatalogURIs`. This example uses `try` and `catch` blocks to check any error object thrown prior to requesting the Catalog, and assembles an appropriate message to display in an alert box or does other processing for each case:

```
try {
    mySession.addCatalog( myCatalogURIs ).done(
        function( session, result, details ) {
            alert("All Catalogs loaded.");
        }).fail(
        function( session, result, details ) {
            var numCats = details.length;
            for ( i = 0; i < numCats; i++ ) {
                if (details[i].result
                    === progress.data.Session.AUTHENTICATION_FAILURE) {
                    alert("Authentication error: " + details[i].catalogURI);
                } else if (details[i].result
                           === progress.data.Session.GENERAL_FAILURE) {
                    alert("General Catalog load error: "
                          + details[i].catalogURI);
                    if (details[i].errorObject) {
                        // Process thrown error object during load . . .
                    }
                    if (details[i].xhr) {
                        // Process XHR object sent for the load . . .
                    }
                } else {
                    alert("Not sure what is wrong with "
                          + details[i].catalogURI);
                }
            } // for each load attempt
        });
}
catch(errObj) {
    var msg;

    msg = '\n' + errObj.message;
    alert("Unexpected addCatalog() error: " + msg);
}
```

See also:

catalogURIs property on page 215, getSession( ) stand-alone function on page 248, login( ) method (JSDOSession class) on page 275

addCatalog( ) method (Session class)

Note: Deprecated in Progress Data Objects Version 4.4 and later. Please use the addCatalog( ) method (JSDOSession class) on page 159 instead.

Loads a local or remote Data Service Catalog into the current `Session` object.

The appropriate Catalog must be loaded before creating a JSDO for any resource defined in the Catalog.

If a Catalog is on a web server (remote), this method throws an exception if it is not possible to send a request to the specified web application.

This method can be called synchronously or asynchronously, depending on parameters that you pass.

Note: You can call this method either before or after the `Session` object has successfully established a JSDO login session using its `login( )` method. However, if the Data Service Catalog is stored remotely in the web application that hosts the Data Object Service, Progress recommends that you log into the web application **before** calling this method.

Return type: `number`

Applies to: progress.data.Session class on page 136

Syntax

```
addCatalog ( catalog-uri [ , cat-user-name [ , cat-password ] ] )
addCatalog ( parameter-object )
```

Note: If you call `addCatalog( )` passing *`catalog-uri`*, the method executes synchronously. If you call it passing *`parameter-object`*, the method executes synchronously or asynchronously, depending on the setting of *`parameter-object`*.

`catalog-uri`

A `string` expression that specifies the URI of a Data Service Catalog file. This URI can specify either a location (remote) in a web application running on a web server or a location (local) that is relative to the device where the app is currently running. If the URI specifies a remote location, it is typically the location of the web application that hosts the Data Object Service and where the `Session` object has a JSDO login session. If the URI is a relative path (e.g., `catalogs/OrderEntrySvc.json`), the *`catalog-uri`* is assumed to be relative to the location from which the app was loaded.

If the mobile app from which you are logging in is a hybrid app that will be installed to run directly in a native device container, or if it is a web app deployed to a different web server from the web application to which the `Session` object has already logged in, you must specify an absolute URI for the Catalog that includes the Tomcat server domain or host and port, for example, `http://www.progress.com:8980/SportsMobileApp/static/OrderEntrySvc.json`, or perhaps for testing, `http://localhost:8980/SportsMobileApp/static/OrderEntrySvc.json`.

If the `Session` object has already logged in and the mobile app is a web app deployed to the same Apache Tomcat server as the web application that hosts the Data Object Service, you can specify *`catalog-uri`* as a relative URI, for example: `/SportsMobileApp/static/OrderEntrySvc.json`, which is relative to the deployment end point (Tomcat server domain or host and port).

Note: The default Catalog URI for a Catalog created for an OpenEdge Data Object Service, relative to the Apache Tomcat server domain or host and port where the session is logged in, is the following: */`WebApplicationName`*`/static/`*`ServiceName`*`.json`, where *`WebApplicationName`* is the name of the web application and *`ServiceName`* is the name of the Data Object Service for which the Data Service Catalog is created.

cat-user-name

A string expression containing a user ID to authenticate access to a protected Catalog. If you do not specify *cat-user-name*, Catalog access is authorized using existing user credentials (if any).

cat-password

A string expression containing a password (if required) to authenticate the user specified by *cat-user-name*.

parameter-object

An object that has one or more of the following properties:

- **catalogURI** — (Required) Same value as the *catalog-uri* parameter.
- **userName** — (Optional) Same value as the *cat-user-name* parameter.
- **password** — (Optional) Same value as the *cat-password* parameter.
- **async** — (Optional) A boolean that, if true, tells addCatalog( ) to execute asynchronously. If false or absent, addCatalog( ) executes synchronously.
- **iOSBasicAuthTimeout** — A (Optional) number that specifies the time, in milliseconds, that the addCatalog( ) method waits for a response before returning an error. This error might mean that the user entered invalid credentials. If you set this value to zero (0), no timeout is set, and addCatalog( ) can wait indefinitely before returning an error. If you do not set the iOSBasicAuthTimeout property, addCatalog( ) uses 4000 (4 seconds) as the default value.

  Note: Any non-zero timeout value (default or otherwise) for this *parameter-object* property operates **only** under certain conditions, **including** when addCatalog( ) is executed asynchronously. Otherwise, any setting of this property has no effect. For more information, see the notes of the progress.data.Session class on page 136 description.

Note: If the Session object is already logged in, you typically do not need to specify *cat-user-name* and *cat-password* (or the equivalent userName and password properties). These optional parameters are available primarily if the object is not yet logged in, or if you store the Catalog somewhere other than in the web application that hosts the Data Object Service (where the Session object has a JSDO login session).

Caution: Although you can use a single Session object to load a Catalog that is stored somewhere on the web other than the web application that hosts the Data Object Service, this typically works only if the Catalog is protected with Anonymous or HTTP Basic authentication.

Caution: To help ensure that HTTP Forms access to web applications works in certain web browsers, such as Firefox, when the web application is configured for Cross-Origin Resource Sharing (CORS), always call the addCatalog( ) method asynchronously.

You can read the catalogURIs property to return the URIs for all catalogs previously loaded for the login session.

When the method completes, it returns one of the following numeric constants to indicate the result:

- **progress.data.Session.ASYNC_PENDING** — The method is called asynchronously, or if there is an error preparing to send the request to the server, it throws an Error object. All asynchronous execution results (including any server errors) are returned in the afterAddCatalog event that is fired on the

`Session` object after the method completes. Before calling the method, you can subscribe one or more handlers to this event using the `subscribe( )` method on the `Session` object.

- **`progress.data.Session.SUCCESS`** — The specified Data Service Catalog loaded successfully.
- **`progress.data.Session.CATALOG_ALREADY_LOADED`** — The specified Data Service Catalog did not load because it is already loaded.
- **`progress.data.Session.AUTHENTICATION_FAILURE`** — The Catalog failed to load because of a user authentication error.
- **`progress.data.Session.GENERAL_FAILURE`** — The specified Catalog failed to load because of an error other than an authentication failure.

It is also possible for this method to throw an `Error` object, in which case it does not return a value at all. For more detailed information about any response (successful or unsuccessful) returned from the web server, you can also check the XMLHttpRequest object (XHR) returned by the `lastSessionXHR` property.

Note: When called asynchronously, any handler for the `afterAddCatalog` event is passed the same information that is available for the `addCatalog( )` method when it is called synchronously, including the `Session` object that executed `addCatalog( )` (with the same property settings) and any `Error` object that was thrown processing the server response.

Caution: To help ensure that HTTP Forms access to web applications works in certain web browsers, such as Firefox, when the web application is configured for Cross-Origin Resource Sharing (CORS), always call the `addCatalog( )` method asynchronously.

Example

The following code fragment calls the `addCatalog( )` method synchronously on the session, `empSession` by omitting the `async` property from its object parameter. For a similar example with the method called asynchronously, see the reference entry for the `afterAddCatalog` event. This synchronous example uses `try` and `catch` blocks to check for either expected success and failure return values, or a thrown `Error` object with an unknown error, and assembles an appropriate message to display in an alert box for each case:

```
try {
    var retValue;

    retValue = empSession.addCatalog( {
        catalogURI :
            "http://localhost:8980/SportsMobileApp/static/OrderEntrySvc.json" } );

    if (retValue ===  progress.data.Session.LOGIN_AUTHENTICATION_FAILURE ) {
        alert("add Employee Catalog failed. Authentication error");
    }
    else if ( retValue ) {
        alert("Catalogs loaded.");
    }
}
catch(errObj) {
    var msg;

    msg = '\n' + errorObject.message;
    alert("unexpected addCatalog error." + msg);
}
```

See also:

afterAddCatalog event on page 179, catalogURIs property on page 215, lastSessionXHR property on page 274, login( ) method (Session class) on page 280, subscribe( ) method (Session class) on page 349

addItem( ) method

Adds an item to a list view based on the working record of a specified table reference.

The appearance and content of the list view item are controlled by a specified template and format, respectively:

- The template is defined by `setItemTemplate( )`, optionally overridden for this list view by `setListView( )`.
- The format is defined by `setListView( )`, optionally overridden for this item by the `format` parameter.

Return type: `null`

Applies to: progress.ui.UIHelper class on page 142, table reference property (UIHelper class) on page 353

Syntax

```
uihelper-ref.addItem ( [ format ] )
uihelper-ref.table-ref.addItem ( [ format ] )
```

uihelper-ref

A reference to a `UIHelper` instance. You can call the method on *uihelper-ref* if the JSDO associated with the instance has only a single table reference, and that table reference has a working record.

table-ref

A table reference that has a working on the JSDO associated with the `UIHelper` instance.

format

A `String` value specifying the fields displayed for the list item, overriding the format defined by `setListView( )`.

Example

The following code fragment adds an item to the list view associated with the working record, using the specified format, where `dataSet` is a JSDO that has the table reference, `eCustomer`:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );
uihelper = new progress.ui.UIHelper( { jsdo: dataSet } );
uihelper.eCustomer.addItem ( '{CustNum}<br>{Name}<br>{State}<br>{Country}' )
```

See also:

setListView( ) method on page 336, setItemTemplate( ) method on page 335

addLocalRecords( ) method

Reads the record objects stored in the specified local storage area and updates JSDO memory based on these record objects, including any pending changes and before-image data, if they exist.

This method updates all affected tables of the JSDO with the locally stored record objects. These record objects are merged into JSDO memory and affect existing data according to a specified merge mode and optional key fields.

Return type: `boolean`

Applies to: progress.data.JSDO class on page 110

Working record: After execution, the working record set for each table in the JSDO depends on the specified merge mode.

Syntax

```
addLocalRecords ( [ storage-name , ] add-mode [ , key-fields ] )
```

storage-name

The name of the local storage area from which to update JSDO memory. If *storage-name* is not specified, blank, or `null`, the name of the default storage area is used. The name of this default area is `jsdo_`*serviceName*`_`*resourceName*, where *serviceName* is the name of the Data Object Service that supports the JSDO instance, and *resourceName* is the name of the resource (table, dataset, etc.) for which the JSDO instance is created.

Note: Record objects read in from local storage can contain before-image data, which this method merges into JSDO memory along with the data from the record objects. However, if a record object read in from local storage contains before-image data that conflicts with existing before-image data in JSDO memory for that same record object, `addLocalRecords( )` throws an exception.

add-mode

An integer constant that represents a merge mode to use. Each merge mode handles duplicate keys in a particular manner, depending on your specification of *key-fields*. You can specify the following numeric constants, which affect how the table record objects in the specified local storage area are added to JSDO memory:

- **`progress.data.JSDO.MODE_REPLACE`** — Adds the table record objects in the specified local storage area to the existing record objects in JSDO memory. If duplicate keys are found between record objects in local storage and record objects in JSDO memory, the record objects with duplicate keys in JSDO memory are replaced with the corresponding records in local storage.

  Note: For the current release, only this single merge mode is supported. Use of any other merge mode (for example, as specified for the `addRecords( )` method) throws an exception.

Caution:

If any specified *key-fields* match the unique indexes of corresponding tables on the server, adding the contents of the specified local storage area can result in records with duplicate keys. If the corresponding server tables have unique indexes, you must make any affected duplicate key fields unique before calling `saveChanges( )`.

key-fields

An object with a list of primary key fields to check for records with duplicate keys. For example, when merging with a multi-table resource that has `eCustomer` and `eOrder` table references, you might use the following object:

```
{
  eCustomer: [ "CustNum" ],
  eOrder: [ "CustNum", "Ordernum" ]
}
```

When merging with a single-table reference, you might use the following array object:

```
[ "CustNum", "Ordernum" ]
```

Note: For any *key-fields* that have the `string` data type, the character values for these fields are compared to identify duplicates according to the value of the `caseSensitive` property on each affected table reference.

If *key-fields* is specified, the method checks for duplicate keys using the specified primary keys found in *key-fields*. If *key-fields* is **not** specified, the method searches other possible sources for definitions of primary keys in the following order, and uses the first source of definitions found:

1. Primary key annotations from any OpenEdge resource (as identified in the Data Service Catalog)
2. Unique ID properties associated with the resource (for example, the `idProperty` property as identified in the Data Service Catalog for a Rollbase object)

If no source of primary key definitions is found, the method adds **all** local storage records to JSDO memory, regardless of the specified *add-mode*, and regardless of any duplicate records that might result.

Note: For Rollbase resources, all tables that correspond to Rollbase objects have a defined primary key. For OpenEdge resources, the Progress Developer Studio for OpenEdge defines the primary key automatically using a `primaryKey` annotation when you create a Business Entity for the resource, either with a new Express project or with the New Business Entity wizard using the **Select database table** option. If you use the **Select schema from file** option in the New Business Entity wizard or you create the Business Entity manually, you must add this annotation for each temp-table definition yourself, which you can do using the Define Service Interface wizard in Developer Studio. For more information on service interface annotations, see the Progress Developer Studio for OpenEdge online help.

Note: After this method checks for any duplicate keys and completes adding record objects to JSDO memory, and if you have set up automatic sorting using the `autoSort` property, all the record objects for the affected table references are sorted accordingly. If the sorting is done using sort fields, any `string` values in the specified sort fields are compared according to the value of the `caseSensitive` property.

This method returns `true` if it successfully reads the data from the local storage area; it then updates JSDO memory with this data according to the specified *`add-mode`*. If *`storage-name`* does not exist, but otherwise encounters no errors, the method leaves JSDO memory unchanged and returns `false`. If the method does encounter errors (for example, with reading the data in the specified storage area), it also leaves JSDO memory unchanged and throws an exception.

Example

The following code fragment fills memory for a JSDO (`dataset`) with records from an OpenEdge ProDataSet named `csCustomerOrder`, which is used to implement a multi-table resource on the server. The JSDO then contains tables that correspond to the `Customer` and `Order` tables of the OpenEdge `sports2000` database:

```
var dataset = progress.data.JSDO( "dsCustomerOrder" );
dataset.fill(); // Loads the JSDO with all available records from the ProDataSet resource

// Adds records
dataset.addLocalRecords( progress.data.JSDO.MODE_REPLACE, [ "CustNum", "Ordernum" ] );
```

The fragment then calls `addLocalRecords( )` on the JSDO to add a set of similar records to JSDO memory from the default local storage area, where the records were previously stored using the JSDO `saveLocal( )` method. Duplicate `Customer` and `Order` records are checked and replaced with the records from local storage based on the respective primary key fields, `CustNum` and `Ordernum`.

See also:

autoSort property on page 203, caseSensitive property on page 213, fill( ) method on page 220, getId( ) method on page 242, readLocal( ) method on page 304, saveChanges( ) method on page 314, saveLocal( ) method on page 330

addPlugin( ) method

Note: Applies to Progress Data Objects Version 4.3 or later.

Adds (registers) a JSDO plugin object with a specified name in the current document context that can be used with a supported Data Object operation invoked on a JSDO instance.

Return type: `null`

Applies to: progress.data.PluginManager class on page 135

Syntax

```
progress.data.PluginManager.addPlugin( name , plugin )
```

name

A `string` expression that evaluates to the name of the plugin. If a plugin by this name already exists, the method throws an exception.

Note: The `PluginManager` class provides access to a built-in plugin with the reserved name, `"JFP"`, which you can customize as described further in this topic. You can also customize any other JSDO plugin that you have already registered.

plugin

A plugin `Object` that you can define with certain properties. Each property references the definition of a function that you provide. Each function executes according to the requirements of the supported Data Object operation invoked by any JSDO instance.

This method currently supports only MappingType plugins registered for the Read operation. A MappingType plugin object can contain any or all of the following properties:

- **`requestMapping`** — Defines a function that **must** return an object with properties that define the parameters passed to the JSDO Read operation method (`fill( )` or `read( )`). This function thus allows you to specify or modify any or all of the properties that you can include in a Read operation method's *`parameter-object`* argument before the method executes, thus affecting how the Read operation request is invoked across the network. The function must have the following signature:

```
function ( jsdo , params , info )
```

jsdo

A reference to the `progress.data.JSDO` object that corresponds to the JSDO on which the Read operation method was invoked.

params

A reference to the `Object` passed to the Read operation method as its *`parameter-object`* argument.

info

A reference to an `Object` with a single `operation` property set to the value, `"read"`.

- **`responseMapping`** — Defines a function that **must** return an object with properties that provide the data returned from the corresponding Read operation executed on the server. This function allows you to process the network response from the Read operation in some way. The function must have the following signature:

```
function ( jsdo , response , info )
```

jsdo

A reference to the `progress.data.JSDO` object that corresponds to the JSDO on which the Read operation method was invoked.

response

A reference to the `Object` that contains data properties returned from the Read operation across the network. For more information, see the description of the response property on page 312 of JSDO request object for returning Read operation output.

Note: If this `responseMapping` function customizes the built-in `"JFP"` plugin or defines a custom plugin based on this built-in plugin, in the body of the function, you can set the value of the JSDO user-defined `"server.count"` property to a corresponding *response* property that holds the total count of records returned in the Read operation result set. This has the equivalent effect of specifying a Count operation for the server Data Object to return the same value. For more information, see the Example on page 173 in this topic.

info

A reference to an `Object` with a single `operation` property set to the value, `"read"`.

The built-in `"JFP"` MappingType plugin is available to support JSDO access for Rollbase or by a Kendo UI DataSource. For more information on this `"JFP"` plugin, see the description of the fill( ) method on page 220. Although you cannot create your own plugins with this name, you can reference the built-in `"JFP"` plugin, for example, to create a custom `requestMapping` or `responseMapping` function for it, or to create an entirely new plugin based on the existing built-in `"JFP"` plugin. For more information see the Example on page 173 in this topic.

To allow an OpenEdge Data Object Read operation to use either the built-in `"JFP"` or your custom MappingType plugin, you must annotate the ABL method that implements the operation to identify the plugin to use. For more information, see the Notes on page 174 in this topic.

Example

The following JSDO plugin example adds a new plugin, `"MYJFP"`, which is based on the built-in `"JFP"` plugin. The example begins by returning a reference to the built-in `"JFP"` plugin object in the current document context as `jfpPlugin` and invokes `addPlugin( )` to register the new `"MYJFP"` plugin, as follows:

```
jsdo = new progress.data.JSDO( . . . );

var jfpPlugin = progress.data.PluginManager.getPlugin("JFP");

progress.data.PluginManager.addPlugin("MYJFP", {

    requestMapping: function(jsdo, params, info) {

        var requestParams = {},
            object = {};
        params = jfpPlugin.requestMapping(jsdo, params, info);
        if (params && typeof params.filter === "string") {
            object = JSON.parse(params.filter);
        }
        object.mydata = jsdo.getProperty("mydata");
        requestParams.filter = JSON.stringify(object);

        // You must return the [updated] parameter object
        return requestParams;

    }

    responseMapping: function(jsdo, response, info) {
```

```
        var record;
        var newData = response.dsEmployee.ttEmployee;
        if (info.operation === "read") {

            for (var i = 0; i < newData.length; i++) {
                record = newData[i];
                record.VacDays = record.VacDays + 10;
            }

            jsdo.setProperty("server.count", response.myTotal);
        }

        // You must return the [updated] response object
        return response;

    }
});
```

For the new `requestMapping` function, the example builds and returns a custom `params` object (`requestParams`) from both the `params` object returned by the existing `requestMapping` function invoked on the built-in `"JFP"` plugin and an additional `mydata` property returned as a custom JSDO property that has already been set on the currently executing JSDO instance (`jsdo`).

For the new `responseMapping` function, the example first tests that the function is executing in the context of a Read operation and applies a common update to the returned `dsEmployee.ttEmployee` table, adding 10 days to the `VacDays` field of each record. It then sets the reserved custom JSDO property, `"server.count"`, on the currently executing JSDO instance (`jsdo`) to the custom value returned from the server as `response.myTotal` before returning the updated `response` from the Read operation.

Notes

- If you want to dynamically update the definition for an existing plugin, whether it is the built-in `"JFP"` plugin or a custom plugin you have previously registered, you can invoke the `getPlugin( )` method to return a reference to the plugin object and assign the appropriate property to the updated function definition. For example, to assign an updated definition for the `responseMapping` function of an existing custom plugin named `"myReadPlugin"`, you can invoke code similar this:

  ```
  progress.data.PluginManager.getPlugin("myReadPlugin").responseMapping
      = function(jsdo, response, info) { /* Your new definition */ };
  ```

  For an example, see the description of the getPlugin( ) method on page 244.

- For an OpenEdge Data Object Read operation to use the specified MappingType plugin from the client document context, you must properly annotate the definition of the Read operation method in the ABL Business Entity that defines the server Data Object. This annotation defines a `mappingType` property in the Data Service Catalog with a string value that is identical to the name of a MappingType plugin you have registered for the JSDO instance on the client. You specify this annotation with the following syntax:

  ```
  @openapi.openedge.method.property (name="mappingType", value="plugin-name").
  ```

Where "*plugin-name*" is the quoted name that you used to register the custom plugin with the addPlugin( ) method. For example, on the definition of an ABL Read operation method, ReadEmployee( ), you might add this annotation for a plugin registered with the name "MYJFP":

```
@openapi.openedge.export(type="REST", useReturnValue="false",
writeDataSetBeforeImage="true").
@progress.service.resourceMapping(type="REST", operation="read",
URI="?filter=~{filter~}", alias="", mediaType="application/json").
@openapi.openedge.method.property (name="mappingType", value="MYJFP").
METHOD PUBLIC VOID ReadEmployee(
    INPUT filter AS CHARACTER,
    OUTPUT DATASET dsEmployee):
    SUPER:ReadData(filter).

END METHOD.
```

When this ReadEmployee( ) method executes on the server, it uses the parameter values passed by the client JSDO and provided by any requestMapping function defined in the "MYJFP" JSDO plugin, and the JSDO then handles the response according to the processing provided by any responseMapping function defined in the same plugin.

Note: When annotating for use of the built-in "JFP" plugin only, you must also add an annotation that defines an associated capabilities property for that plugin. For more information on annotations required for using the built-in "JFP" plugin, see the description of Business Entity updates for access by the Kendo UI DataSource and Rollbase external objects in *OpenEdge Development: Web Services*. However note, if you are annotating a custom plugin that is **based** on the built-in "JFP" plugin (such as the sample "MYJFP" plugin annotated here), you do **not** also need to annotate an associated capabilities property for that custom plugin.

See also

fill( ) method on page 220, getPlugin( ) method on page 244, progress.data.JSDO class on page 110

addRecords( ) method

Updates JSDO memory with one or more record objects read from an array, single-table, or multi-table resource that are passed in an object parameter, including any pending changes and before-image data, if they exist.

This method updates all tables or a specified table in JSDO memory, depending on how the method is called. The data is merged into JSDO memory and affects existing data according to a specified merge mode and optional key fields.

Return type: null

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Working record: After execution, the working record set for each table in the JSDO depends on the specified merge mode.

Syntax

```
jsdo-ref.addRecords ( merge-object , add-mode [ , key-fields ] )
jsdo-ref.table-ref.addRecords ( merge-object , add-mode [ , key-fields ] )
```

jsdo-ref

A reference to the JSDO. If you call the method on *jsdo-ref*, the method merges data for all referenced tables in JSDO memory.

table-ref

A table reference on the JSDO. If you call the method on *table-ref*, the method merges data only for the referenced table in JSDO memory.

merge-object

An object with the data to merge. If you call the method on *table-ref*, the object can either be an object that contains an array of record objects to merge with the referenced table or a multi-table-formatted object containing such an array.

Note: This object must have a supported JavaScript object format that matches the data returned from the resource Read operation (JSDO `fill( )` method). For example, an object returned as output from an invocation method called on a single-table or multi-table resource will work if it has the same schema as output from the resource Read operation.

The following formats are supported for *merge-object*:

- A single table object with an array of record objects. For example:

```
{
  eCustomer: [
    // Record objects ...
  ]
}
```

- An array of record objects for either a single table object or a multi-table object containing only a single table object. For example:

```
[
  // Record objects ...
]
```

- A multi-table object with a single table object or multiple table objects at the same level only. For example:

```
{
  dsCustomerOrder: {
    eCustomer: [
      // Record objects ...
    ],
    eOrder: [
      // Record objects ...
    ]
  }
}
```

Note: The record objects passed in *merge-object* can contain before-image data, which this method merges into JSDO memory along with the data from the record objects. However, if a record object read in from local storage contains before-image data that conflicts with existing before-image data in JSDO memory for that same record object, `addLocalRecords( )` throws an exception.

add-mode

An integer that represents a merge mode to use. If you also specify *key-fields*, each merge mode handles duplicate keys in a particular manner as described here. If you **do not** specify *key-fields*, the method adds **all** the records of *merge-object* regardless of the mode. You can specify the following numeric constants, which affect how the table record objects in *merge-object* are added to JSDO memory:

- **progress.data.JSDO.MODE_APPEND** — Adds the table record objects in *merge-object* to the existing record objects in JSDO memory. If a duplicate key is found between a record object in *merge-object* and a record object in JSDO memory, the method throws an error.
- **progress.data.JSDO.MODE_MERGE** — Adds the table record objects in *merge-object* to the existing record objects in JSDO memory. If duplicate keys are found between record objects in *merge-object* and record objects in JSDO memory, the method ignores (does not add) the record objects with duplicate keys in *merge-object*.
- **progress.data.JSDO.MODE_REPLACE** — Adds the table record objects in *merge-object* to the existing record objects in JSDO memory. If duplicate keys are found between record objects in *merge-object* and record objects in JSDO memory, the record objects with duplicate keys in JSDO memory are replaced with the corresponding records in *merge-object*.
- **progress.data.JSDO.MODE_EMPTY** — Empties all table record objects from JSDO memory and replaces them with the contents of *merge-object*.

Note: If *merge-object* is an empty object (`{}`), this mode effectively empties the data from JSDO memory.

After execution, if the specified merge mode was `progress.data.JSDO.MODE_EMPTY`, the working record set for any table references is `undefined`, because JSDO memory is completely emptied or replaced. For any other merge mode, the working record set for each JSDO table reference remains unchanged.

Caution:

If a table *key-fields* matches the unique indexes of corresponding tables, adding the contents of *merge-object* can result in records with duplicate keys. If the corresponding tables have unique indexes, you must make any affected duplicate key fields unique before calling `saveChanges( )`.

key-fields

An object with a list of key fields to check for records with duplicate keys. For example, when merging into a JSDO that has `eCustomer` and `eOrder` table references, you might use the following object:

```
{
  eCustomer: [ "CustNum" ],
  eOrder: [ "CustNum", "Ordernum" ]
}
```

When merging with a single table reference, you might use the following array object:

```
[ "CustNum", "Ordernum" ]
```

Note: For any *key-fields* that have the `string` data type, the character values for these fields are compared to identify duplicates according to the value of the `caseSensitive` property on each affected table reference.

Note: After this method checks for any duplicate keys and completes adding record objects to JSDO memory from *merge-object*, and if you have set up automatic sorting using the `autoSort` property, all the record objects for the affected table references are sorted accordingly. If the sorting is done using sort fields, any `string` values in the specified sort fields are compared according to the value of the `caseSensitive` property.

A typical use for `addRecords( )` is to merge additional data returned by an invocation method without having to re-load JSDO memory with all the data from the `fill( )` method.

Example

Given a JSDO (`dataset`) that you fill with available records from the `eCustomer` and `eOrder` tables of an OpenEdge ProDataSet, you might retrieve a new `eOrder` record as the result of a `getNewOrder( )` invocation method on the JSDO and add the new record to JSDO memory as follows:

```
var dataset = progress.data.JSDO( "dsCustomerOrder" );
dataset.fill(); // Loads the JSDO with all available records from the ProDataSet resource

// Adds a new eOrder record restrieved from the service
var request = dataset.getNewOrder(null,false);
dataset.eOrder.addRecords( request.response, progress.data.JSDO.MODE_APPEND,
  [ "CustNum", "Ordernum" ],
  );
```

This code fragment adds the `eOrder` record for an existing `eCustomer` record specified by the `CustNum` property and a new order number specified by the `Ordernum` property of the single record object returned in `result.dsCustomerOrder.eOrder[0]`.

See also:

autoSort property on page 203, caseSensitive property on page 213, getId( ) method on page 242, fill( ) method on page 220, invocation method on page 262, saveChanges( ) method on page 314

afterAddCatalog event

Fires when the `addCatalog( )` method on the current `Session` object completes execution after it was called asynchronously.

Applies to:progress.data.Session class on page 136

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( login-session , result , error-object )
```

login-session

A reference to the `Session` object that fired the event.

result

A numeric constant that the `addCatalog( )` method returns when called synchronously. Possible constants include:

- **`progress.data.Session.SUCCESS`** — The specified Data Service Catalog loaded successfully.
- **`progress.data.Session.CATALOG_ALREADY_LOADED`** — The specified Data Service Catalog did not load because it is already loaded.
- **`progress.data.Session.AUTHENTICATION_FAILURE`** — The Catalog failed to load because of a user authentication error.
- **`progress.data.Session.GENERAL_FAILURE`** — The specified Catalog failed to load because of an error other than an authentication failure.

Note: This value can also be returned if invalid user credentials triggered a Catalog access timeout according to the value of the `addCatalog( )` method's *`parameter-object`*`.iOSBasicAuthTimeout` property.

For all other non-authentication errors, this event returns an `Error` object reference as *`error-object`*. For more detailed information about any response (successful or unsuccessful) returned from the web server, you can also check the XMLHttpRequest object (XHR) returned by the `lastSessionXHR` property.

If *`error-object`* is **not** `null`, *`result`* will be `null`.

error-object

A reference to any Error object that might have been thrown as *login-session* processed the addCatalog( ) method response. This value can be null. If it is **not** null, *result* will be null.

Note: If this object is thrown because of a Catalog access timeout triggered according to the value of the addCatalog( ) method's *parameter-object*.iOSBasicAuthTimeout property, the message property of the error object is set to "addCatalog timeout expired".

Note: These callback parameters provide the same information that is available after a synchronous invocation of addCatalog( ).

Application code can subscribe a callback for this event by invoking the subscribe( ) method on a Session object.

Example

The following code fragment subscribes the function, onAfterAddCatalog, to handle the afterAddCatalog event fired on the session, empSession, after the addCatalog( ) method is called asynchronously. The event callback checks for either expected success and failure return values, or a thrown Error object with an unknown error, and assembles an appropriate message to display in an alert box for each case:

```
var retValue;
empSession.subscribe('afterAddCatalog', onAfterAddCatalog);

retValue = empSession.addCatalog( {
    catalogURI : "http://testmach:8980/SportsMobileApp/static/OrderEntrySvc.json",
    async : true }  );
/* ( retValue is progress.data.Session.ASYNC_PENDING ) */


/* invoked by empSession when it processes the response from getting the Catalog
   from the web application */
function onAfterAddCatalog( pdsession, addCatalogResult, errorObject ) {
 var msg;

    if (addCatalogResult ===  progress.data.Session.LOGIN_AUTHENTICATION_FAILURE ) {
        alert("Add Employee Catalog failed: Authentication error");
    }
    if (addCatalogResult ===  progress.data.Session.GENERAL_FAILURE ) {
        alert("Add Employee Catalog failed: Non-authentication error");
    }
    else if (addCatalogResult ) { // Either SUCCESS or CATALOG_ALREADY_LOADED
        alert("Catalogs loaded.");
    }
    else { // errorObject likely returns other non-authentication error info
           if (errorObject) {
               msg = '\n' + errorObject.message;
           }
        alert("Unexpected addCatalog error: " + msg);
    }
};
```

See also:

addCatalog( ) method (Session class) on page 164, lastSessionXHR property on page 274, subscribe( ) method (Session class) on page 349

afterCreate event

Fires after the JSDO, by means of a `saveChanges( )` call following an `add( )` call, sends a request to create a record in the Data Object resource and receives a response to this request from the server.

This event fires after the response from a Create operation request (sent without using Submit) has been received, and fires after the response from a Submit operation request for each one of possibly multiple record creates has been received.

Note: A single Create operation request is sent for each record object that you have created in the JSDO with a single call to `saveChanges( )` or `saveChanges(false)`. A single Submit operation request for any and all created, updated, and deleted JSDO record objects is sent with a single call to `saveChanges(true)`.

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , record , success , request )
```

jsdo

A reference to the JSDO that invoked the create request. For more information, see the description of jsdo property on page 272 of the request object.

record

A reference to the table record upon which the create request acted. For more information, see the description of jsrecord property on page 272 of the request object.

success

A `boolean` that is `true` if the create request was successful. For more information, see the description of success property on page 350 of the request object.

request

A reference to the request object returned after the create request completes. For more information, see the description of request object on page 146.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance or one of its table references.

Example

The following code fragment subscribes the function, `onAfterCreate`, to handle the `afterCreate` event fired on the JSDO, `myjsdo`, created for an OpenEdge single-table resource, `ttCustomer`, where `newRecordData` is an object containing the field values to set in the new record:

```
/* subscribe to event */
myjsdo.subscribe( 'afterCreate', onAfterCreate );

/* some code that might add one or more
   records and save them on the server */
var jsrecord = myjsdo.add( newRecordData );
. . .

myjsdo.saveChanges();

function onAfterCreate ( jsdo , record , success , request ) {
    var myField,
        lenErrors,
        errors,
        errorType;
    if (success) {
        /* for example, get the values from the record for redisplay */
        myField = record.data.myField;
        . . .
    }
    else { /* Handle Create operation errors */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.DATA_ERROR:
                    errorType = "BI Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
            }
            /* Log error type and message for current error */
            console.log("ERROR: " + errorType + errors[idxError].error);
            if (errors[idxError].id) { /* Log info for error with record object */
                console.log("RECORD ID: " + errors[idxError].id);
                /* If the error record ID matches the operation
                   record ID, log the record data values */
                if  (errors[idxError].id == record.getId()) {
                    console.log ("Customer Record Fields:");
                    console log ("    CustNum = " + record.data.CustNum);
                    . . .
                }
            }
            /* Log additional HTTP text for web or other server errors */
            if (errors[idxError].responseText) {
                console.log("HTTP FULL TEXT: " + errors[idxError].responseText);
            }
        }
    } /* End Create operation errors */
};
```

See also:

add( ) method on page 158, beforeCreate event on page 206, record property on page 305, saveChanges( ) method on page 314, subscribe( ) method (JSDO class) on page 347, unsubscribe( ) method (JSDO class) on page 353

afterDelete event

Fires after the JSDO, by means of a `saveChanges( )` call following a `remove( )` call, sends a request to delete a record in the Data Object resource and receives a response to this request from the server.

This event fires after the response from a Delete operation request (sent without using Submit) has been received, and fires after the response from a Submit operation request for each one of possibly multiple record deletes has been received.

Note: A single Delete operation request is sent for each record object that you have deleted in the JSDO with a single call to `saveChanges( )` or `saveChanges(false)`. A single Submit operation request for any and all created, updated, and deleted JSDO record objects is sent with a single call to `saveChanges(true)`.

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , record , success , request )
```

jsdo

A reference to the JSDO that invoked the delete request. For more information, see the description of jsdo property on page 272 of the request object.

record

A reference to the table record upon which the delete request acted. For more information, see the description of jsrecord property on page 272 of the request object.

success

A `boolean` that is `true` if the delete request was successful. For more information, see the description of success property on page 350 of the request object.

request

A reference to the request object returned after the delete request completes. For more information, see the description of request object on page 146.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance or one of its table references.

Example

The following code fragment subscribes the function, `onAfterDelete`, to handle the `afterDelete` event fired on the JSDO, `myjsdo`, created for a single-table resource, `ttCustomer`, where `myid` is the known ID of a record to find and delete:

```
/* subscribe to event */
myjsdo.subscribe( 'afterDelete', onAfterDelete );

/* some code that might delete one or more
   records, then remove them on the server */
var jsrecord = myjsdo.findById(myid);
if (jsrecord) {jsrecord.remove();};
. . .

myjsdo.saveChanges();

function onAfterDelete ( jsdo , record , success , request ) {
    var myField,
        lenErrors,
        errors,
        errorType;
    if (success) {
        /* for example, get the values from the record that was
           deleted to display a confirmation message */
        myKeyField = record.data.myKeyField;
        . . .
    }
    else { /* Handle Delete operation errors */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.DATA_ERROR:
                    errorType = "BI Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
            }
            /* Log error type and message for current error */
            console.log("ERROR: " + errorType + errors[idxError].error);
            if (errors[idxError].id) { /* Log info for error with record object */
                console.log("RECORD ID: " + errors[idxError].id);
                /* If the error record ID matches the operation
                   record ID, log the record data values */
                if  (errors[idxError].id == record.getId()) {
                    console.log ("Customer Record Fields:");
                    console log ("    CustNum = " + record.data.CustNum);
                    . . .
                }
            }
            /* Log additional HTTP text for web or other server errors */
            if (errors[idxError].responseText) {
                console.log("HTTP FULL TEXT: " + errors[idxError].responseText);
            }
        }
    } /* End Delete operation errors */
};
```

See also:

beforeDelete event on page 207, remove( ) method on page 311, saveChanges( ) method on page 314, subscribe( ) method (JSDO class) on page 347, unsubscribe( ) method (JSDO class) on page 353

afterFill event

Fires on the JSDO after a call to the `fill( )` method executes and returns the results from the server to JSDO memory for a Read operation request on its Data Object resource.

Alias: `afterRead`

Applies to: progress.data.JSDO class on page 110

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , success , request )
```

jsdo

A reference to the JSDO that invoked the `fill( )` method to initialize JSDO memory with the data in its resource. For more information, see the description of the jsdo property on page 272 of the request object.

success

A `boolean` that is `true` if the Read operation on the resource was successful. For more information, see the description of the success property on page 350 of the request object.

request

A reference to the request object returned after the `fill( )` method completed execution either successfully or unsuccessfully. For more information, see the description of the request object on page 146.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance.

Example

The following code fragment subscribes the callback function, `onAfterFill`, to handle the `afterFill` event fired on the JSDO, `myjsdo`, after results are returned for a Read operation on a single-table resource, `ttCustomer`:

```
* subscribe to event */
myjsdo.subscribe( 'afterFill', onAfterFill );

myjsdo.fill();

function onAfterFill( jsdo , success , request ) {
    var lenErrors,
        errors,
        errorType;
    if (success) {
        /* for example, add code to display all records on a list */
        jsdo.foreach(function (jsrecord) {
            /* you can reference the fields as jsrecord.data.field */
        });
    }
    else {/* Handle Read operation errors */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
            }
            /* Log error type and message for current error */
            console.log("READ ERROR: " + errorType + errors[idxError].error);
            /* Log additional HTTP text for web or other server errors */
            if (errors[idxError].responseText) {
                console.log("HTTP FULL TEXT: " + errors[idxError].responseText);
            }
        }
    }

};
```

See also:

beforeFill event on page 208, fill( ) method on page 220, subscribe( ) method (JSDO class) on page 347, unsubscribe( ) method (JSDO class) on page 353

afterInvoke event

Fires after a custom invocation method is called asynchronously on a JSDO and a response to the Invoke operation request is received from the server.

Synchronous invocation method calls do not cause this event to fire.

Applies to: progress.data.JSDO class on page 110

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , success , request )
```

jsdo

A reference to the JSDO that invoked the method. For more information, see the description of jsdo property on page 272 of the request object.

success

A `boolean` that is `true` if the operation was successful. For more information, see the description of success property on page 350 of the request object.

request

A reference to the request object returned after the operation completes. For more information, see the description of request object on page 146.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance.

Note: To subscribe a handler for this event, the `subscribe( )` method requires that you pass, as a parameter, the name of the invocation method to which the event applies.

Example

The following code fragment subscribes the function, `onAfterInvokeGetOrderTotalAndStatus`, to handle the `afterInvoke` event fired on the JSDO, `dataSet`, for an invocation of the `getOrderTotalAndStatus( )` invocation method passed the parameters specified by `paramObject`:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );
dataSet.subscribe( 'afterInvoke', 'getOrderTotalAndStatus',
                   onAfterInvokeGetOrderTotalAndStatus );

dataSet.getOrderTotalAndStatus( paramObject );

function onAfterInvokeGetOrderTotalAndStatus( jsdo , success , request )
    if (success) {

        var response = request.response;
        var ordTotal = response._retVal;
        var ordStatus = response.pcStatus;
        /* process successful results */
        . . .

    }
    else {
        if (request.response && request.response._errors &&
            request.response._errors.length > 0) {

            var lenErrors = request.response._errors.length;
            for (var idxError=0; idxError < lenErrors; idxError++) {

                var errorEntry = request.response._errors[idxError];
                var errorMsg = errorEntry._errorMsg;
                var errorNum = errorEntry._errorNum;
                /* handle error */
                . . .

            }
        }
    }

};
```

See also:

beforeInvoke event on page 209, invocation method on page 262, subscribe( ) method (JSDO class) on page 347, unsubscribe( ) method (JSDO class) on page 353

afterLogin event

Fires when the `login( )` method on the current `Session` object completes execution after it was called asynchronously.

Applies to:progress.data.Session class on page 136

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( login-session , result , error-object )
```

login-session

A reference to the `Session` object that fired the event.

result

A numeric constant that the `login( )` method returns when called synchronously. Possible constants include:

- **`progress.data.Session.LOGIN_SUCCESS`** — JSDO login session started successfully.
- **`progress.data.Session.LOGIN_AUTHENTICATION_FAILURE`** — JSDO session login failed because of invalid user credentials.
- **`progress.data.Session.LOGIN_GENERAL_FAILURE`** — JSDO session login failed because of a non-authentication failure.

  Note: This value can also be returned if invalid user credentials triggered a login timeout according to the value of the `login( )` method's *`parameter-object`*`.iOSBasicAuthTimeout` property.

For all other errors, this event returns an `Error` object reference as *`error-object`*. You can also return the result for the most recent login attempt on *`login-session`* by reading its `loginResult` property. For a more specific status code returned in the HTTP response, you can check the value of its `loginHttpStatus` property. For more detailed information about any response (successful or unsuccessful) returned from the Web server, you can also check the XMLHttpRequest object (XHR) returned by its `lastSessionXHR` property.

If *`error-object`* is **not** `null`, *`result`* will be `null`.

error-object

A reference to any `Error` object that might have been thrown as *`login-session`* processed the `login( )` method response. This value can be `null`. If it is **not** `null`, *`result`* will be `null`.

Note: If this object is thrown because of a login timeout triggered according to the value of the `login( )` method's *`parameter-object`*`.iOSBasicAuthTimeout` property, the `message` property of the error object is set to `"login timeout expired"`.

Note: These callback parameters provide the same information that is available after a synchronous invocation of `login( )`.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a `Session` object.

Example

The following code fragment subscribes the function, `onAfterLogin`, to handle the `afterLogin` event fired on the session, `empSession`, after the `login( )` method is called asynchronously. The event callback checks for either an expected return value, an invalid return value (as part of a test), or a thrown `Error` object with an unknown error (passed as `errorObject` to the callback), then assembles an appropriate message to display in an alert box:

```
var retValue;
empSession.subscribe('afterLogin', onAfterLogin);

retValue = empSession.login( {
    serviceURI : "http://testmach:8980/SportsMobileApp",
    userName : uname,
    password : pw,
    async : true }  );
/* ( retValue is progress.data.Session.ASYNC_PENDING ) */

/* Invoked by empSession when it processes the login response from the web
   application */
function onAfterLogin( pdsession, result, errorObject ) {
    var msg;

    if ( result === null ) {
        msg = "Employee Login failed: Error attempting to call login()";
    }
    else if ( result === progress.data.Session.LOGIN_AUTHENTICATION_FAILURE ) {
            msg = "Employee Login failed: Authentication error";
    }
    else if ( result === progress.data.Session.LOGIN_GENERAL_FAILURE  ) {
        msg = "Employee Login failed: Unspecified error";
    }
    else  if ( result === progress.data.Session.LOGIN_SUCCESS  ) {
        msg = "Logged in successfully”;
    }
    else {
           if (errorObject) {
               msg = '\n' + errorObject.message;
           }
        msg = "TEST ERROR! UNEXPECTED login result: " + msg;
    }
    msg = msg +
              "\nloginResult: " + pdsession.loginResult +
              "\nloginHttpStatus: " +  pdsession.loginHttpStatus +
              "\nuserName: " +  pdsession.userName +
              "\nlastSessionXHR: " +  pdsession.lastSessionXHR;

    alert(msg);
}
```

See also:

lastSessionXHR property on page 274, login( ) method (Session class) on page 280, loginHttpStatus property on page 284, subscribe( ) method (Session class) on page 349

afterLogout event

Fires when the `logout( )` method on the current `Session` object completes execution after it was called asynchronously.

Applies to:progress.data.Session class on page 136

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( login-session , error-object )
```

login-session

A reference to the `Session` object that fired the event.

error-object

A reference to any `Error` object that might have been thrown as *`login-session`* processed the `logout( )` method response. This value can be `null`. For more detailed information about any response (successful or unsuccessful) returned from the web server, you can also check the XMLHttpRequest object (XHR) returned by its `lastSessionXHR` property.

Note: These callback parameters provide the same information that is available after a synchronous invocation of `logout( )`. Also, the `logout( )` method does not send a request to the web application if it is using Anonymous authentication. In this case, `logout( )` invoked asynchronously will nevertheless invoke any `afterLogout` event callback that has been subscribed when it is done executing.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a `Session` object.

Example

The following code fragment subscribes the function, `onAfterLogout`, to handle the `afterLogout` event fired on the session, `empSession`, after the `logout( )` method is called asynchronously. If an `Error` object is passed in, the event callback displays a message:

```
empSession.subscribe('afterLogout', onAfterLogout);

empSession.logout( { async : true }  );


/* Invoked by empSession when it finishes executing the logout operation */
function onAfterLogout( pdsession, errorObject ) {
    var msg;

    msg = errorObject ? '\n' + errorObject.message : '';
    if ( pdsession.lastSessionXHR === null ) {
        alert("logout succeeded");
        return;
    }
    alert("There was an error attempting log out." + msg);
}
```

See also:

lastSessionXHR property on page 274, logout( ) method (Session class) on page 288, subscribe( ) method (Session class) on page 349

afterSaveChanges event

Fires once for each call to the `saveChanges( )` method on a JSDO, after responses to all create, update, and delete record requests sent to a Data Object resource have been received from the server.

This event fires after all responses have been received from one or more Create, Update, and Delete (CUD) operation requests (sent without using Submit), and fires after the one response has been received from a single Submit operation request after its one or more record changes have completed.

Note: A single CUD operation request is sent for each record object that you have created, updated, or deleted (respectively) in the JSDO with a single call to `saveChanges( )` or `saveChanges(false)`. A single Submit operation request for any and all created, updated, and deleted JSDO record objects is sent with a single call to `saveChanges(true)`.

Applies to: progress.data.JSDO class on page 110

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , success , request )
```

jsdo

A reference to the JSDO that invoked the `saveChanges( )` method for one or more record changes on its resource. For more information, see the description of the jsdo property on page 272 of the request object.

success

A `boolean` that is `true` if all record changes invoked on the resource by `saveChanges( )` were successful. For more information, see the description of the success property on page 350 of the request object.

request

A reference to the request object returned after the `saveChanges( )` method completed execution and returned all results from its record changes on the server. For more information, see the description of the request object on page 146.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance.

Example

The following code fragment subscribes the function, `onAfterSaveChanges`, to handle the `afterSaveChanges` event fired on the JSDO, `myjsdo`, for changes saved on an OpenEdge single-table resource, `ttCustomer` without using a Submit operation:

```
/* subscribe to event */
myjsdo.subscribe( 'afterSaveChanges', onAfterSaveChanges );
/* some code that initiates multiple CRUD operations and
   sends them to the server */
var newrec = myjsdo.add();
. . .

var jsrecord = myjsdo.findById(myid);
if (jsrecord) {jsrecord.remove();};

myjsdo.saveChanges(); /* invoked without Submit */

function onAfterSaveChanges( jsdo , success , request ) {
    /* number of resource operations invoked by saveChanges() */
    var lenErrors,
        errors,
        errorType;
    if (success) {
        /* all resource operations invoked by saveChanges() succeeded */
        /* for example, redisplay records in list */
        jsdo.foreach( function(jsrecord) {
            /* reference the record/field as jsrecord.data.<fieldName> */
        });
    }
    else {
        /* handle all operation errors */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.DATA_ERROR:
                    errorType = "Server Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
                case default:
                    errorType = null; // Unexpected errorType value
                    break;
            }
            if (errorType) { /* log all error text
                console.log("ERROR: " + errorType + errors[idxError].error);
                if (errors[idxError].id) { /* error with record object */
                    console.log("RECORD ID: " + errors[idxError].id);
                    /* possibly log the data values for the record with this ID
                       from the matching request.batch.operations[index].jsrecord */
                }
                if (errors[idxError].responseText) {
                    console.log("HTTP FULL TEXT: "
                                + errors[idxError].responseText);
                }
            }
            else { /* unexpected errorType */
                console.log("UNEXPECTED ERROR TYPE: "
                            + errors[idxError].type);
            }
        }
    }
};
```

If successful, the example loops to display the data from every record in JSDO memory (not shown). If unsuccessful, the example loops through the array of error objects returned from the call to `saveChanges( )` (the result of calling `jsdo.ttCustomer.getErrors( )`) and logs annotated values for each error object returned, depending on its type (`errors[idxError].type`). If a record ID is included for a given error object, you can display the field values for the associated record (not shown) that you can return from `request.batch.operations[`*`index`*`].jsrecord`, where *`index`* identifies a returned request object for an operation in the `request.batch.operations` array and the error record ID matches `getId(request.batch.operations[`*`index`*`].jsrecord)`.

Note: This example takes advantage of the default setting of the JSDO `autoApplyChanges` property (`true`), which automatically accepts or rejects changes to JSDO memory based on whether all CUD operations were successful. Note also that for an unsuccessful result, the error information returned by `getErrors( )` is still available until the next invocation of `fill( )` or `saveChanges( )`.

See also:

beforeSaveChanges event on page 210, saveChanges( ) method on page 314, subscribe( ) method (JSDO class) on page 347, unsubscribe( ) method (JSDO class) on page 353

afterUpdate event

Fires after the JSDO, by means of a `saveChanges( )` call following an `assign( )` call, sends a request to update a record in the Data Object resource and receives a response to this request from the server.

This event fires after the response from an Update operation request (sent without using Submit) has been received, and fires after the response from a Submit operation request for each one of possibly multiple record updates has been received.

Note: A single Update operation request is sent for each record object that you have updated in the JSDO with a single call to `saveChanges( )` or `saveChanges(false)`. A single Submit operation request for any and all created, updated, and deleted JSDO record objects is sent with a single call to `saveChanges(true)`.

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , record , success , request )
```

`jsdo`

A reference to the JSDO that invoked the update request. For more information, see the description of jsdo property on page 272 of the request object.

`record`

A reference to the table record upon which the update request acted. For more information, see the description of jsrecord property on page 272 of the request object.

success

A `boolean` that is `true` if the update request was successful. For more information, see the description of success property on page 350 of the request object.

request

A reference to the request object returned after the update request completes. For more information, see the description of request object on page 146.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance or one of its table references.

Example

The following code fragment subscribes the function, `onAfterUpdate`, to handle the `afterUpdate` event fired on the JSDO, `myjsdo`, created for an OpenEdge single-table resource, `ttCustomer`, where `myid` is the known ID of a record to find and update:

```
/* subscribe to event */
myjsdo.subscribe( 'afterUpdate', onAfterUpdate );

/* some code that might update one or more
   records, then apply the updates on the server */
var jsrecord = myjsdo.findById(myid);
if (jsrecord) {jsrecord.assign( updatedDataObject );};
. . .

myjsdo.saveChanges();

function onAfterUpdate ( jsdo , record , success , request ) {
    var myField,
        lenErrors,
        errors,
        errorType;
    if (success) {
        /* for example, get the values updated by the server from the record
           to redisplay */
        var newValue = record.data.myField;
        . . .
    }
    else { /* Handle Update operation errors */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.DATA_ERROR:
                    errorType = "BI Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
            }
            /* Log error type and message for current error */
            console.log("ERROR: " + errorType + errors[idxError].error);
            if (errors[idxError].id) { /* Log info for error with record object */
                console.log("RECORD ID: " + errors[idxError].id);
                /* If the error record ID matches the operation
                   record ID, log the record data values */
                if  (errors[idxError].id == record.getId()) {
                    console.log ("Customer Record Fields:");
                    console log ("    CustNum = " + record.data.CustNum);
                    . . .
                }
            }
            /* Log additional HTTP text for web or other server errors */
            if (errors[idxError].responseText) {
                console.log("HTTP FULL TEXT: " + errors[idxError].responseText);
            }
        }
    } /* End Update operation errors */
};
```

See also:

assign( ) method (JSDO class) on page 198, beforeUpdate event on page 211, saveChanges( ) method on page 314, subscribe( ) method (JSDO class) on page 347, unsubscribe( ) method (JSDO class) on page 353

assign( ) method (JSDO class)

Updates field values for the specified `JSRecord` object in JSDO memory.

The specified record object can be either the working record of a JSDO table reference or any record provided by a `JSRecord` object.

To synchronize the change on the server, call the `saveChanges( )` method.

Alias: `update( )`

Return type: `boolean`

Applies to: progress.data.JSRecord class on page 133, progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Syntax

```
jsrecord-ref.assign ( update-object )
jsdo-ref.assign ( update-object )
jsdo-ref.table-ref.assign ( update-object )
```

jsrecord-ref

A reference to a `JSRecord` object for a table record in JSDO memory.

You can obtain a `JSRecord` object by:

- Invoking a JSDO method that returns record objects from a JSDO table reference (`find( )`, `findById( )`, or `foreach( )`)
- Accessing the `record` property on a JSDO table reference that already has a working record.
- Accessing the *`record`* parameter passed to the callback of a JSDO `afterCreate`, `afterDelete`, or `afterDelete` event.
- Accessing each record object provided by the `jsrecords` property on the *`request`* object parameter passed to the callback of a JSDO `afterSaveChanges` event, or passed to the callback of any Promise object returned from the `saveChanges( )` method. The `jsrecords` property is only available on completion of a Submit operation (`saveChanges(true)`) on a resource that supports before-imaging, and **not** if the resource supports before-imaging without Submit.

jsdo-ref

A reference to the JSDO. You can call the method on *`jsdo-ref`* if the JSDO has only a single table reference, and that table reference has a working record.

table-ref

A table reference on the JSDO that has a working record.

update-object

Passes in the data to update the specified record object in JSDO memory. Each property of the object has the name of a table field and the value to set for that field in the specified record. Any table fields without corresponding properties in *update-object* remain unchanged in the record.

Note: After this method updates the specified record object, and if you have set up automatic sorting using the `autoSort` property, all the record objects for the affected table reference are sorted accordingly. If the sorting is done using sort fields, any `string` fields are compared according to the value of the `caseSensitive` property.

Example

The following code fragment shows a jQuery event defined on a save button to save the current field values for a customer detail form to the corresponding `eCustomer` record in JSDO memory:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );

$('#btnSave').bind('click', function(event) {
  var jsrecord = dataSet.eCustomer.findById($('#custdetail #id').val());
  if (jsrecord) {jsrecord.assign(update-object);};
  dataSet.saveChanges();
});
```

The form has been displayed with previous values of the same record. When the button is clicked, the event handler uses the `findById( )` method to find the original record with the matching internal record ID (`jsrecord`) and invokes the `assign( )` method on `jsrecord` with an object parameter to update the fields in `eCustomer` with any new values entered into the form.

See also:

autoSort property on page 203, caseSensitive property on page 213, getSchema( ) method on page 247, saveChanges( ) method on page 314

assign( ) method (UIHelper class)

Updates a specified working record in JSDO memory with the values currently displayed on the detail page form.

Return type: `boolean`

Applies to: progress.ui.UIHelper class on page 142, table reference property (UIHelper class) on page 353

Syntax

```
uihelper-ref.assign ( )
uihelper-ref.table-ref.assign ( )
```

uihelper-ref

A reference to a `UIHelper` instance. You can call the method on *`uihelper-ref`* if the JSDO associated with the instance has only a single table reference, and that table reference has a working record.

table-ref

A table reference that has a working on the JSDO associated with the `UIHelper` instance.

Note: After this method updates the working record in JSDO memory, and if you have set up automatic sorting using the `autoSort` property, all the record objects for the affected table reference are sorted accordingly. If the sorting is done using sort fields, any `string` fields are compared according to the value of the `caseSensitive` property.

See also:

autoSort property on page 203, caseSensitive property on page 213, getFormFields( ) method on page 240

async property

A `boolean` that indicates, if set to `true`, that the Data Object resource operation was executed asynchronously in the mobile app.

Data type: `boolean`

Access: Read-only

Applies to: request object on page 146

The `async` property is available only for the following JSDO events or in the request object returned to a jQuery Promise callback:

- `afterCreate`
- `afterDelete`
- `afterFill`
- `afterInvoke`
- `afterUpdate`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe or register to JSDO events, any returned jQuery Promises, and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO. This object is also returned as the value of any JSDO invocation method that you execute synchronously.

See also:

add( ) method on page 158, remove( ) method on page 311, fill( ) method on page 220, invocation method on page 262, invoke( ) method on page 264, saveChanges( ) method on page 314

authenticationModel property (JSDOSession class)

Returns a `string` constant that was passed as an option to the object's class constructor, and specifies the authentication model that the current `JSDOSession` object requires to start a JSDO login session in the web application for which the `JSDOSession` object was also created.

Data type: `string`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 124

Values that can be returned include:

- **`progress.data.Session.AUTH_TYPE_ANON`** — The web application supports Anonymous access. No authentication is required. This is the default value if none is passed to the `JSDOSession` constructor.
- **`progress.data.Session.AUTH_TYPE_BASIC`** — The web application supports HTTP Basic authentication and requires a valid username and password. To have the `JSDOSession` object manage access to the web application's resources for you, you need to pass these credentials in a call to the `JSDOSession` object's `login( )` method. Typically, you would require the user to enter their credentials into a login dialog provided by your mobile app, either using a form of your own design or using a template provided by Progress Software.
- **`progress.data.Session.AUTH_TYPE_FORM`** — The web application uses HTTP Form-based authentication. Like HTTP Basic, Form-based authentication requires user credentials for access to protected resources; the difference is that the web application itself sends a form to the client to get the credentials. However, when you have the `JSDOSession` object manage access to the web application's resources, you handle Form-based authentication the same way that you handle Basic—get the user's credentials yourself and pass them to the `login( )` method. The `JSDOSession` intercepts the form sent by the web application and handles the authentication without that form being displayed.

If the web application requires authentication, you must set this value correctly in the `JSDOSession` constructor to ensure that users can log in.

See also:

The constructor description for the progress.data.JSDOSession class on page 124, login( ) method (JSDOSession class) on page 275

authenticationModel property (Session class)

Note: Deprecated in Progress Data Objects Version 4.4 and later. Please use the authenticationModel property (JSDOSession class) on page 201 instead.

A `string` constant that you can set to specify the authentication model that a given web application requires for a mobile app to start a JSDO login session for this `Session` object.

Data type: `string`

Access: Readable/Writable

Applies to: progress.data.Session class on page 136

Valid values are:

- **`progress.data.Session.AUTH_TYPE_ANON`** — The web application supports Anonymous access. No authentication is required. This is the default value if you do not set it.
- **`progress.data.Session.AUTH_TYPE_BASIC`** — The web application supports HTTP Basic authentication and requires a valid username and password. To have the `Session` object manage access to the web application's resources for you, you need to pass these credentials in a call to the `Session` object's `login( )` method. Typically, you would require the user to enter their credentials into a login dialog provided by your mobile app, either using a form of your own design or using a template provided by Progress Software.
- **`progress.data.Session.AUTH_TYPE_FORM`** — The web application uses HTTP Form-based authentication. Like HTTP Basic, Form-based authentication requires user credentials for access to protected resources; the difference is that the web application itself sends a form to the client to get the credentials. However, when you have the `Session` object manage access to the web application's resources, you handle Form-based authentication the same way that you handle Basic—get the user's credentials yourself and pass them to the `login( )` method. The `Session` intercepts the form sent by the web application and handles the authentication without that form being displayed.

If the web application requires authentication, you must set this value correctly prior to invoking the `login( )` method on this `Session` object in order to ensure that users can log in.

For more information on these authentication models and how to configure them for a web application, see the sections on configuring web server authentication models in the server documentation for your Data Object Service.

See also:

login( ) method (Session class) on page 280

autoApplyChanges property

A `boolean` on a JSDO that indicates if the JSDO automatically accepts or rejects changes to JSDO memory when you call the `saveChanges( )` method.

When set to `true`, and after you have invoked the `saveChanges( )` method, the JSDO automatically accepts all changes to JSDO memory that are successfully applied for a given resource operation on the server, and rejects all changes in JSDO memory that are associated with the same resource operation that are completed with an error.

The default setting is `true`.

You can set this property both during JSDO instantiation and on an existing JSDO.

Data type: `boolean`

Access: Readable/Writable

Applies to: progress.data.JSDO class on page 110

When set to `false`, you must invoke one of the following methods at the appropriate time to accept or reject the changes in JSDO memory:

- `acceptChanges( )`
- `acceptRowChanges( )`
- `rejectChanges( )`
- `rejectRowChanges( )`

You typically invoke one of these methods either in the appropriate event handler for a JSDO event or in the appropriate Promise method associated with execution of the `saveChanges( )` method. For information on detecting successful and unsuccessful resource operations, where you might invoke these methods, see the description of the `saveChanges( )` method.

Example

The following code fragment sets the property both when the JSDO is instantiated and after it is instantiated:

```
var jsdoCustomers = new progress.data.JSDO( { autoApplyChanges : false } );
. . .
jsdoCustomers.autoApplyChanges = true;
```

See also:

acceptChanges( ) method on page 153, acceptRowChanges( ) method on page 156, rejectChanges( ) method on page 305, rejectRowChanges( ) method on page 308, saveChanges( ) method on page 314

autoSort property

A `boolean` on a JSDO and its table references that indicates if record objects are sorted automatically on the affected table references in JSDO memory at the completion of a supported JSDO operation.

When set to `true`, and after you have specified a sorting method for each affected table reference, record objects are sorted after the JSDO operation completes its update of JSDO memory. When set to `false`, or if no sorting method is specified for a given table reference, no automatic sorting occurs after the JSDO operation completes. The default setting is `true` for all table references of a JSDO.

Data type: `boolean`

Access: Readable/Writable

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

When set on a JSDO, the property setting affects the sorting of record objects for all table references in the JSDO. When set on a single table reference, the property setting affects the sorting of record objects only for the specified table reference. For example, to set this property to `true` on only a single table reference in the JSDO:

1. Set the value on the JSDO to `false`, which sets `false` on all its table references.
2. Set the value on the selected table reference to `true`, which sets `true` on only the this one table reference.

In order to activate automatic sorting for an affected table reference, you must invoke one of the following JSDO methods to specify a sorting method for the table reference:

- **`setSortFields( )`** — Identifies the *sort fields* to use in the record objects and whether each field is sorted in ascending or descending order according to its data type. Any `string` fields specified for a table reference are sorted using letter case according to the setting of the `caseSensitive` property (`false` by default).

Note: Changing the value of the `caseSensitive` property triggers an automatic sort if the `autoSort` property is also set to `true` for the affected table reference.

- **`setSortFn( )`** — Identifies a *sort function* that compares two record objects according to the criteria you specify and returns a value that indicates if one record sorts later than the other in the sort order, or if the two records sort at the same position in the sort order. The `caseSensitive` property setting has no effect on the operation of the specified sort function unless you choose to involve the setting of this property in your criteria for comparison.

If you specify both sort fields and a sort function to sort the record objects for a table reference, the sort function takes precedence. You can also call the `setSortFields( )` and `setSortFn( )` functions to clear one or both settings of the sort fields and sort function. However, at least one setting must be active for automatic sorting to occur on a table reference.

The following supported JSDO operations trigger automatic sorting on any affected table references before they complete their updates to JSDO memory:

- **Invoking the `add( )` method** — Sorts the record objects of the affected table reference.
- **Invoking the `addRecords( )` method** — Sorts the record objects of either the single affected table reference or all affected table references in the JSDO. (Unaffected table references do not participate in the sort, including those for which `autoSort` is `false`, those for which no sort fields or sort function are set, or those other than the single JSDO table reference on which `addRecords( )` is called, if it is called only on a single table reference.)
- **Invoking the `addLocalRecords( )` method** — Sorts the record objects of all affected table references in the JSDO. (Unaffected table references do not participate in the sort, including those for which `autoSort` is `false` and those for which no sort fields or sort function are set.)
- **Invoking the `assign( )` method (JSDO or UIHelper class)** — Sorts the record objects of the affected table reference.
- **Assigning a value to a field reference directly on the working record of a table reference (*`jsdo-ref.table-ref.field-ref = value`*)** — Sorts the record objects of the affected table reference.

Note: Assignment to a field referenced on the `data` property **never** triggers automatic sorting (for example, *`jsdo-ref.table-ref`*`.data.`*`field-ref = value`*)

- **Changing the value of the `caseSensitive` property** — Sorts the record objects of the affected table reference, or of all affected table references if the property value is changed on the JSDO.
- **Invoking either the `acceptChanges( )` or `rejectChanges( )` method** — Sorts the record objects of all affected table references in the JSDO. (Unaffected table references do not participate in the sort, including any table references for which `autoSort` is `false`, or for which no sort fields or sort function are set.)
- **Invoking either the `acceptRowChanges( )` or `rejectRowChanges( )` method** — Sorts the record objects of the affected table reference.
- **Invoking the `fill( )` method** — Sorts the record objects of all affected table references in the JSDO. (Unaffected table references do not participate in the sort, including any table references for which `autoSort` is `false`, or for which no sort fields or sort function are set.)

Note: Invoking the `remove( )` method does not trigger an automatic sort and has no effect on any existing sort order established for the table reference. However, if there is a sort order that depends on the presence or absence of the record object you are removing, and you want to establish the appropriate sort order when this record object is absent, you must manually sort the remaining record objects using the `sort( )` method by passing it the same sort function that you used to establish the sort order when this record object was present.

Caution: Because automatic sorting executes in JavaScript on the client side, sorting a large set of record objects can take a significant amount of time and make the UI appear to be locked. You might set a wait or progress indicator just prior to any action that can sort a large record set to alert the user that the app is working.

Example

In the following code fragment, automatic local sorting is turned off for all table references of the `dsCustOrds` JSDO by setting its `autoSort` property to `false`. Automatic sorting is then turned on for the `eCustomer` table reference of the JSDO by setting its `autoSort` value to `true` and using the `setSortFields( )` method to set its `Name` field as the single, descending sort field:

```
dsCustOrds = new progress.data.JSDO( { name: 'dsCustomerOrders' });
dsCustOrds.autoSort = false.
dsCustOrds.eCustomer.autoSort = true.
dsCustOrds.eCustomer.setSortFields( "Name:DESC" );
dsCustOrds.fill();
. . .
```

When the `fill( )` method executes on the JSDO, all the referenced tables are loaded from the AppServer into JSDO memory with their record objects already sorted in case-insensitive, primary key order (by default). The record objects for `eCustomer` are then sorted locally in case-insensitive, descending order of the `Name` field.

See also:

acceptChanges( ) method on page 153, acceptRowChanges( ) method on page 156, add( ) method on page 158, addRecords( ) method on page 175, assign( ) method (JSDO class) on page 198, assign( ) method (UIHelper class) on page 199, caseSensitive property on page 213, fill( ) method on page 220, rejectChanges( ) method on page 305, rejectRowChanges( ) method on page 308, setSortFields( ) method on page 339, setSortFn( ) method on page 341, sort( ) method on page 344, table reference property (JSDO class) on page 351

batch property

A reference to an object with a property named `operations`, which is an array containing the request objects for each of the one or more record-change operations on a resource performed in response to calling the JSDO `saveChanges( )` method without using Submit (either with an empty parameter list or with the single parameter value of `false`).

Data type: `Object`

Access: Read-only

Applies to: request object on page 146

In addition to the properties documented for the request object, each request object returned in the `operations` property array also contains an `operation` property with a `number` value that indicates the record-change operation associated with the request object:

- `1` — Create
- `3` — Update
- `4` — Delete

The `batch` property is available only for the following JSDO events or in the request object returned to a Promise object callback, and **only** after calling `saveChanges( )` either with an empty parameter list or with the single parameter value of `false` to invoke individual Data Object Create, Update, or Delete (CUD) operations:

- `afterSaveChanges`
- `beforeSaveChanges`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe both to JSDO events and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO.

See also:

saveChanges( ) method on page 314, jsrecords property on page 273

beforeCreate event

Fires before the JSDO, by means of a `saveChanges( )` call following an `add( )` call, sends a request to create a record in the Data Object resource on the server.

This event fires before each Create operation request is sent without using Submit, and fires before each one of possibly multiple record creates are sent with a single Submit operation request.

Note: A single Create operation request is sent for each record object that you have created in the JSDO with a single call to `saveChanges( )` or `saveChanges(false)`. A single Submit operation request for any and all created, updated, and deleted JSDO record objects is sent with a single call to `saveChanges(true)`.

Applies to:progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , record , request )
```

jsdo

A reference to the JSDO that is invoking the create request. For more information, see the description of jsdo property on page 272 of the request object.

record

A reference to the table record upon which the create request is about to act. For more information, see the description of jsrecord property on page 272 of the request object.

request

A reference to the request object before the create request is sent. For more information, see the description of request object on page 146.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance or one of its table references.

Example

The following code fragment subscribes the function, `onBeforeCreate`, to handle the `beforeCreate` event fired on the JSDO, `myjsdo`, created for a single-table resource, by assigning data to the newly created record before sending it to the server:

```
/* subscribe to event */
myjsdo.subscribe( 'beforeCreate', onBeforeCreate );

/* some code that might add one or more
   records and save them on the server */
var jsrecord = myjsdo.add();

. . .

myjsdo.saveChanges();

function onBeforeCreate( jsdo , record , request ) {
    /* for example, here you might update data in the record
       before it is sent to the server to be created */
    record.assign( { myField1 = myvalue, myField2 = myvalue2 } );
};
```

See also:

add( ) method on page 158, afterCreate event on page 181, record property on page 305, saveChanges( ) method on page 314, subscribe( ) method (JSDO class) on page 347, unsubscribe( ) method (JSDO class) on page 353

beforeDelete event

Fires before the JSDO, by means of a `saveChanges( )` call following a `remove( )` call, sends a request to delete a record in the Data Object resource on the server.

This event fires before each Delete operation request is sent without using Submit, and fires before each one of possibly multiple record deletes are sent with a single Submit operation request.

Note: A single Delete operation request is sent for each record object that you have deleted in the JSDO with a single call to `saveChanges( )` or `saveChanges(false)`. A single Submit operation request for any and all created, updated, and deleted JSDO record objects is sent with a single call to `saveChanges(true)`.

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , record , request )
```

jsdo

A reference to the JSDO that is invoking the delete request. For more information, see the description of jsdo property on page 272 of the request object.

record

A reference to the table record upon which the delete request is about to act. For more information, see the description of jsrecord property on page 272 of the request object.

request

A reference to the request object before the delete request is sent. For more information, see the description of request object on page 146.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance or one of its table references.

Example

The following code fragment subscribes the function, `onBeforeDelete`, to handle the `beforeDelete` event fired on the JSDO, `myjsdo`, created for a single-table resource, where `myid` is the known ID of a record to find and delete:

```
/* subscribe to event */
myjsdo.subscribe( 'beforeDelete', onBeforeDelete );

/* some code that might delete one or more
   records and remove them on the server */
var jsrecord = myjsdo.findById( myid );
if (jsrecord) {jsrecord.remove();};

. . .

myjsdo.saveChanges();

function onBeforeDelete( jsdo , record , request ) {
    /* code to execute before sending delete request to the server */

};
```

See also:

afterDelete event on page 183, remove( ) method on page 311, saveChanges( ) method on page 314, subscribe( ) method (JSDO class) on page 347, unsubscribe( ) method (JSDO class) on page 353

beforeFill event

Fires on the JSDO before a call to the `fill( )` method executes and sends a Read operation request to its Data Object resource.

Alias: `beforeRead`

Applies to: progress.data.JSDO class on page 110

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , request )
```

jsdo

A reference to the JSDO that is invoking the Read operation on the resource. For more information, see the description of the jsdo property on page 272 of the request object.

request

A reference to the request object before the Read operation request is sent to the resource. For more information, see the description of the request object on page 146.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance.

Example

The following code fragment subscribes the callback function, `onBeforeFill`, to handle the `beforeFill` event fired on the JSDO, `myjsdo`:

```
/* subscribe to event */
myjsdo.subscribe( 'beforeFill', onBeforeFill );

myjsdo.fill();

function onBeforeFill ( jsdo , request ) {
    /* for example, do any preparation to receive data from the server */
};
```

See also:

afterFill event on page 185, fill( ) method on page 220, subscribe( ) method (JSDO class) on page 347, unsubscribe( ) method (JSDO class) on page 353

beforeInvoke event

Fires when a custom invocation method is called asynchronously on a JSDO before the request for the Invoke operation is sent to the server.

Synchronous invocation method calls do not cause this event to fire.

Applies to: progress.data.JSDO class on page 110

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , request )
```

jsdo

A reference to the JSDO that is invoking the method. For more information, see the description of jsdo property on page 272 of the request object.

request

A reference to the request object returned before the operation begins. For more information, see the description of request object on page 146.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance.

Note: To subscribe a handler for this event, the `subscribe( )` method requires that you pass, as a parameter, the name of the invocation method to which the event applies.

Example

The following code fragment subscribes the function, `onBeforeInvokeGetOrderTotalAndStatus`, to handle the `beforeInvoke` event fired on the JSDO, `dataSet`, for an invocation of the `getOrderTotalAndStatus( )` invocation method passed the parameters specified by `paramObject`:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );
dataSet.subscribe( 'beforeInvoke', 'getOrderTotalAndStatus',
                   onAfterInvokeGetOrderTotalAndStatus );

dataSet.getOrderTotalAndStatus( paramObject );

function onAfterInvokeGetOrderTotalAndStatus ( jsdo , request ) {
    /* code to execute before sending request to the server */
};
```

See also:

afterInvoke event on page 186, invocation method on page 262, subscribe( ) method (JSDO class) on page 347, unsubscribe( ) method (JSDO class) on page 353

beforeSaveChanges event

Fires once on the JSDO before a call to the `saveChanges( )` method sends the first request to create, update, or delete a record in its Data Object resource on the server.

This event fires before the first of possibly multiple Create, Update, and Delete operation requests are sent without using Submit, and fires before the first one of possibly multiple record changes are sent with a single Submit operation request.

Note: A single Create, Update, or Delete operation request is sent for each record object that you have created, updated, or deleted (respectively) in the JSDO with a single call to `saveChanges( )` or `saveChanges(false)`. A single Submit operation request for any and all created, updated, and deleted JSDO record objects is sent with a single call to `saveChanges(true)`.

Applies to: progress.data.JSDO class on page 110

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , request )
```

jsdo

A reference to the JSDO that is about to invoke the `saveChanges( )` method for one or more record changes on its resource. For more information, see the description of jsdo property on page 272 of the request object.

request

A reference to the request object before the first record-change request is sent to the resource. For more information, see the description of request object on page 146.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance.

Example

The following code fragment subscribes the function, `onBeforeSaveChanges`, to handle the `beforeSaveChanges` event fired on the JSDO, `myjsdo`, where `myid` is the known ID of a record to find and process for resource operations being sent to the server:

```
/* subscribe to event */
myjsdo.subscribe( 'beforeSaveChanges', onBeforeSaveChanges );

/* some code that initiates multiple CUD requests and
   sends them to the server */
var newrec = myjsdo.add();
var jsrecord = myjsdo.findById(myid);
if (jsrecord) {jsrecord.remove();};

. . .

myjsdo.saveChanges();

function onBeforeSaveChanges ( jsdo , request ) {
    /* code to execute before sending first (or only) request to the server */
};
```

See also:

afterSaveChanges event on page 192, saveChanges( ) method on page 314, subscribe( ) method (JSDO class) on page 347, unsubscribe( ) method (JSDO class) on page 353

beforeUpdate event

Fires before the JSDO, by means of a `saveChanges( )` call following an `assign( )` call, sends a request to update a record in the Data Object resource on the server.

This event fires before each Update operation request is sent without using Submit, and fires before each one of possibly multiple record updates are sent with a single Submit operation request.

Note: A single Update operation request is sent for each record object that you have updated in the JSDO with a single call to `saveChanges( )` or `saveChanges(false)`. A single Submit operation request for any and all created, updated, and deleted JSDO record objects is sent with a single call to `saveChanges(true)`.

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , record , request )
```

jsdo

A reference to the JSDO that is invoking the update request. For more information, see the description of jsdo property on page 272 of the request object.

record

A reference to the table record upon which the update request is about to act. For more information, see the description of jsrecord property on page 272 of the request object.

request

A reference to the request object before the update request is sent. For more information, see the description of request object on page 146.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance or one of its table references.

Example

The following code fragment subscribes the function, `onBeforeUpdate`, to handle the `beforeUpdate` event fired on the JSDO, `myjsdo`, created for a single-table resource, where `myid` is the known ID of a record to find and update. In this case, the `onBeforeUpdate` event callback assigns additional data to the updated record before sending it to the server:

```
/* subscribe to event */
myjsdo.subscribe( 'beforeUpdate', onBeforeUpdate );

/* some code that might update one or more
   records and save them on the server */
var jsrecord = myjsdo.findById(myid);
if (jsrecord) {jsrecord.assign( { myField1 = myvalue, myField2 = myvalue2 } );};

. . .

myjsdo.saveChanges();

function onBeforeUpdate( jsdo , record , request ) {
    /* for example, here you might update data in the record
       further before it is sent to the server */
    record.assign( { myField4 = myvalue4, myField5 = myvalue5 } );
};
```

See also:

assign( ) method (JSDO class) on page 198, afterUpdate event on page 195, saveChanges( ) method on page 314, subscribe( ) method (JSDO class) on page 347, unsubscribe( ) method (JSDO class) on page 353

caseSensitive property

A `boolean` on a JSDO and its table references that indicates if `string` field comparisons performed by supported JSDO operations are case sensitive or case-insensitive for the affected table references in JSDO memory.

When set to `true`, all supported comparisons on `string` fields for an affected table reference are case sensitive. When set to `false`, all supported comparisons on `string` fields for an affected table reference are case insensitive. The default setting is `false` for all table references of a JSDO.

Note: This default setting (case insensitive) matches the default setting for letter case comparison in OpenEdge Data Objects (ABL).

Data type: `boolean`

Access: Readable/Writable

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

When set on a JSDO, the property setting affects all table references in the JSDO. When set on a single table reference, the property setting affects only the specified table reference. For example, to set this property to `true` on only a single table reference in the JSDO:

1. Set the value on the JSDO to `false`, which sets `false` on all its table references.
2. Set the value on the selected table reference to `true`, which sets `true` on only the one table reference.

The JSDO operations that follow this property setting in `string` field comparisons include:

- Sorting record objects in JSDO memory, including automatic sorting using sort fields that you specify using the `autoSort` property and the `setSortFields( )` method, and manual sorting using specified sort fields that you perform using the `sort( )` method

  Note: Changing the value of this property triggers an automatic sort if the `autoSort` property is also set to `true` for the affected table reference.

- Merging record objects into JSDO memory for all merge modes that perform record field comparisons during execution of the `addRecords( )` method

Note: Any default `string` field comparisons that you might do in JavaScript within the callback functions that you specify for other JSDO methods and events are always case sensitive according to JavaScript rules and ignore this property setting.

Note: To conform to Unicode default letter case mapping, the JSDO support for case-insensitive `string`-field comparison and sorting relies on the `toUpperCase( )` JavaScript function instead of the `toLocaleUpperCase( )` JavaScript function. The latter function uses the locale letter case mapping, which might be different from the default letter case mapping in Unicode.

Note: Unlike character string comparisons in OpenEdge ABL, all JSDO-supported `string` field comparisons **include** trailing spaces and **ignore** any OpenEdge-specified collation tables.

Example

In the following code fragment, automatic local sorting is set up for the `eCustomer` table reference on a JSDO created for an OpenEdge resource (`dsCustOrds`), with its `Name` field as the single descending sort field. All other table references on `dsCustOrds` have no automatic local sorting set up by default. Because OpenEdge sorting on `string` fields is case-insensitive by default, the fragment makes the local sort on the `Name` field case sensitive by setting `caseSensitive` on `eCustomer` to `true`:

```
dsCustOrds = new progress.data.JSDO( { name: 'dsCustomerOrders' } );

dsCustOrds.autoSort = false;
dsCustOrds.eCustomer.autoSort = true;
dsCustOrds.eCustomer.setSortFields( "Name:descending" );
dsCustOrds.eCustomer.caseSensitive = true;

dsCustOrds.fill();
. . .
```

When the `fill( )` method executes on the JSDO, after all the referenced tables are loaded from the server, with their record objects already sorted in case-insensitive, primary key order (by default), the record objects for `eCustomer` are then sorted locally in case-sensitive, descending order by the `Name` field.

See also:

addRecords( ) method on page 175, autoSort property on page 203, setSortFields( ) method on page 339, sort( ) method on page 344

catalogURIs property

Returns the list of URIs successfully used to load Data Service Catalogs into the current `JSDOSession` or `Session` object.

Data type: `string` array

Access: Read-only

Applies to: progress.data.JSDOSession class on page 124, progress.data.Session class on page 136

This list includes the URI for each Data Service Catalog loaded using the `addCatalog( )` method on the current `JSDOSession` or `Session` object. To return a corresponding list of Data Object Services for which the Data Service Catalogs are loaded, read the `services` property.

See also:

addCatalog( ) method (JSDOSession class) on page 159, addCatalog( ) method (Session class) on page 164, services property on page 331

clearItems( ) method

Clears the items from a list view associated with a single table reference.

Return type: `null`

Applies to: progress.ui.UIHelper class on page 142, table reference property (UIHelper class) on page 353

Syntax

```
uihelper-ref.clearItems ( )
uihelper-ref.table-ref.clearItems ( )
```

uihelper-ref

A reference to a `UIHelper` instance. You can call the method on *uihelper-ref* if the JSDO associated with the instance has only a single table reference.

table-ref

A table reference on the JSDO associated with the `UIHelper` instance.

See also:

addItem( ) method on page 168, setListView( ) method on page 336, showListView( ) method on page 343

clientContextId property

The value of the most recent client context identifier (CCID) that the current `JSDOSession` or `Session` object has found in the `X-CLIENT-CONTEXT-ID` HTTP header of a server response message.

If none has yet been found, the value is `null`.

Data type: `string`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 124, progress.data.Session class on page 136

The `JSDOSession` or `Session` object automatically detects, stores, and returns the CCID sent by any appropriately configured web application for which it has started a JSDO login session.

Note: You can set up an OpenEdge web application to send a CCID when configuring the web application to use SSO. For more information, depending on the OpenEdge application server you use, see the sections on enabling SSO for a web application in the administration documentation for the OpenEdge AppServer or the Progress Application Server for OpenEdge.

Note: For an OpenEdge resource, this CCID is the same as the value of the `ClientContextId` property on the ABL `Progress.Lang.OERequestInfo` object that is passed from an application server client (in this case, the web application hosting the Data Object Service) to the server process that is executing a Data Object Service request. You can access this `OERequestInfo` object from the resource Business Entity using the ABL `CURRENT-REQUEST-INFO` attribute on the ABL `SESSION` system handle. This CCID value is also available as the ABL `SESSION-ID` attribute of the single sign-on (SSO) client-principal handle returned by the `GetClientPrincipal( )` method of the same ABL `OERequestInfo` class-based object.

Note: For a Rollbase resource, this CCID is created and managed internally by the Rollbase server.

See also:

login( ) method (JSDOSession class) on page 275, login( ) method (Session class) on page 280

connected property

Returns a `boolean` that indicates the most recent online status of the current `JSDOSession` or `Session` object when it last determined if the web application it manages was available.

If the property value is `true`, the object most recently determined that the session is connected and logged in to its web application. If its value is `false`, the session was last found to be disconnected. The default value is `false`.

Note:

Because of the dynamics of any network environment, the value of this property might not reflect the current status of the object's connection to its web application. You can therefore invoke the object's `ping( )` method (either explicitly or automatically by setting the value of its `pingInterval` property) to update the object's most recent online status.

Data type: `boolean`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 124, progress.data.Session class on page 136

The most recent session online status determination might be identified from any of the following:

- A successful result of the `JSDOSession` or `Session` object executing its `login( )` method, which sets the property to `true`. Prior to calling `login( )` for the first time, the value of this property is `false`.
- A successful result of the `JSDOSession` or `Session` object executing its `logout( )` or `invalidate( )` method, which sets the property to `false`.
- The `JSDOSession` or `Session` object receiving an `offline` or `online` event from its window object.
- A JSDO attempting to send a request to a Rollbase or OpenEdge Data Object Service that the `JSDOSession` or `Session` object manages.
- The result of the `JSDOSession` or `Session` object executing its `ping( )` method.

See also:

login( ) method (JSDOSession class) on page 275, login( ) method (Session class) on page 280, offline event on page 292, online event on page 294, ping( ) method (JSDOSession class) on page 297, ping( ) method (Session class) on page 300

data property

The data (field) and state values for a record associated with a `JSRecord` object.

Data type: `Object`

Access: Read-only

Applies to: progress.data.JSRecord class on page 133

The returned object contains a field reference property (*`field-ref`* in syntax) for each field (column) in the table, where the property name is identical to a table field name and the property value for the corresponding JavaScript data type.

You can obtain a `JSRecord` object by invoking one of the JSDO methods that returns record objects from a JSDO table reference (`find( )`, `findById( )`, or `foreach( )`) or by accessing the `record` property on a JSDO table reference that already has a working record.

Note: If a given JSDO table has a working record, you can access each *`field-ref`* of the working record directly on the corresponding table reference property (*`table-ref`*) of the JSDO. For the working record of a table reference, then, references to the `JSRecord` object of the working record and its `data` property are both implied by the table reference alone.

Caution: Never write directly to a to a *field-ref* using this `data` property; in this case, use *field-ref***only to read** the data. Writing field values using the `data` property does **not** mark the record for update when calling the `saveChanges( )` method, nor does it re-sort the record in JSDO memory according to any order you have established using the `autoSort` property. To mark a record for update and automatically re-sort the record according to the `autoSort` property, you must assign a field value either by setting a *jsdo-ref.table-ref.field-ref* for a working record or by calling the `assign( )` method on a valid *table-ref* or `JSRecord` object reference. For information on table references (*table-ref*), see the reference entry for the table reference property (JSDO).

See also:

assign( ) method (JSDO class) on page 198, assign( ) method (UIHelper class) on page 199, autoSort property on page 203, find( ) method on page 227, findById( ) method on page 229, foreach( ) method on page 231, record property on page 305, table reference property (JSDO class) on page 351

deleteLocal( ) method

Clears out all data and changes stored in a specified local storage area, and removes the cleared storage area.

Return type: `undefined`

Applies to: progress.data.JSDO class on page 110

Syntax

```
deleteLocal ( [ storage-name ] )
```

storage-name

The name of the local storage area to be removed. If *storage-name* is not specified, blank, or `null`, the name of the default storage area is used. The name of this default area is `jsdo_`*serviceName*`_`*resourceName*, where *serviceName* is the name of the Data Object Service that supports the JSDO instance, and *resourceName* is the name of the resource (table, dataset, etc.) for which the JSDO instance is created.

If this method encounters any errors, it leaves the specified storage area unchanged and throws an exception.

Example

The following code fragment clears out all the data currently stored in the default storage area and removes the storage area:

```
dataSet = new progress.data.JSDO( 'dsStaticData' );
dataSet.fill();
dataSet.saveLocal();
.
.
.
dataSet.deleteLocal();
```

See also:

acceptChanges( ) method on page 153, rejectChanges( ) method on page 305, saveChanges( ) method on page 314, saveLocal( ) method on page 330

display( ) method

Copies the field values of a given record to corresponding fields in the current HTML document for display in the form on the detail page.

The record is the working record of a table referenced in the JSDO memory of a JSDO that is associated with a `UIHelper` instance.

This method has no effect on the existing working record setting.

Return type: `null`

Applies to: progress.ui.UIHelper class on page 142, table reference property (UIHelper class) on page 353

Syntax

```
uihelper-ref.display ( )
uihelper-ref.table-ref.display ( )
```

uihelper-ref

A reference to a `UIHelper` instance. You can call the method on *uihelper-ref* if the JSDO associated with the instance has only a single table reference, and that table reference has a working record.

table-ref

A table reference that has a working on the JSDO associated with the `UIHelper` instance.

If a form field's `id` attribute, as specified by the HTML DOM, matches the name of a record field, the form field displays the value of the record field. If no HTML field corresponds to a given record field, the value of that field is not displayed.

Example

The following code fragment shows the `display( )` method displaying each record of an `eCustomer` table in JSDO memory to its respective row of a previously established jQuery listview:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );
uihelper = new progress.ui.UIHelper({ jsdo: dataSet });
uihelper.eCustomer.setListView({
    name:'listview',
    format: '{CustNum}<br>{Name}<br>{State}',
    autoLink: true
});
```

See also:

getFormFields( ) method on page 240

fill( ) method

Initializes JSDO memory with record objects from the data records in a single-table resource, or in one or more tables of a multi-table resource, as returned by the Read operation of the Data Object resource for which the JSDO is created.

This method also causes an `offline` or `online` event to fire if it detects that there has been a change in the JSDO login session's online state.

This method always executes asynchronously and returns results (either or both) in subscribed JSDO event callbacks or in callbacks that you register using methods of a Promise object returned as the method value. A Promise object is always returned as the method value if jQuery Promises (or the exact equivalent) are supported in your development environment. Otherwise, this method returns `undefined`.

Alias: `read( )`

Return type: jQuery Promise or `undefined`

Applies to: progress.data.JSDO class on page 110

Working record: After completing execution, the working record for each JSDO table is set to its first record, depending on any active parent-child relationships (for a multi-table resource) and automatic sort settings. So, for each child table, the first record object is determined by its table reference sort order (if any) and its relationship to the related working record in its parent table.

Syntax

```
fill ( [ parameter-object | filter-string ] )
```

parameter-object

An `Object` initialized with a set of properties that can be used on the server depending on a JSDO MappingType plugin that is both registered on the client and for which a corresponding Read operation is defined in the server Data Object. For more information on JSDO Mapping Type plugins and how to register and use them, see the description of the progress.data.PluginManager class on page 135.

The JSDO also provides a built-in MappingType plugin named `"JFP"` that can be used if the server Data Object is built to support it. This `"JFP"` plugin allows you to specify properties that can be used on the server to select, sort, and page the records to be returned. This built-in plugin supports any JSDO accessed by the JSDO dialect of the Kendo UI DataSource or any JSDO involved in Rollbase data access. If you are using the JSDO dialect of the Kendo UI DataSource to bind a JSDO instance to Kendo UI widgets, the DataSource initializes and invokes `fill( )` with specific *`parameter-object`* properties that are defined based on the settings of the DataSource `serverPaging`, `serverFiltering`, and `serverSorting` configuration properties. Also, if OpenEdge Data Objects are being accessed as Rollbase external objects, as well as in calls to standard Rollbase objects from Kendo UI, `fill( )` is invoked with similar *`parameter-object`* properties based on Rollbase requirements.

Note: Upon execution, the `fill( )` method immediately passes the *`parameter-object`* to a `requestMapping( )` function in the plugin to perform conversions.

The `fill( )` method further converts these *`parameter-object`* property settings to a format that is customized for each server Data Object resource, depending on its type (OpenEdge or Rollbase). This format is then applied to the resource Read operation based on the presence of a `mappingType` property set to `"JFP"` in the Data Service Catalog. For information on the requirements and the effects of using a `"JFP"` MappingType plugin for a Read operation in OpenEdge resources, see the topics on updating Business Entities for access by Kendo UI DataSources and Rollbase external objects in *OpenEdge Development: Web Services*. For Rollbase internal objects accessed using the JSDO, the requirements for using the built-in `"JFP"` MappingType plugin are managed internally by the Rollbase server.

The *`parameter-object`* properties required for the built-in `"JFP"` MappingType plugin, which you can initialize when you call `fill( )` directly, include:

- **`filter`** — An `Object` containing property settings used to select the records to be returned. These property settings are in the same format as the property settings in the Kendo UI DataSource `filter` property object. For more information, see the `filter` configuration property description in the Kendo UI DataSource documentation. The format that the `fill( )` method uses to pass this selection criteria to the server is specified, again, in the Data Service Catalog using the `mappingType` property.

  If a `filter` object is not specified, the resource Read operation determines the records to return without selection criteria sent from the client.

- **`id`** — A `string` specifying a unique ID that the resource understands to identify a specific record.

  Note: This property is not currently used by the Kendo UI.

- **`skip`** — A `number` that specifies how many records to skip before returning (up to) a page of data. You must specify this property together with the `top` property.
- **`sort`** — An expression that specifies how to sort the records to be returned using one of the following formats:
  - An `Object` with property settings in the same format as the property settings in the Kendo UI DataSource `sort` property object. For more information, see the `sort` configuration property description in the Kendo UI DataSource documentation.
  - An `Array` of strings in the same format as the *`sort-fields`* parameter of the `setSortFields( )` method.

  If this property is not specified, the resource Read operation determines the order of records to return without sort information sent from the client.

- **`tableRef`** — A `string` specifying the name of a table reference in the JSDO. This property is required when the JSDO represents a multi-table resource and the `filter` property `Object` is **also** specified with filter information.
- **`top`** — A `number` that specifies how many records (the page size) to return in a single page of data after using `skip`. You must specify this property together with the `skip` property.

  Note: The final page of a larger result set can contain a smaller number of records than `top` specifies.

Note: You must specify both `skip` and `top` to implement server paging. For example, if you want to return the 5th page of data with a page size (`top`) of 10, set `skip` to 40. If these properties are not specified together, the resource Read operation determines the records to return without using any paging information sent from the client.

Note: If you access the JSDO using only the JSDO dialect of the Kendo UI DataSource, the DataSource invokes the `fill( )` method with appropriate settings for these properties based on the corresponding Kendo UI settings. For more information, see Using the JSDO dialect of the Kendo UI DataSource on page 17.

`filter-string`

This is a `string` containing selection criteria used by the Read operation as required by the resource:

- **For an OpenEdge resource** — This is a string that the resource defines for use on an OpenEdge application server to select records to be returned, much like the `WHERE` option of the ABL record phrase. The actual format of this string and its affect on the records returned is determined by the ABL routine that implements the resource Read operation. For example, you might pass:
  - A single key value (e.g., `"30"`)
  - A relational expression (e.g., `"CustNum > 10 AND CustNum < 30"`)
  - An actual `WHERE` string (e.g., `'Item.CatDescription CONTAINS "ski & (gog* ! pol*)"'`)

  Note: For an OpenEdge resource, the JSDO requires the URI for the Read operation of the resource to contain the following query string: `"?filter=~{filter~}"`, where *`filter`* is the name of a string input parameter defined for the ABL routine that implements the operation (`INPUT filter AS CHARACTER`).

  Caution: Using an actual `WHERE` string for a dynamic ABL query can create a potential security issue.

- **For a Rollbase resource** — This is a string containing selection criteria as required by the Rollbase server to select records to be returned.

Note: If you want to use sorting or paging that your server Data Object resource supports, you must pass *`parameter-object`* instead of *`filter-string`*.

If you do not pass either *`parameter-object`* or *`filter-string`* to this method, the records returned depend entirely on the Read operation that the resource implements.

Promise method callback signatures

jQuery Promise objects define methods that register a callback function with a specific signature. The callback signatures depend on the method that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that `fill( )` returns:

Syntax:

```
promise.done( function ( jsdo , success , request ) )
promise.fail( function ( jsdo , success , request ) )
promise.always( function ( jsdo , success , request ) )
```

promise

A reference to the Promise object that is returned as the value of the `fill( )` method. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDO class on page 110.

jsdo

A reference to the JSDO that invoked the `fill( )` method (Read operation) on its resource. For more information, see the description of the jsdo property on page 272 of the request object.

success

A `boolean` that is `true` if the Read operation was successful. For more information, see the description of the success property on page 350 of the request object.

Note: It is not always necessary to test the value of *success* in a Promise method callback for the `fill( )` method, especially if the callback is registered using *promise*`.done( )` and *promise*`.fail( )`, where the callback executes according to this value.

---

request

A reference to the request object returned after the `fill( )` method completed execution and returned any results from its Read operation on the server. For more information, see the description of the request object on page 146.

General operation

This method invokes the Read operation on the resource defined for the current JSDO. The result of calling this method replaces any prior data in JSDO memory with the record objects returned by this Read operation. The record objects are stored in one or more JSDO tables that correspond to the table or tables defined for the resource. If the JSDO is accessing multi-table resource (with one or more tables), such as an OpenEdge ProDataSet, and the resource supports before-imaging, the JSDO also updates the state of its JSDO memory with any before-image data sent with the loaded record objects, including any changes to record objects recorded in their before-image data.

Caution: If the JSDO has pending record changes from the client that you want to save on the server, do not call this method before you call the JSDO `saveChanges( )` method. Otherwise, the pending changes will be lost when JSDO memory is initialized with records from the Read operation.

Note: After this method initializes JSDO memory with record objects, and if you have set up automatic sorting using the `autoSort` property, the record objects of each affected JSDO table reference are sorted in JSDO memory according to the sort order you have established for the JSDO. If sorting is done using sort fields, any `string` fields are compared according to the value of the `caseSensitive` property on a given table reference. If the `autoSort` property setting is `false` for a given table reference, its record objects are loaded in the order that they were serialized from the corresponding resource table.

Returning and handling results

This method returns results asynchronously in two different ways, and in the following order, depending on the development environment used to build the mobile app:

1. **Using JSDO named events for all environments** — The following events fire before and after the `fill( )` method executes, respectively, with results handled by callback functions that you subscribe as documented for each event:
   a. beforeFill event on page 208
   b. afterFill event on page 185
2. **Using a Promise object returned for environments that support jQuery Promises** — Any callbacks that you register using Promise object methods all execute both **after** the `fill( )` method itself and **after** any subscribed `afterFill` event callbacks complete execution. Note that the signatures of all Promise method callbacks match the signature of the `afterFill` event callback function so you can specify an existing `afterFill` event callback as the callback function that you register using any Promise method.

Because the callbacks that you register with any returned Promise methods execute only after all subscribed `afterFill` event callbacks complete execution, you can invoke logic in the Promise method callbacks to modify any processing done by the event callbacks.

If the `fill( )` method completes successfully, the *`success`* parameter for any `afterFill` event callback or Promise method and the `success` property of each handler's *`request`* parameter object are both set to `true`, and any data records returned by the resource Read operation are loaded into JSDO memory. Otherwise, both the *`success`* parameter and `success` property are set to `false`, and you can read any error results by calling the `getErrors( )` method on a single JSDO table reference, or by inspecting the setting of the `response` property in the same *`request`* parameter object.

Note: When returning before-image data from an OpenEdge ProDataSet resource, it is possible (though rare) for record-change errors that result from server update activity to be returned as well. The `fill( )` method does not return an unsuccessful result in this case. If you need to identify before-image errors in records returned by a Read operation, you can query the record objects in JSDO memory and call `getErrorString( )` on each record object to identify any such errors that were returned.

You can read any record objects loaded into JSDO memory by `fill( )` using the `find( )`, `findById( )`, `foreach( )`, and `getData( )` methods of the JSDO. You can return the schema for this data by using the `getSchema( )` method. You can create a new record object in JSDO memory using the JSDO `add( )` method, and you can update or delete a single record object in JSDO memory by using the `assign( )` or `remove( )` method, respectively. You can display a record in a UI form that you dynamically bind to the record by calling the `display( )` method on a `progress.ui.UIHelper` class instance. You can merge data returned by an invocation method with the data in JSDO memory using the `addRecords( )` method, and you can store and merge records in JSDO memory both to and from a local storage location using the `saveLocal( )`, `readLocal( )`, and `addLocalRecords( )` methods.

Example

The following code fragment shows the `fill( )` method invoked on a JSDO for an OpenEdge single-table resource (`dsCustomerOrder`), with results returned using the `afterFill` event:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );
dataSet.subscribe("afterFill", onAfterFill);

dataSet.fill();

function onAfterFill( jsdo , success , request ) {
    var lenErrors,
        errors,
        errorType;

    if (success) {
        /* for example, add code to display all records on a list */
        jsdo.foreach(function (jsrecord) {
            /* you can reference the fields as jsrecord.data.field */
        });
    }
    else {
        /* handle Read operation errors */
        errors = jsdo.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
            }
            console.log("READ ERROR: " + errorType + errors[idxError].error);
            if (errors[idxError].responseText) {
                console.log("HTTP FULL TEXT: " + errors[idxError].responseText);
            }
        }
    }
};
```

Note that for an OpenEdge resource, the `getErrors( )` method can return one or more error messages for a Read operation in an array of error objects.

The following code fragment shows the `fill( )` method invoked on a JSDO for a similar OpenEdge single-table resource (`dsCustomerOrder`), with results returned using a Promise object:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );

dataSet.fill().done(
    function( jsdo, success, request ) {
        /* for example, add code to display all records in a list */
        jsdo.foreach(function (jsrecord) {
            /* you can reference the fields as jsrecord.data.<fieldName> */
        });
    }).fail(
    function( jsdo, success, request ) {
        var lenErrors,
        errors,
        errorType;

        /* handle Read operation errors */
        errors = jsdo.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
            }
            console.log("READ ERROR: " + errorType + errors[idxError].error);
            if (errors[idxError].responseText) {
                console.log("HTTP FULL TEXT: " + errors[idxError].responseText);
            }
        }
    }
});
```

Using a Promise object, the Promise `done` and `fail` functions do not have to test the `success` parameter for a successful (`true`) or failed (`false`) execution of the `fill( )` method, because `done` executes only when `fill( )` succeeds and `fail` executes only when `fill( )` fails.

The following code fragment shows the `fill( )` method invoked on a JSDO for a Rollbase resource (`Orders`), with results returned using a Promise object:

```
dataSet = new progress.data.JSDO( 'Orders' );

dataSet.fill().done(
    function( jsdo, success, request ) {
        /* for example, add code to display all records on a list */
        jsdo.foreach(function (jsrecord) {
            /* you can reference the fields as jsrecord.data.<fieldName> */
        });
    }).fail(
    function( jsdo, success, request ) {
        var lenErrors,
        errors,
        errorType;

        /* handle operation errors */
        errors = jsdo.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            console.log("READ ERROR: " + errors[idxError].error);
            if (errors[idxError].responseText) {
                console.log("HTTP FULL TEXT: " + errors[idxError].responseText);
            }
        }
    });
```

For a Rollbase resource, any error results from a Read operation are returned by `getErrors( )` as the single error type `progress.data.JSDO.ERROR` with the `errors[idxError].error` property set to a single returned `string` value.

See also:

autoSort property on page 203, caseSensitive property on page 213, getErrors( ) method on page 233, getId( ) method on page 242, invocation method on page 262, progress.data.PluginManager class on page 135, progress.ui.UIHelper class on page 142, response property on page 312, saveChanges( ) method on page 314, setSortFields( ) method on page 339, success property on page 350

find( ) method

Searches for a record in a table referenced in JSDO memory and returns a reference to that record if found. If no record is found, it returns `null`.

Return type: progress.data.JSRecord class on page 133

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Working record: After completing execution, any record found becomes the working record for the associated table. If the searched table has child tables in the same multi-table resource, and the `useRelationships` property is `true`, the working record of the result set for each child is set to the first record as determined by the relationship to its respective parent. If a record is not found, the working record is not set, and the working records of any child tables are also not set.

Syntax

```
jsdo-ref.find ( funcRef )
jsdo-ref.table-ref.find ( funcRef )
```

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference.

table-ref

A table reference on the JSDO.

funcRef

A reference to a JavaScript callback function that returns a `boolean` value and has the following signature:

Syntax:

```
function [ func-name ] ( jsrecord-ref )
```

Where *func-name* is the name of a callback function that you define external to the `find( )` parameter list and *jsrecord-ref* is a `JSRecord` reference to the next available record on the specified table reference. You can then pass *func-name* to the `find( )` method as the *funcRef* parameter. Alternatively, you can specify *funcRef* as the entire inline function definition without *func-name*.

The `find( )` method executes your *funcRef* callback for each record of the table reference, until it returns `true`, indicating that the callback has found the record. You can then test the field values on the `data` property of *jsrecord-ref* to determine the result. Otherwise, your callback returns `false` and the `find( )` method executes the callback again on the next available record.

If your *funcRef* callback finds the record, `find( )` completes execution with both its return value and the `record` property of the associated table reference set to the `JSRecord` reference of the found working record. If `find( )` reaches the end of the available records without *funcRef* returning `true`, `find( )` completes execution with both its return value and the `record` property on the table reference set to `null`, indicating that the sought for record was not found.

If the associated table reference is for a child table in a multi-table resource, if the `useRelationships` property is `true`, `find( )` uses the relationship to filter out all but the child records of the working record in the parent table. However, if the working record of the parent is not set, `find( )` throws an error. If `useRelationships` is `false`, the search includes all records of the child table and no error is thrown.

Example

In following code fragment, `jsdo` references a single `customer` table:

```
var jsdo = new progress.data.JSDO( 'customer' );

jsdo.find(function(jsrecord) {
  return (jsrecord.data.CustNum == 10);
});
```

The inline function passed to `find( )` returns `true` or `false` based on the value of the `CustNum` property of the object returned by the `data` property for the currently available `JSRecord` reference.

See also:

data property on page 217, foreach( ) method on page 231, record property on page 305

findById( ) method

Locates and returns the record in JSDO memory with the internal ID you specify.

If no record is found, it returns `null`. You can access the internal ID of the working record of a table reference, or any `JSRecord` object, by calling the `getId( )` method on the object returned by the `data` property of the `JSRecord`.

Return type: progress.data.JSRecord class on page 133

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Working record: After completing execution, any record found becomes the working record for the associated table. If the searched table has child tables in the same multi-table resource, and the `useRelationships` property is `true`, the working record of the result set for each child is set to the first record as determined by the relationship to its respective parent. If a record is not found, the working record is not set, and the working records of any child tables are also not set.

Syntax

```
jsdo-ref.findById ( id )
jsdo-ref.table-ref.findById ( id )
```

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference.

table-ref

A table reference on the JSDO.

id

The internal record ID used to match a record of the table reference. This is the same value originally returned for the record using the `getId( )` function. It is typically used to create a jQuery listview

row to display the record or a detail form used to display the record in the current HTML document. Later, when a listview row or detail form is selected, the corresponding `id` attribute with this value can be used to return the record from the JSDO, possibly to update the record with new data values input by the user.

If `findById( )` locates a record with the matching record ID, it completes execution with both its return value and the `record` property of the table reference set to the `JSRecord` reference of the found working record. If the function does not locate the record, it completes execution with both its return value and the `record` property on the table reference set to `null`, indicating that no record of the table reference has a matching internal record ID.

If the table reference references a child table in a multi-table resource, when the `useRelationships` property is `true`, `findById( )` uses the relationship to filter out all but the child records of the working record in the parent table; the remaining child records are excluded from the search. If `useRelationships` is `false` or the working record of the parent is not set, the search includes all records of the child table and no error is thrown.

Example

The following code fragment shows a jQuery event defined on a save button to save the current field values for a customer detail form to the corresponding `eCustomer` record in JSDO memory:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );

$('#btnSave').bind('click', function(event) {
  var jsrecord = dataSet.eCustomer.findById($('#custdetail #id').val());
  if (jsrecord) {jsrecord.assign();};
  dataSet.saveChanges();
});
```

The form has been displayed with previous values of the same record. When the button is clicked, the event handler finds the original `eCustomer` record by calling `findById( )` with the `id` attribute of the form (`$('#custdetail #id').val()`), which is set to the internal ID of the record. The `jsrecord.assign( )` method then updates the record from the values of the corresponding form fields and `saveChanges( )` invokes the resource `"update"` operation on the AppServer to save the updated record to its data source.

See also:

data property on page 217, foreach( ) method on page 231, getId( ) method on page 242

fnName property

For an Invoke operation, the name of the custom JSDO invocation method that executed the operation.

The `fnName` property is `null` in the case of a request object returned by a resource Create, Read, Update, Delete, or Submit operation.

Data type: `string`

Access: Read-only

Applies to: request object on page 146

The `fnName` property is available only for the following JSDO event or in the request object returned to a jQuery Promise callback:

- `afterInvoke`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe or register to JSDO events, the jQuery Promise returned from an `invoke( )` method, and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO. This object is also returned as the value of any JSDO invocation method that you execute synchronously.

Note: The value of the `fnName` property is the same as that of the *`op-name`* parameter passed to the `subscribe( )` method that subscribed to the current invoke operation event.

See also:

invocation method on page 262, invoke( ) method on page 264, subscribe( ) method (JSDO class) on page 347

foreach( ) method

Loops through the records of a table referenced in JSDO memory and invokes a user-defined callback function as a parameter on each iteration.

With each iteration, it also sets the current record as the working record and passes it as a parameter to the callback function. This function can then operate on the working record and return a value indicating whether the `foreach( )` terminates the loop or invokes the callback function on the next working record of the table.

If the referenced table has child tables in the same multi-table resource, and the `useRelationships` property is `true`, with each iteration through the loop, the working record of the result set for each child is set to the first record as determined by the relationship to its respective parent.

Return type: `null`

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Working record: After completing execution, the working records of the associated table, and any child tables, are the most recent working records established when the method terminates the loop.

Syntax

```
jsdo-ref.foreach ( funcRef )
jsdo-ref.table-ref.foreach ( funcRef )
```

`jsdo-ref`

A reference to the JSDO. You can call the method on *`jsdo-ref`* if the JSDO has only a single table reference.

`table-ref`

A table reference on the JSDO.

funcRef

A reference to a JavaScript callback function that returns a `boolean` value and has the following signature:

Syntax:

```
function [ func-name ] ( jsrecord-ref )
```

Where *func-name* is the name of a callback function that you define external to the `foreach( )` parameter list and *jsrecord-ref* is a `JSRecord` object reference to the next working record on the table reference. You can then pass *func-name* to the `foreach( )` method as the *funcRef* parameter. Alternatively, you can specify *funcRef* as the entire inline function definition without *func-name*.

The `foreach( )` method executes your *funcRef* callback for each record of the table reference, making this record the working record and passing it in as *jsrecord-ref*. You can then access the field values of the working record using the `data` property on *jsrecord-ref* or any field references available from the table reference. You can also invoke other JSDO methods, for example, to operate on the working record, including additional calls to `foreach( )` to operate on working records of any child tables.

Your *funcRef* callback can terminate the `foreach( )` loop by returning `false`. If the callback does not return `false`, the loop continues.

If the table reference references a child table in a multi-table resource, when the `useRelationships` property is `true`, `foreach( )` uses the relationship to filter out all but the child records of the working record in the parent table. However, if the working record of the parent is not set, `foreach( )` throws an error. If `useRelationships` is `false`, the loop includes all records of the child table and no error is thrown.

Example

After creating a JSDO for a `dsCustomer` resource and loading it with record objects, the following code fragment shows the `foreach( )` method looping through `eCustomer` records in JSDO memory and displaying the `CustNum` and `Name` fields from each record, one record per line, to the current HTML page, and also to the console log:

```
jsdo = new progress.data.JSDO({ name: 'dsCustomer' });
jsdo.subscribe( 'AfterFill', onAfterFillCustomers, this );

jsdo.fill();

function onAfterFillCustomers(jsdo, success, request) {
  jsdo.eCustomer.foreach( function(customer) {
    document.write(customer.data.CustNum + ' ' + customer.data.Name + '<br>');
    console.log(customer.data.CustNum + ' ' + customer.data.Name);
  } );
};
```

See also:

data property on page 217, find( ) method on page 227, progress.data.JSRecord class on page 133

getData( ) method

Returns an array of record objects for a table referenced in JSDO memory.

If this is a child table in a multi-table resource, and the `useRelationships` property is `true`, the specific record objects in the result set depends on the relationship to its parent.

Return type: `Object` array

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Syntax

```
jsdo-ref.getData ( )
jsdo-ref.table-ref.getData ( )
```

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference.

table-ref

A table reference on the JSDO.

See also:

getSchema( ) method on page 247

getErrors( ) method

Note: Updated to support all possible types of error returns in Progress Data Objects 4.3 or later.

Returns an array of errors from the most recent invocation of Create, Read, Update, Delete, or Submit operations (CRUD or Submit) that you have invoked by calling the JSDO `fill( )` or `saveChanges( )` method on a Data Object resource.

You can call this method on a single JSDO table reference at any point after `fill( )` or `saveChanges( )` is invoked, typically in a callback function executed for each fired JSDO `after*` event or for any Promise object returned by `fill( )` or `saveChanges( )`. The errors returned from this method apply either to the data in the single JSDO table that is involved in the invoked operation or operations, or to the web or application servers that handle the requests. These errors are cleared with the next invocation of `fill( )` or `saveChanges( )` on the same resource, no matter the setting of the JSDO `autoApplyChanges` property.

This method allows you to access error information returned for every possible error, whether it is generated by:

- **Routines** on the application server that implement the CRUD or Submit operations on a given Data Object resource, and whether or not the resource supports before-imaging

- The **application server** that implements the specified Data Object Service, regardless of the Data Object resource and its executed CRUD or Submit operations
- The **specified web server or web application**, regardless of the Data Object Service you are accessing

---

Note: Using this method, you have no need to inspect error information returned by either the `response` property or the `xhr` property (XMLHttpRequest object) in any request object returned by `fill( )` or `saveChanges( )`, or the error string returned by the `getErrorString( )` method called on a `JSRecord` object.

Note: If you have a reference to the JSDO used to instantiate the JSDO dialect of the Kendo UI DataSource, you can also call this method in any callback function registered for the `error` event on the DataSource. For more information, see Using the JSDO dialect of the Kendo UI DataSource on page 17.

Return type: Array of `Object`

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Syntax

```
jsdo-ref.getErrors ( )
jsdo-ref.table-ref.getErrors ( )
```

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference.

table-ref

A table reference on the JSDO.

Response

The array returned by this method contains errors associated with the resource operation or operations that were invoked by the most recent call to `fill( )` or `saveChanges( )`. Each element of the array returned by this method is an `Object` that corresponds to a single type of error returned by an operation on the resource and contains the following properties:

- **`error`** — A `string` value that describes the error depending on the error type. For more information, see the `type` property of this `Object`.
- **`errorNum`** — A `numeric` value that returns an error number. This property is returned only if the error type is `progress.data.JSDO.APP_ERROR`. For more information, see the `type` property of this `Object`.

  Note: Applies to Progress Data Objects Version 4.3 or later.

- **`id`** — A `string` value containing the internal ID of the record object on which the resource operation has returned the error. This property can be returned, if appropriate, for any error type **except** `progress.data.JSDO.ERROR`. If the error type is `progress.data.JSDO.APP_ERROR` and is returned for a resource Read operation, the `id` property is **never** returned. If the error type is `progress.data.JSDO.DATA_ERROR` and is returned for a resource CUD or Submit operation, the `id`

property is **always** returned. How you can use this value to return information about the record object depends on the error type. For more information, see the `type` property of this `Object`.

- **`responseText`** — A `string` value that contains the complete text of the HTTP response when the error type is `progress.data.JSDO.ERROR`. For example, this can be the HTML text of an error page displayed by the browser, which you can also display using the browser's web inspector. For more information, see the `type` property of this `Object`.

  Note: Applies to Progress Data Objects Version 4.3 or later.

- **`type`** — A numeric constant that identifies the origin of the error returned by this `Object`:

  Note: Applies to Progress Data Objects Version 4.3 or later.

  - **`progress.data.JSDO.DATA_ERROR`** — An error returned for a CUD or Submit operation on a record object in a resource with before-image data. The response for this type of error uses HTTP status code 200 OK, and the error information is returned using `prods` properties of the ProDataSet object in the response.

    For this error type, the `error` property returns an error string associated with the record object and this record object is identified by the `id` property. For a Submit operation, you can look up the corresponding before-image data for the record by invoking the `getId( )` method on each element of the array returned by the `jsrecords` property on the returned request object. You can also find the current data for the record in JSDO memory, if it exists, using the JSDO `findById( )` method.

    Note: From an OpenEdge Data Object resource that supports before-imaging, this error type can be returned by setting both the ABL `ERROR` and `ERROR-STRING` attributes on a buffer object handle set to the corresponding record in the corresponding ABL temp-table. For more information on these attributes, see *OpenEdge Development: ABL Reference* and *OpenEdge Development: ProDataSets*.

  - **`progress.data.JSDO.APP_ERROR`** — An HTTP 4xx or 5xx error returned for a CRUD operation on a resource **without** before-image data (and sometimes also for a CUD or Submit operation **with** before-image data).

    For this error type, the `error` property returns an error message associated with the CRUD operation and the `errorNum` property returns the numeric value of the error number. For a CUD operation, the associated record object in the resource can also be identified using the value of the `id` property. You can also find the current data for the record in JSDO memory, if it exists, using the JSDO `findById( )` method.

    Note: From an OpenEdge Data Object resource, this error type can be returned using a constructor or method of the built-in ABL class, `Progress.Lang.AppError`. For more information, see *OpenEdge Development: ABL Reference*.

    Other properties are available to interpret this information, depending on the resource operation and event callback where `getErrors( )` is called, and include:

    - In the callback for a CUD operation event (`afterCreate`, `afterUpdate`, or `afterDelete`), you can also return the record object identified by the `id` property as the value of the `jsrecord` property on the request object returned by the callback.
    - In the callback for an `afterSaveChanges` event on completion of a call to `saveChanges(false)` (**without** Submit), you can identify the request object returned for each CUD operation by inspecting the value of the `batch` property on the request object returned by the callback and compare this

information with the information and data returned by the `error` property and `id` property for this error type.

- In the callback for an `afterSaveChanges` event on completion of a call to `saveChanges(true)` (**with** Submit), if this error type is returned, it is typically not returned with an `id` property, as the error type does not apply to a particular record object.
- In the callback for an `afterFill` event (on completion of a Read operation invoked by a call to `fill( )`), the `id` property is never returned, as the error type does not apply to a particular record object.

Note: The same options that apply to callbacks for `afterSaveChanges` and `afterFill` events also apply to corresponding callbacks registered to returned Promise objects.

- **`progress.data.JSDO.RETVAL`** — An HTTP 4xx or 5xx error that returns a CRUD operation return value on a resource **without** before-image data (and sometimes also for a CUD or Submit operation **with** before-image data).

  For this error type, the `error` property returns the CRUD operation return value as a string. For a CUD operation, the associated record object in the resource can also be identified by the `id` property. You can find the current data for the record in JSDO memory, if it exists, using the JSDO `findById( )` method.

  Note: From an OpenEdge Data Object resource, this error type can be returned using the `RETURN ERROR` statement, or using a constructor or method of the built-in ABL class, `Progress.Lang.AppError`. For more information, see *OpenEdge Development: ABL Reference*.

  Note: OpenEdge Data Object Services using the WebHandler service provider can return multiple `RETVAL` errors per operation in the array returned by `getErrors( )`, including one for the operation itself and one for each routine on the operation call stack. Any `RETVAL` error, regardless of the Data Object Service, can be followed by one or more `APP_ERROR` errors.

- **`progress.data.JSDO.ERROR`** — The text of a JavaScript exception or a web server HTTP 4xx or 5xx error returned without regard to any particular CRUD or Submit operation that was invoked.

  For an HTTP 4xx or 5xx error, the `error` property returns an HTTP error string in the following form:

  ```
  HTTP error-number error-text
  ```

  Where:

  - *`error-number`* is the HTTP 4xx or 5xx numeric value. For example, 500.
  - *`error-text`* is a brief description of the web server error. For more information, you can inspect the value of the `responseText` property or view the associated web page displayed by the browser web inspector.

If there are no errors for the operation and resource on which `getErrors( )` is called, this method returns an empty array (`[]`).

Note: For a CUD or Submit operation on a resource with before-image data that can return a `DATA_ERROR`, `APP_ERROR`, or `RETVAL` error, the operation can return **either** a `DATA_ERROR` error **or** it can return a `RETVAL` error followed by one or more `APP_ERROR` errors, but not **both**.

Note: If the method returns any errors, the JSDO also returns the request object for the operation with its `success` property set to `false`. This is also the value of the *`success`* parameter passed to any callback function subscribed to a JSDO `after*` event or passed to any callback function registered by a JSDO Promise object returned by `fill( )` or `saveChanges( )`.

Note: This method does not track errors returned by an Invoke operation.

Example

The following code fragment shows the `getErrors( )` method invoked on the JSDO in the callback function (`onAfterSaveChanges`) that is subscribed to the JSDO `afterSaveChanges` event after invoking a Submit operation (`saveChanges(true)`) on the JSDO OpenEdge ProDataSet resource named `'Customer'` with a `ttCustomer` table, and which supports both before-imaging and the Submit operation:

```
dsCustomer = new progress.data.JSDO({ name: 'Customer' });
dsCustomer.subscribe('afterFill', onAfterFillCustomers, this);
dsCustomer.subscribe('afterSaveChanges', onAfterSaveChanges, this);
. . .

dsCustomer.saveChanges(true); /* Invoke Submit */
. . .

function onAfterSaveChanges(jsdo, success, request) {
    var errors;
    var errorType;
    console.log("DEBUG: AfterSaveChanges: " + success + " errors: " + errors.length);

    if (!success) {
        errors = jsdo.ttCustomer.getErrors();
        for (var i = 0; i < errors.length; i++) {
            switch(errors[i].type) {
                case progress.data.JSDO.DATA_ERROR:
                    errorType = "Server Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #" + errors[i].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
                case default:
                    errorType = null; // Unexpected errorType value
                    break;
            }
            if (errorType) {
                console.log("ERROR: " + errorType + errors[i].error);
                if (errors[i].id) { /* error with record object */
                    console.log("RECORD ID: " + errors[i].id);
                    /* Possibly log record change information based on data error
                       record object found in the request.jsrecords using getId()
                    */
                }
                if (errors[i].responseText) {
                    console.log("HTTP FULL TEXT: "
                                + errors[i].responseText);
                }
            }
            else { /* unexpected errorType */
                console.log("UNEXPECTED ERROR TYPE: "
                            + errors[i].type);
            }
        }
    }
}
```

Note: If the `'Customer'` resource supports before-imaging but does not support a Submit operation, this same code fragment continues to work by changing the `saveChanges( )` method call from `saveChanges(true)` to `saveChanges(false)` (or simply, `saveChanges()`). However, in this case, there is no `request.jsrecords` property available for access. Instead, you might locate information on the record and its record change operation by inspecting the `request.batch` property.

See also:

autoApplyChanges property on page 202, batch property on page 205, getErrorString( ) method on page 239, fill( ) method on page 220, findById( ) method on page 229, getId( ) method on page 242, jsrecord property on page 272, progress.data.JSRecord class on page 133, response property on page 312, saveChanges( ) method on page 314, xhr property on page 358

getErrorString( ) method

Returns any before-image error string in the data of a record object referenced in JSDO memory that was set as the result of a resource Create, Update, Delete, or Submit operation.

If there is no error string in the data of the specified record object, this method returns `undefined`.

Note: This error string can be returned from an OpenEdge Data Object resource that supports before-imaging by setting both the ABL `ERROR` and `ERROR-STRING` attributes on a buffer object handle set to the corresponding record in the corresponding ABL temp-table. For more information on these attributes, see *OpenEdge Development: ABL Reference* and *OpenEdge Development: ProDataSets*.

The specified record object can be either the working record of a referenced table, or any record provided by a `JSRecord` object.

Return type: `string`

Applies to: progress.data.JSRecord class on page 133, progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Syntax

```
jsrecord-ref.getErrorString ( )
jsdo-ref.getErrorString ( )
jsdo-ref.table-ref.getErrorString ( )
```

jsrecord-ref

A reference to a `JSRecord` object for a table record in JSDO memory.

You can obtain a `JSRecord` object by:

- Invoking a JSDO method that returns record objects from a JSDO table reference (`find( )`, `findById( )`, or `foreach( )`)
- Accessing the `record` property on a JSDO table reference that already has a working record.
- Accessing the *`record`* parameter passed to the callback of a JSDO `afterCreate`, `afterDelete`, or `afterDelete` event.

- Accessing each record object provided by the `jsrecords` property on the *`request`* object parameter passed to the callback of a JSDO `afterSaveChanges` event, or passed to the callback of any Promise object returned from the `saveChanges( )` method. The `jsrecords` property is only available on completion of a Submit operation (`saveChanges(true)`) on a resource that supports before-imaging, and **not** if the resource supports before-imaging without Submit.

`jsdo-ref`

A reference to the JSDO. You can call the method on *`jsdo-ref`* if the JSDO has only a single table reference, and that table reference has a working record.

`table-ref`

A table reference on the JSDO that has a working record.

The error string returned by this function contains error information only for a Data Object Service operation on a resource that supports before-imaging. This information can only be returned for a resource Create, Update, Delete, or Submit operation (`saveChanges(true)`) that affects the before-image data of a resource record. The JSDO stores the error information from each affected resource record in the corresponding JSDO record object as an internal property of its data with a reserved name.

Note: This method is most useful when the `autoApplyChanges` property is `false`. When `autoApplyChanges` is `true`, the method automatically accepts or rejects record changes, and clears all associated error conditions and information after the final `after*` event is fired and handled (if handled) for a given operation.

See also:

addRecords( ) method on page 175, afterCreate event on page 181, afterDelete event on page 183, afterUpdate event on page 195, autoApplyChanges property on page 202, find( ) method on page 227, findById( ) method on page 229, foreach( ) method on page 231, saveChanges( ) method on page 314, record property on page 305

getFormFields( ) method

Reads the schema of the referenced table and returns a `string` containing the HTML text of field definitions created by processing a `UIHelper` form field template against fields that you specify.

Return type: `string`

Applies to: progress.ui.UIHelper class on page 142, table reference property (UIHelper class) on page 353

Syntax

```
uihelper-ref.getFormFields ( [ array ] )
uihelper-ref.table-ref.getFormFields ( [ array ] )
```

`uihelper-ref`

A reference to a `UIHelper` instance. You can call the method on *`uihelper-ref`* if the JSDO associated with the instance has only a single table reference.

table-ref

A table reference on the JSDO associated with the `UIHelper` instance.

array

An array of `string` specifying the names of a subset of fields from the table schema for which to return HTML field definitions. If this parameter is omitted, the HTML definitions for all fields in the table are returned.

If you do not specify the fields for which to return HTML, the method returns HTML field definitions for all the fields in the table, including a field definition for the JSDO-reserved internal record ID.

If you do specify fields, only the HTML definitions for the specified fields are returned. If you also want to return the HTML field definition for the internal record ID, you must specify `_id` as a field to return. (See the example.)

The `UIHelper` class provides a default form field template for a jQuery Mobile environment. The following code fragment shows the default template values:

```
<div data-role="fieldcontain">
    <label for="{__name__}">{__label__}</label>
    <input id="{__name__}" name="{__name__}" placeholder="" value="" type="text" />
</div>
```

As shown above, the default template uses the following substitution parameters:

- **`{__name__}`** — The field name as defined in the schema
- **`{__label__}`** — The field's title property as defined in the schema

You can define a template to be used in place of the default by calling `setFieldTemplate( )`.

Example

The following code fragment invokes `getFormFields( )` to generate HTML definitions for three fields of the the associated JSDO `eCustomer` table, including a field for the JSDO-reserved record ID:

```
var dataSet = new progress.data.JSDO( 'dsCustomerOrder' );
var htmlText = "";

uihelper = new progress.ui.UIHelper({ jsdo: dataSet });
htmlText = uiHelper.eCustomer.getFormFields([ '_id', 'CustNum' , 'Name' ]);
/* write out htmlText to the current document */
```

Unspecified code typically writes out the HTML from `htmlText` to a field container in the current document.

See also:

getFormRecord( ) method on page 241, setFieldTemplate( ) method on page 334

getFormRecord( ) method

Returns a `JSRecord` object for the record shown in the form on the detail page.

Return type: progress.data.JSRecord class on page 133

Applies to: progress.ui.UIHelper class on page 142, table reference property (UIHelper class) on page 353

Syntax

```
uihelper-ref.getFormRecord ( [ detail-page-name ] )
uihelper-ref.table-ref.getFormRecord ( [ detail-page-name ] )
```

uihelper-ref

A reference to a `UIHelper` instance. You can call the method on *uihelper-ref* if the JSDO associated with the instance has only a single table reference.

table-ref

A table reference on the JSDO associated with the `UIHelper` instance, and that table reference has the record displayed in the form.

detail-page-name

An optional `string` specifying the name of the detail page in the current form. If not specified, the method uses the name set with the `setDetailPage( )` method.

See also:

getFormFields( ) method on page 240, setDetailPage( ) method on page 333

getId( ) method

Returns the unique internal ID for the record object referenced in JSDO memory.

The specified record object can be either the working record for a referenced table, or any record provided by a `JSRecord` object.

Return type: `string`

Applies to: progress.data.JSRecord class on page 133, progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Syntax

```
jsrecord-ref.getId ( )
jsdo-ref.getId ( )
jsdo-ref.table-ref.getId ( )
```

jsrecord-ref

A reference to a `JSRecord` object for a table record in JSDO memory.

You can obtain a `JSRecord` object by:

- Invoking a JSDO method that returns record objects from a JSDO table reference (`find( )`, `findById( )`, or `foreach( )`)
- Accessing the `record` property on a JSDO table reference that already has a working record.
- Accessing the *`record`* parameter passed to the callback of a JSDO `afterCreate`, `afterDelete`, or `afterDelete` event.
- Accessing each record object provided by the `jsrecords` property on the *`request`* object parameter passed to the callback of a JSDO `afterSaveChanges` event, or passed to the callback of any Promise object returned from the `saveChanges( )` method. The `jsrecords` property is only available on completion of a Submit operation (`saveChanges(true)`) on a resource that supports before-imaging, and **not** if the resource supports before-imaging without Submit.

`jsdo-ref`

A reference to the JSDO. You can call the method on *`jsdo-ref`* if the JSDO has only a single table reference, and that table reference has a working record.

`table-ref`

A table reference on the JSDO that has a working record.

The internal record ID returned by this function is a unique value generated by the JSDO for each record object loaded into JSDO memory using the `fill( )`, `add( )`, or `addRecords( )` methods. The JSDO adds this value to the data of each record object in a property with a reserved name. The JSDO uses this record ID in the `progress.ui.UIHelper` class, to map the record objects in JSDO memory to the HTML elements of mobile apps.

To return and set the specified record as the working record, you can pass any value returned by this method to the `findById( )` method called on the associated table reference.

Note: The value assigned to the internal record ID for any given record object can change with each invocation of the `fill( )` method.

See also:

add( ) method on page 158, addRecords( ) method on page 175, fill( ) method on page 220, findById( ) method on page 229

getListViewRecord( ) method

Returns a `JSRecord` object for a specified item in a list view.

Return type: progress.data.JSRecord class on page 133

Applies to: progress.ui.UIHelper class on page 142, table reference property (UIHelper class) on page 353

Syntax

```
uihelper-ref.getListViewRecord ( list_item )
uihelper-ref.table-ref.getListViewRecord ( list_item )
```

uihelper-ref

A reference to a `UIHelper` instance. You can call the method on *uihelper-ref* if the JSDO associated with the instance has only a single table reference.

table-ref

A table reference on the JSDO associated with the `UIHelper` instance, and that table reference has the record displayed as the specified item in the list view.

list_item

The HTML list item element corresponding to the desired record, provided as a `string` in the following form:

Syntax:

```
<li data-id="id-value"> ... </li>
```

See also:

addItem( ) method on page 168, getFormRecord( ) method on page 241, setListView( ) method on page 336

getPlugin( ) method

Note: Applies to Progress Data Objects Version 4.3 or later.

Returns a reference to the registered JSDO plugin object with the specified name.

Return type: `Object`

Applies to: progress.data.PluginManager class on page 135

Syntax

```
progress.data.PluginManager.getPlugin( name )
```

name

A `string` expression that evaluates to the name of the plugin. If a plugin by this name does not exist, the method returns `undefined`.

Example

The JSDO provides a built-in "JFP" plugin with a requestMapping function that supports the JSON Filter Pattern (JFP). The following example dynamically modifies this built-in plugin to handle the response from a particular OpenEdge Data Object resource, as follows:

```
jsdo = new progress.data.jsdo( . . . );

var defaultPlugin = progress.data.PluginManager.getPlugin("JFP");
defaultPlugin.responseMapping =

    function ( jsdo, response, info ) {


        var newData = response.dsCustomer.eCustomer;

        for (var i = 0; i < newData.length; i++) {
            newData[i].Country = "USA";
        }

        return response;

    };
```

First, the getPlugin( ) method returns a reference to this built-in plugin as defaultPlugin. A custom responseMapping function is then dynamically registered for the plugin that modifies data in the dsCustomer.eCustomer table returned from the OpenEdge Data Object resource.

Upon execution, the fill( ) method performs conversions using the plugin's default requestMapping function, then invokes the resource Read operation on the server. When the response from the server is returned, this customized responseMapping function executes to modify response data before it is loaded into JSDO memory.

Note: For the OpenEdge Read operation to make use of this built-in "JFP" plugin, the ABL method that implements the operation in the Business Entity (the Data Object source) must be annotated with a mappingType property set to "JFP". For more information on annotating this property, see the Notes in the description of the addPlugin( ) method.

See also

addPlugin( ) method on page 171, fill( ) method on page 220

getProperties( ) method

Note: Applies to Progress Data Objects Version 4.3 or later.

Returns an object containing the names and values of the user-defined properties defined in the current JSDO instance.

Return type: Object

Applies to: progress.data.JSDO class on page 110

Syntax

```
getProperties ( )
```

Example

The following code fragment replaces any existing user-defined properties in `jsdoCustOrders` with the two user-defined properties, `"prop1"` and `"prop2"`. It then calls `getProperties( )` to return an object containing all the JSDO's user-defined properties (`"prop1"` and `"prop2"`) and writes their values to the console log:

```
jsdoCustOrders = new progress.data.JSDO(. . .);

jsdoCustOrders.setProperties({"prop1", 100, "prop2", 500});
var props = jsdoCustOrders.getProperties();
console.log("prop1: " + props.prop1);
console.log("prop2: " + props.prop2);
```

The output from the `console.log` statements in this example appear as follows:

```
prop1: 100
prop2: 500
```

See also:

getProperty( ) method on page 246, setProperties( ) method on page 337

getProperty( ) method

Note: Applies to Progress Data Objects Version 4.3 or later.

Returns the value of the specified JSDO user-defined property.

Return type: Any

Applies to: progress.data.JSDO class on page 110

Syntax

```
getProperty ( name )
```

name

The name of a user-defined property to query from the JSDO.

Example

The following code fragment writes the value of the user-defined property, `"prop1"`, defined in `jsdoCustOrders` to the console log:

```
jsdoCustOrders = new progress.data.JSDO(. . .);

console.log("prop1: " + jsdoCustOrders.getProperty("prop1");
```

See also:

getProperties( ) method on page 245, setProperty( ) method on page 338

getSchema( ) method

Returns an array of objects, one for each field that defines the schema of a table referenced in JSDO memory.

The properties of each object define the schema elements of the respective field.

Return type: Array of `Object`

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Syntax

```
jsdo-ref.getSchema ( )
jsdo-ref.table-ref.getSchema ( )
```

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference.

table-ref

A table reference on the JSDO.

The properties in each `Object` of the array include those defined in the Data Service Catalog for each field of the specified resource table schema, for example:

- `ablType`
- `default`
- `format`
- `name`
- `type`

For more information on these and other properties that can define the schema for a table field in a Data Service Catalog, see the specification for the Cloud Data Object (CDO) Catalog file in:

https://github.com/CloudDataObject/CDO/blob/master/CloudDataObject_Catalog.pdf

See also:

getData( ) method on page 233, fill( ) method on page 220

getSession( ) stand-alone function

Note: Applies to Progress Data Objects Version 4.3 or later.

A stand-alone function that creates and returns a `progress.data.JSDOSession` instance with a specified JSDO login session already established and a specified Data Service Catalog already loaded.

This function combines the features of:

- Preparing the `JSDOSession` class constructor and instantiating the instance.
- Calling the `login( )` method on the instance to establish a JSDO login session.
- Calling the `addCatalog( )` method on the instance to load a single Data Service Catalog.

The resulting login session is then ready to support creation of JSDO instances for the one or more resources provided by the specified Data Object Service. In addition, this function provides automatic support for page refresh in web apps, if specified, when you call it.

This method is always executed asynchronously and returns results in callbacks that you register using methods of a Promise object returned as the function value.

Note: In order to invoke this function successfully, jQuery Promises must be supported in your development environment. Otherwise, the function throws an exception.

Note: This function does not support proxy servers (servers that function as a security service).

Return type: jQuery Promise

Applies to: The `progress.data` namespace

Syntax

```
progress.data.getSession ( parameter-object )
```

parameter-object

An object that contains the following configuration properties:

- **`serviceURI`** — (Required) A `string` expression containing the URI of the web application for which to start a JSDO login session. This web application must support one or more Data Object Services in order to create JSDOs for the resources provided by the application. This URI is appended with a string that identifies a resource to access as part of the login process.

If the mobile app from which you are logging in is a hybrid app that will be installed to run directly in a native device container, or if it is a web app deployed to a different web server from the web application that hosts the JSDO login session, you must specify `serviceURI` as an absolute URI to the Tomcat server domain or host and port, for example, `http://www.progress.com/SportsMobileApp`, or perhaps for testing, `http://localhost:8980/SportsMobileApp`.

If the mobile app from which you are logging in is a web app deployed to the same Apache Tomcat server as the web application that hosts the JSDO login session, you can specify `serviceURI` as a relative URI, for example, `/SportsMobileApp`, which is relative to the deployment end point (Tomcat server domain or host and port).

- **`catalogURI`** — (Required) A `string` expression that specifies the URI of a Data Service Catalog file. This URI can specify a location in a web application running on a web server. It is typically the location of the web application that hosts the Data Object Service where the returned `JSDOSession` object has a JSDO login session.

  If the Catalog is deployed to the same web server (or device) that the app UI is loaded from, `catalogURI` can be specified using a relative path, for example, `/SportsMobileApp/static/OrderEntrySvc.json`, where `/SportsMobileApp` is the relative URI of the web application that hosts the Catalog. Otherwise, it must be specified using an absolute path that includes the Tomcat server domain or host and port, for example, `http://www.progress.com:8980/SportsMobileApp/static/OrderEntrySvc.json`, or perhaps for testing, `http://localhost:8980/SportsMobileApp/static/OrderEntrySvc.json`.

  Note: The default Catalog URI for a Catalog created for an OpenEdge Data Object Service, relative to the Apache Tomcat server domain or host and port where the session is logged in, is the following: */WebApplicationName*/static/*ServiceName*.json, where *WebApplicationName* is the name of the web application and *ServiceName* is the name of the Data Object Service for which the Data Service Catalog is created.

- **`authenticationModel`** — (Required only for Basic or Form authentication) A `string` constant that specifies one of the three authentication models that the returned `JSDOSession` object supports, depending on the web application configuration:

  - **`progress.data.Session.AUTH_TYPE_ANON`** — (Default) The web application supports Anonymous access. No authentication is required.
  - **`progress.data.Session.AUTH_TYPE_BASIC`** — The web application supports HTTP Basic authentication and requires a valid username and password. To have the `JSDOSession` object manage access to the web application's resources, you need to pass these credentials using the **`username`** and **`password`** properties. Typically, you would require the user to enter their credentials into a login dialog provided by your mobile app, either using a form of your own design or using a template provided by Progress Software Corp.
  - **`progress.data.Session.AUTH_TYPE_FORM`** — The web application uses Form-based authentication. Like HTTP Basic, Form-based authentication requires user credentials for access to protected resources; the difference is that the web application itself sends a form to the client to get the credentials. However, when you have the `JSDOSession` object manage access to the web application's resources, you handle Form-based authentication the same way that you handle Basic—get the user's credentials using the **`username`** and **`password`** properties. The `JSDOSession` intercepts the form sent by the web application and handles the authentication without that form being displayed.

  If the web application requires authentication, you must set this value correctly to ensure that users can log in.

For more information on these authentication models and how to configure them for a web application, see the sections on configuring web server authentication models in the administration documentation for the server hosting your Data Object Service.

- **username** — (Required only for Basic or Form authentication) A `string` expression containing a user ID for the function to send to the web server for authentication.

  Note: The `userName` property of the returned `JSDOSession` object contains this value.

- **password** — (Required only for Basic or Form authentication) A `string` expression containing a password for the function to send to the web server to authenticate the specified user.
- **name** — (Optional) A `string` with an operative value that you define to enable page refresh support for the returned `JSDOSession` object. The operative value can be any value other than the empty string (`""`), `null`, or `undefined`.

  If this page refresh support is enabled and the function successfully logs the newly instantiated `JSDOSession` into its server web application, the object stores the state of its JSDO login session using this `name` property value as a key. Then, at any time after log in, if the user initiates a browser refresh on the web app page, the `JSDOSession` object automatically identifies and restores its login session using this value. This helps to ensure, after a page refresh, that the web app does not need to prompt the user again for credentials in order to re-establish its connection to the web application for this `JSDOSession`.

  If you do not specify an operative value for `name` (the default), or you specify a non-operative value, such as the empty string (`""`), `null`, or `undefined`, the `JSDOSession` is instantiated without this page refresh support.

  For more information on how a login session is restored for a `JSDOSession` object with page refresh support, see the Notes for the progress.data.JSDOSession class on page 124description.

  Note: If you pass the `getSession( )` function the same value for `name` as an existing `JSDOSession`, it will return a `JSDOSession` using that same key. Both of them have the potential to overwrite each other. You must ensure that you pass a unique `name` value for each call to `getSession( )` (or the `JSDOSession` constructor).

  Note: Page refresh supports **only** Anonymous and Form-based authentication. You cannot enable page refresh support when you set `authenticationModel` for a `JSDOSession` to `progress.data.Session.AUTH_TYPE_BASIC`; in this case, any setting of the `name` property is ignored. To enable page refresh support, you must set the `authenticationModel` property to either `progress.data.Session.AUTH_TYPE_ANON` or `progress.data.Session.AUTH_TYPE_FORM`.

  Note: When calling this `getSession( )` function, you have no need to call the `isAuthorized( )` method on the returned `JSDOSession` object in order to test that an authorized login session is established it to manage page refresh. This function automatically manages page refresh for any successful call by a web app.

Note: Similar to the `login( )` method on the `progress.data.JSDOSession` class, the `getSession( )` function always relies on the default web resource, `/static/home.html`, as the login target. In addition, if you are both using Basic authentication and the web app is running on an iOS device, this function waits up to four (4) seconds to time out before failing to authenticate a login session or load a Data Service Catalog.

Promise method signatures

jQuery Promise objects define methods that register a callback function with a specific signature. The callback signatures depend on the function that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that `getSession( )` returns:

Syntax:

```
promise.done( function ( JSDOsession , result ) )
promise.fail( function ( result , info ) )
promise.always( function ( JSDOsession , result | result , info ) )
```

Note: The `always( )` method is always passed what the `done( )` or the `fail( )` methods are passed.

promise

A reference to the Promise object that is returned as the value of the `getSession( )` function. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDOSession class on page 124.

JSDOsession

A reference to the `JSDOSession` object that was successfully created and returned by `getSession( )` with a valid JSDO login session and a single loaded Data Service Catalog.

result

An `integer` constant indicating the overall result of the call that can have one of the following values:

- **`progress.data.Session.SUCCESS`** — The `JSDOSession` was created, logged in, and a Catalog added successfully. You can use JSDOs to access any Data Object Services supported by the web application to which the `JSDOSession` object has logged in.
- **`progress.data.Session.AUTHENTICATION_FAILURE`** — Login failed because of an authentication error. This might be because the JSDO session login had invalid credentials or the authentication error happened when adding the Catalog.
- **`progress.data.Session.GENERAL_FAILURE`** — `JSDOSession` creation failed because of something other than an authentication error.

  Note: This value can also be returned if invalid user credentials triggered a login timeout using Basic authentication on an iOS device.

Note: It is not always necessary to test the value of *`result`* in a Promise method callback for `getSession( )`, especially if the callback is registered using *`promise`*`.done( )`, where the callback always executes with the same value (`progress.data.Session.SUCCESS`).

info

A JavaScript object that can have the following properties:

- **`errorObject`** — Any error object thrown as a result of sending a login or add Catalog request to the web server.

Note: If this object is thrown because of a timeout triggered according to the four (4) second maximum wait time to login or load a Catalog, the `message` property of the error object is set to `"login timeout expired"`.

- **`xhr`** — A reference to the XMLHttpRequest object that was used to make the server request to either log in or add a catalog.

General web server interaction

The general web server interaction with and response to this method depends on the authentication model that the web server uses and how resources are accessed and protected. You configure the authentication model for each web application deployed to the Apache Tomcat and specify both the web application URI and its corresponding authentication model to the `getSession( )` function.

For more information on these authentication models and how to configure them for a web application, see the sections on configuring web server authentication models in the server documentation for your Data Object Service.

Caution: You must be sure that security is configured to complete authentication before the application requests resources in the Data Service Catalog. Although it is possible to configure application security so that only the Data Object resources in the Catalog require authentication, Progress does not recommend this approach. Instead, Progress recommends that you require authentication for application resources in addition to those defined in the Catalog, and require that the authentication occur prior to accessing any resources in the Catalog. Once the user is authenticated, the web server provides access to all other resources, including Catalog resources, according to the user's authorization settings.

Example

The following code fragment calls `getSession( )` in a `try` block, handling successful results with the Promise `done( )` method, which invokes one inline function, and handling unsuccessful results with the Promise `fail( )` method, which invokes another inline function. For a successful call, it saves the returned `JSDOSession` and logs a message to the consule, then creates and initializes a JSDO for further processing. For an unsuccessful call, it only logs a message to the console. If the call to `getSession( )` itself throws an exception, this is handled in a `catch` block by logging the associated message to the console:

```
var promise,
    myjsdosession;

try {
    promise = progress.data.getSession({
        name: "mySession",
        authenticationModel: progress.data.Session.AUTH_TYPE_FORM,
        serviceURI: "https://someHost:8810/myWebApplication",
        catalogURI: "https://someHost:8810/myWebApplication/static/Customer.json",
        username: "restuser",
        password: "password"
    });
    promise.done( function( jsdosession, result ) {
        var jsdo;

        myjsdosession = jsdosession;
        console.log("Success on getSession().");

        // create a JSDO
        jsdo = new progress.data.JSDO({
                    name: 'MyCustomerBE'
        });
    });
    promise.fail( function( result, info ) {
        console.log("Failure from getSession()");
    });
}
catch(ex) {
    console.log("Exception: " + ex.message);
};
```

See also:

addCatalog( ) method (JSDOSession class) on page 159, login( ) method (JSDOSession class) on page 275, progress.data.JSDOSession class on page 124, serviceURI property on page 333, userName property on page 358

hasChanges( ) method

Returns `true` if JSDO memory contains any pending changes (with or without before-image data), and returns `false` if JSDO memory has no pending changes.

Return type: `boolean`

Applies to: progress.data.JSDO class on page 110

Syntax

```
hasChanges ( )
```

This method always returns `true` if any change to JSDO memory has marked a record object it contains as created, updated, or deleted.

This method always returns `false` if you execute it immediately after invoking any one of the following methods on the JSDO:

- `fill( )`
- `saveChanges( )`, if the `autoApplyChanges` property is also set to `true`
- `acceptChanges( )`
- `rejectChanges( )`

A typical use of this method is to determine if there are any changes in JSDO memory that you want to save to local storage using the JSDO `saveLocal( )` method.

Example

The following code fragment shows an example of how you might use `hasChanges( )` to decide how to save JSDO memory to local storage:

```
dataSet = new progress.data.JSDO( 'dsStaticData' );
dataSet.fill();
dataSet.autoApplyChanges = false;

/* Work done with the dataSet JSDO memory */
.
.
.
if (dataSet.hasChanges())
{
  dataSet.saveLocal(progress.data.JSDO.CHANGES_ONLY);
}
else
{
  dataSet.saveLocal(progress.data.JSDO.ALL_DATA);
}
```

The fragment first invokes the `fill( )` method on a JSDO (`dataSet`) to load JSDO memory, sets the `autoApplyChanges` property on the JSDO to not automatically accept or reject changes saved to the server based on the success or failure of the save, and after a certain amount of work is done with the JSDO, decides to save **all** the data in JSDO memory to the default local storage area, or to save **only** the pending changes, based on whether any pending changes currently exist in JSDO memory.

See also:

acceptChanges( ) method on page 153, autoApplyChanges property on page 202, fill( ) method on page 220, hasData( ) method on page 255, rejectChanges( ) method on page 305, saveChanges( ) method on page 314, saveLocal( ) method on page 330

hasData( ) method

Returns `true` if record objects can be found in any of the tables referenced in JSDO memory (with or without pending changes), or in only the single table referenced on the JSDO, depending on how the method is called; and returns `false` if no record objects are found in either case.

Return type: `boolean`

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Syntax

```
jsdo-ref.hasData ( )
jsdo-ref.table-ref.hasData ( )
```

jsdo-ref

A reference to the JSDO. If you call the method on *jsdo-ref*, the method verifies if any data is available in the JSDO, whether it is created for a single-table or a multi-table resource.

table-ref

A table reference on the JSDO. If you call the method on *table-ref*, the method verifies if any data is available in the referenced table. If the `useRelationships` property is `true`, this includes related data in any other JSDO table(s) with which the referenced table has a parent-child relationship in the JSDO.

This method always returns `true` immediately after the `fill( )` method successfully loads JSDO memory with one or more record objects.

Three cases where this method returns `false` include when executing this method:

1. After a JSDO is instantiated but before `fill( )` or any other method (such as `addRecords( )`) has been invoked to load its JSDO memory with records
2. After the `fill( )` method completes successfully, but returns no records, possibly because none match the specification of its *filter* parameter
3. After the `saveChanges( )` method completes successfully on a JSDO, where its `autoApplyChanges` property set to `true` and all the records in the specified JSDO, or its table reference, were marked for deletion

Two typical uses of this method include determining if there is any data in JSDO memory that you might want to save in local storage using the JSDO `saveLocal( )` method, or that you might not want to lose by replacing JSDO memory, using the JSDO `readLocal( )` method, with other data previously saved in local storage.

Example

The following code fragment shows an example of how you might use `hasData( )` to decide when to save JSDO memory to local storage:

```
dataSet = new progress.data.JSDO( 'dsStaticData' );
dataSet.fill();

/* Work done with the dataSet JSDO memory */
.
.
.
if (dataSet.hasData())
{
  dataSet.saveLocal();
}
```

The fragment first invokes the `fill( )` method on a JSDO (`dataSet`) to load JSDO memory, and after a certain amount of work is done with the JSDO, decides to save all the data in JSDO memory to the default local storage area when it finds that records exist in JSDO memory to save.

See also:

addRecords( ) method on page 175, fill( ) method on page 220, hasChanges( ) method on page 253, saveChanges( ) method on page 314, readLocal( ) method on page 304, saveLocal( ) method on page 330

invalidate( ) method

Note: Applies to Progress Data Objects Version 4.4.1 or later.

Terminates the login session managed by the current `JSDOSession` object and permanently disables the object, rendering it unable to start a new login session.

This method throws an error if the underlying communication layer throws an error.

On a successful invalidate, the `JSDOSession` object sets its `connected` property to `false`. If the invalidate fails to logout the session because it cannot access the server, the object still sets its `connected` property to `false` because it is no longer available to manage any login session.

Once `invalidate( )` is executed, no further communication can occur between the mobile app and the server using this `JSDOSession` object. Thus, any call made to its `addCatalog( )`, `isAuthorized( )`, `login( )`, or `ping( )` method, or any attempt to instantiate a `progress.data.JSDO` associated with this `JSDOSession` object, throws an exception. In addition, any calls to methods on an existing `JSDO` instance that communicate with the server associated with this `JSDOSession` throw an exception as well.

This method is always executed asynchronously and returns results in callbacks that you register using methods of a Promise object returned as the method value.

Note: In order to invoke this method successfully, jQuery Promises must be supported in your development environment. Otherwise, the method throws an exception.

Return type: jQuery Promise

Applies to: progress.data.JSDOSession class on page 124

Syntax

```
invalidate ( )
```

Promise method signatures

jQuery Promise objects define methods that register a callback function with a specific signature. The callback signatures depend on the method that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that `invalidate( )` returns:

Syntax:

```
promise.done( function ( session , result , info ) )
promise.fail( function ( session , result , info ) )
promise.always( function ( session , result , info ) )
```

promise

A reference to the Promise object that is returned as the value of the `invalidate( )` method. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDOSession class on page 124.

session

A reference to the `JSDOSession` object on which `invalidate( )` was called.

result

A constant indicating the overall result of the call that can have one of the following values:

- **`progress.data.Session.SUCCESS`** — The invalidate of the `JSDOSession` object completed successfully.
- **`progress.data.Session.GENERAL_FAILURE`** — The invalidate failed because of some error.

Note: It is not always necessary to test the value of *`result`* in a Promise method callback for the `invalidate( )` method, especially if the callback is registered using *`promise`*`.done( )` or *`promise`*`.fail( )`, where the callback always executes with the same value (`progress.data.Session.SUCCESS` and `progress.data.Session.GENERAL_FAILURE`, respectively).

---

info

A JavaScript object that can have the following properties:

- **`errorObject`** — An error object thrown while attempting to terminate the login session.
- **`xhr`** — A reference to the XMLHttpRequest object sent to the web server to terminate the login session.

Example

The following code fragment calls the `invalidate( )` method on the `JSDOsession` object, `session`. This example uses `try` and `catch` blocks in case `invalidate( )` throws an unexpected exception, and displays messages returned from Promise handlers accordingly:

```
var serviceURI = "http://oemobiledemo.progress.com/OEMobileDemoServicesForm/",
session = new progress.data.JSDOSession({
            "authenticationModel": progress.data.Session.AUTH_TYPE_FORM,
            "serviceURI": serviceURI
    });

try { // Login, add catalogs, create additional JSDO instances,
      // and do JSDO tasks for this session, ...then(...) ...
    session.invalidate()
    .then(function(session, result, info) {
        // Success handler of invalidate
        return "Employee session invalidated successfully";
    }, function(session, result, info) {
        // Failure handler of invalidate
        if ( result === progress.data.Session.GENERAL_FAILURE ) {
            if ( info.errorObject ) {
                // Process error object thrown during invalidate...
            }
            if ( info.xhr ) {
                // Process XHR sent during invalidate...
            }
            return "Employee session invalidate failed. Unspecified error.";
        }
        else {
            return "Unexpected failed invalidate result";
        }
    })
    .then(function (msg) {
        console.log(msg);
    });
}
catch(ex) {
    console.log("There was an unexpected error from attempted invalidate: " + ex);
}
```

See also:

invalidateAllSessions( ) stand-alone function on page 258, logout( ) method (JSDOSession class) on page 286, connected property on page 216

invalidateAllSessions( ) stand-alone function

Note: Applies to Progress Data Objects Version 4.4.1 or later.

A stand-alone function that invalidates all current `progress.data.JSDOSession` instances that were created and initialized using the `progress.data.getSession( )` stand-alone function.

This function combines the features of:

- Finding all existing `JSDOSession` instances created using the `getSession( )` stand-alone function.

- Calling the `invalidate( )` method on each instance, which in turn calls the `logout( )` method on the instance.

After `invalidateAllSessions( )` executes, any call made to the `addCatalog( )`, `isAuthorized( )`, `login( )`, or `ping( )` method on an existing `JSDOSession` instance, or any attempt to instantiate a `progress.data.JSDO`, throws an exception. In addition, calls to any method on an existing `JSDO` instance that communicates with the server associated with an invalidated `JSDOSession` throws an exception as well.

This method is always executed asynchronously and returns results in callbacks that you register using methods of a Promise object returned as the function value.

Note: In order to invoke this function successfully, jQuery Promises must be supported in your development environment. Otherwise, the function throws an exception.

Note: This function does not support proxy servers (servers that function as a security service).

Return type: jQuery Promise

Applies to: The `progress.data` namespace

Syntax

```
progress.data.invalidateAllSessions ( )
```

Promise method signatures

jQuery Promise objects define methods that register a callback function with a specific signature. The callback signatures depend on the function that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that `invalidateAllSessions( )` returns:

Syntax:

```
promise.done( function ( result , info ) )
promise.fail( function ( result , info ) )
promise.always( function ( result , info ) )
```

Note: The `always( )` method is always passed what the `done( )` or the `fail( )` methods are passed.

promise

A reference to the Promise object that is returned as the value of the `invalidateAllSessions( )` function. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDOSession class on page 124.

result

An `integer` constant indicating the overall result of the call that can have one of the following values:

- **`progress.data.Session.SUCCESS`** — Invalidate of all `JSDOSession` instances has completed successfully.
- **`progress.data.Session.GENERAL_FAILURE`** — The invalidate failed because of some error.

Note: It is not always necessary to test the value of *result* in a Promise method callback for `invalidateAllSessions( )`, especially if the callback is registered using *promise*`.done( )`, where the callback always executes with the same value (`progress.data.Session.SUCCESS`).

info

A JavaScript object that can have the following properties:

- **`errorObject`** — An error object thrown while attempting to terminate any `JSDOSession` instance.
- **`xhr`** — A reference to the XMLHttpRequest object sent to the web server for the `JSDOSession` object that threw the error object.

Example

The following code fragment calls the `invalidateAllSessions( )` method to terminate and invalidate the login session for every current `JSDOSession` instance that is created using `progress.data.getSession( )`. This example uses `try` and `catch` blocks in case `invalidateAllSessions( )` throws an unexpected exception, and displays messages returned from Promise handlers accordingly:

```
var serviceURI = "http://oemobiledemo.progress.com/OEMobileDemoServicesForm/",
catalogURI =
"http://oemobiledemo.progress.com/OEMobileDemoServicesForm/static/CustomerService.json";

// create a JSDOSession for a specified web application
// that requires Form authentication
try {
    // create and initialize login session
    progress.data.getSession( {
            authenticationModel : progress.data.Session.AUTH_TYPE_FORM,
            serviceURI : serviceURI,
            catalogURI : catalogURI,
            username: "formuser",
            password: "formpassword"
    })
    .then(function( session, result ) {

        // Create a JSDO and fill it with data
        var jsdoCustomer = new progress.data.JSDO('Customer');
        return jsdoCustomer.fill()

    }, function(result, info) {
        console.log("getSession failed");
    })
    .then(function (jsdo, result, request) {
        console.log("Fill went okay!");
        // Do other JSDO related tasks here
    }, function (jsdo, result, request) {
        console.log("Not filling it today");
        // Do JSDO related failure stuff here
    })
    .always(function () {
        // By the time we get here, we should be done with all JSDO related stuff
        // and cleaning up
        return progress.data.invalidateAllSessions();
    })
    .then(function( result, info ) {
        console.log("All sessions are invalidated!");
    }, function() {
        console.log("A session failed validation!");
    });
} catch (ex) {
    console.log("Exception: " + ex.message);
}
```

Note that this example creates only a single login session. Any additional login sessions can be created, and associated JSDO tasks performed, anywhere appropriate in the same Promise chain prior to the Promise `always( )` method handler where `invalidateAllSessions( )` is invoked.

See also:

invalidate( ) method on page 256, logout( ) method (JSDOSession class) on page 286, getSession( ) stand-alone function on page 248, progress.data.JSDOSession class on page 124

invocation method

A custom method on the JSDO that executes an Invoke operation defined for a Data Object resource.

For an OpenEdge resource, this invocation method can be implemented as a corresponding ABL routine on an OpenEdge application server. This implementation can be any ABL routine that is defined in its Data Service Catalog with an Invoke operation type. The invocation method name can be the same as the ABL routine or an alias, as defined by the resource. The method passes any ABL input parameters as properties of an object parameter. The method returns results from the ABL routine, including any return value and output parameters, as properties of a request object that is returned by the method.

For a Rollbase object resource, this invocation method can be implemented as an API defined by the Rollbase server that reads and returns data from a related Rollbase object.

This method also causes an `offline` or `online` event to fire if it detects that there has been a change in the JSDO login session's online state.

This custom invocation method can be executed either synchronously, returning results in an object return value, or asynchronously, returning similar results in an `afterInvoke` event callback, depending on a parameter that you pass. You can also call the invocation method asynchronously using the JSDO `invoke( )` method, which returns results as a jQuery Promise. For more information on using `invoke( )`, see the description of the invoke( ) method on page 264.

Note: The results of an Invoke operation have no effect on JSDO memory. However, you can use the JSDO `addRecords( )` method to merge any record data that is returned by an invocation method into JSDO memory.

Return type: request object on page 146 or `undefined`

Applies to: progress.data.JSDO class on page 110

Syntax

```
op-name ( [ input-object ] [ , async-flag ] )
```

op-name

The name (specified as an identifier) of the invocation method as defined by the resource.

input-object

An object whose properties and values match the case-sensitive names and data types of the input parameters specified for the server routine that implements the invocation method. If the implementing routine does not take input parameters, specify `null` or leave out the argument entirely.

async-flag

A `boolean` that when `true` causes the method to execute asynchronously and when `false` causes the method to execute synchronously. The default value is `true`.

Returning and handling results

For a synchronous invocation, the method returns, as its value, a request object that contains several properties depending on the status of the Invoke operation.

For an asynchronous invocation, the method return value is `undefined`, but returns a similar request object as input to any event handler (callback) function subscribed to the following named events that fire in the following operational order:

1. beforeInvoke event on page 209
2. afterInvoke event on page 186

For a successful execution, if there are any output parameters or a method return value, they are returned as properties of an object referenced by the `response` property of the request object. For an OpenEdge resource, the referenced properties for output parameters match the case-sensitive names and data types of the output parameters defined for the implementing ABL routine. Any method return value is returned by a JSDO-defined `_retVal` property with a matching JavaScript data type. For a Rollbase resource, all successful execution results are returned using this JSDO-defined `_retVal` property, typically in a referenced object. For more information on data type mappings for an OpenEdge resource, see OpenEdge ABL to JavaScript data type mappings on page 363, and for a Rollbase resource, see Rollbase object to JavaScript data type mappings on page 365.

For an unsuccessful execution in an OpenEdge resource, the request object `response` property references an object that can contain a both a `_retVal` property, which is a `string` value with the error information, and an `_errors` property, which is an array with information on one or more errors. In a Rollbase resource, the request object `response` property is itself a `string` value containing the error information. For more information on error information returned in the response property, see the description of the response property on page 312.

Note: For an OpenEdge resource, if you are calling an invocation method that either sends a single table object or a ProDataSet object (with one or more tables) as a property of *`input-object`* or returns a single table object or ProDataSet object as a property of the `response` property object, you need to apply a rule in order to access this table or ProDataSet object. The rule is that wherever you de-reference or reference a table or ProDataSet object, you must reference that value twice, separated by a period or a colon, depending on the context. The reason is that the table or ProDataSet name is both the name of the parameter defined for the ABL routine and also the name of the JavaScript object containing the JSON data returned from the server. For example, to access a table object, `ttCust` returned by the `response` property in a `request` object, you must code the following de-reference: `request.response.ttCust.ttCust`. Similarly, if you pass `ttCust` to an invocation method, `InputTT( )`, you must code the following reference: `jsdo.InputTT( {ttCust: {ttCust:ttCust}});`

Note: For an OpenEdge resource, if the invocation method passes ProDataSet as an input or output parameter, that ProDataSet can contain before-image data. However, the invocation method does no processing of the before-image data in any way. You must therefore manage the object appropriately. For example, you can use an output ProDataSet containing before-image data as a *`merge-object`* parameter to the `addRecords( )` method as long as your JSDO is created with the same ProDataSet schema and the OpenEdge resource supports before-imaging.

Note: For a Rollbase resource, each specified invocation method returns table records from a Rollbase object that is related to the resource-defined Rollbase object. For example, the resource might have an invocation method named `getOrders( )` on a `Customers` resource. So, to return records from a Rollbase `Orders` object that is related to the resource-defined `Customers` object, you can make the following reference on the `request` object returned by `getOrders( )`: `request.response.Orders`.

Example

The following code fragment shows an invocation method (`getOrderTotalAndStatus( )`) called asynchronously on a JSDO as defined for an OpenEdge resource (`dsCustomerOrder`), with results returned in the callback (`onAfterInvokeGetOrderTotalAndStatus`) subscribed to the `afterInvoke` event:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );
dataSet.subscribe( 'afterInvoke', 'getOrderTotalAndStatus',
                   onAfterInvokeGetOrderTotalAndStatus );

dataSet.getOrderTotalAndStatus( paramObject ); // An asynchronous call is the default

function onAfterInvokeGetOrderTotalAndStatus( jsdo , success , request )
    if (success) {

        var response = request.response;
        var ordTotal = response._retVal;
        var ordStatus = response.pcStatus;
        /* process successful results */
        . . .

    }
    else {
        if (request.response && request.response._errors &&
            request.response._errors.length > 0) {

            var lenErrors = request.response._errors.length;
            for (var idxError=0; idxError < lenErrors; idxError++) {

                var errorEntry = request.response._errors[idxError];
                var errorMsg = errorEntry._errorMsg;
                var errorNum = errorEntry._errorNum;
                /* handle error */
                . . .
            }
        }
    }

};
```

Note that the callback is subscribed only for execution of `getOrderTotalAndStatus( )` and the call to `getOrderTotalAndStatus( )` is asynchronous by default, with its input parameters passed in `paramObject`. For a successful execution in the callback, the order total is returned as the method return value in the JSDO-defined `_retVal` property, and the one output parameter with the order status is returned as the value of the ABL-defined `pcStatus` property, both of which are returned in the `response` property object. For an unsuccessful execution, the callback inspects the `response` property to return one or more error messages in an array of objects.

See also:

addRecords( ) method on page 175, invoke( ) method on page 264, fill( ) method on page 220, record property on page 305, request object on page 146, saveChanges( ) method on page 314

invoke( ) method

Asynchronously calls a custom invocation method on the JSDO to execute an Invoke operation defined by a Data Object resource.

Both OpenEdge and Rollbase resources can define and implement custom invocation methods that you can call using this method.

The asynchronous execution of the specified invocation method using `invoke( )` returns results using a jQuery Promise. You can also directly call an invocation method on the JSDO either synchronously, returning results as its return value, or asynchronously, returning results using a callback subscribed to the `afterInvoke` event.

Note: In order to call `invoke( )` successfully, jQuery Promises must be supported in your development environment. Otherwise, the method throws an exception.

For more information on the possible implementations for custom invocation methods, and how to call them directly on the JSDO, see the description of the invocation method on page 262.

Note: The results of an Invoke operation have no effect on JSDO memory. However, you can use the JSDO `addRecords( )` method to merge any record data that is returned by an invocation method into JSDO memory.

Return type: jQuery Promise

Applies to: progress.data.JSDO class on page 110

Syntax

```
invoke ( op-name [ , input-object ] )
```

op-name

A `string` that specifies the name of the invocation method as defined by the resource.

input-object

An object whose properties and values match the case-sensitive names and data types of the input parameters specified for the server routine that implements the invocation method. If the implementing routine does not take input parameters, specify `null` or leave out the argument entirely.

Promise method callback signatures

jQuery Promise objects define methods that register a callback function with a specific signature. The callback signatures depend on the method that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that `invoke( )` returns:

Syntax:

```
promise.done( function ( jsdo , success , request ) )
promise.fail( function ( jsdo , success , request ) )
promise.always( function ( jsdo , success , request ) )
```

promise

A reference to the Promise object that is returned as the value of the `invoke( )` method. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDO class on page 110.

jsdo

A reference to the JSDO that invoked the `invoke( )` method (Invoke operation) on its resource. For more information, see the description of the jsdo property on page 272 of the request object.

success

A `boolean` that is `true` if the Invoke operation was successful. For more information, see the description of the success property on page 350 of the request object.

Note: It is not always necessary to test the value of *`success`* in a Promise method callback for the `invoke( )` method, especially if the callback is registered using *`promise`*`.done( )` and *`promise`*`.fail( )`, where the callback executes according to this value.

---

request

A reference to the request object returned after the `invoke( )` method completed execution and returned any results from its Invoke operation on the server. For more information, see the description of the request object on page 146.

Returning and handling results

This method returns results asynchronously in two different ways, and in the following order, depending on the development environment used to build the mobile app:

1. **Using JSDO named events for all environments** — The following events fire before and after the `invoke( )` method executes, respectively, with results handled by callback functions that you subscribe as documented for each event:
   a. beforeInvoke event on page 209
   b. afterInvoke event on page 186
2. **Using a Promise object returned for environments that support jQuery Promises** — Any callbacks that you register using Promise object methods all execute both **after** the `invoke( )` method itself and **after** any subscribed `afterInvoke` event callbacks for the *`op-name`*-specified invocation method complete execution. Note that the signatures of all Promise method callbacks match the signature of the `afterInvoke` event callback function so you can specify an existing `afterInvoke` event callback as the callback function that you register using any Promise method.

Because the callbacks that you register with any returned Promise methods execute only after all subscribed `afterInvoke` event callbacks for the *`op-name`*-specified invocation method complete execution, you can invoke logic in the Promise method callbacks to modify any processing done by the event callbacks.

If the `invoke( )` method completes successfully, the *`success`* parameter for any `afterInvoke` event callback or Promise method and the `success` property of each handler's *`request`* parameter object are both set to `true`. Otherwise, both the *`success`* parameter and `success` property are set to `false`, and you can read any error results by inspecting the setting of the `response` property in the same *`request`* parameter object.

The request object returned as an input parameter to any subscribed event callback or to any registered Promise object callback contains several properties depending on the status of the Invoke operation.

For a successful execution, if there are any output parameters or a method return value, they are returned as properties of an object referenced by the `response` property of the request object. For an OpenEdge resource, the referenced properties for output parameters match the case-sensitive names and data types of the output parameters defined for the implementing ABL routine. Any method return value is returned by a JSDO-defined `_retVal` property with a matching JavaScript data type. For a Rollbase resource, all successful execution results are returned using this JSDO-defined `_retVal` property, typically in a referenced object. For information on data type mappings for an OpenEdge resource, see OpenEdge ABL to JavaScript data type mappings on page 363, and for a Rollbase resource, see Rollbase object to JavaScript data type mappings on page 365.

For an unsuccessful execution in an OpenEdge resource, the request object `response` property references an object that can contain a both a `_retVal` property, which is a `string` value with the error information, and an `_errors` property, which is an array with information on one or more errors. In a Rollbase resource, the request object `response` property is itself a `string` value containing the error information. For more information on error information returned in the response property, see the description of the response property on page 312.

For more information on working with the results of invocation methods that you call with `invoke( )`, see the notes on handling results in the description of the invocation method on page 262.

Example

The following code fragment shows `invoke( )` calling an invocation method (`getOrderTotalAndStatus( )`) as defined for an OpenEdge resource (`dsCustomerOrder`), with asynchronous results returned in appropriate callbacks registered by methods of the returned Promise object:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );

dataSet.invoke( "getOrderTotalAndStatus" , paramObject ).done(
    function( jsdo, success, request ) {

        var response = request.response;
        var ordTotal = response._retVal;
        var ordStatus = response.pcStatus;
        /* process successful results */
        . . .

    }).fail(
    function( jsdo, success, request ) {
        if (request.response && request.response._errors &&
            request.response._errors.length > 0) {

            var lenErrors = request.response._errors.length;
            for (var idxError=0; idxError < lenErrors; idxError++) {

                var errorEntry = request.response._errors[idxError];
                var errorMsg = errorEntry._errorMsg;
                var errorNum = errorEntry._errorNum;
                /* handle error */
                . . .
            }
        }
    });
```

Note that for the call to `getOrderTotalAndStatus( )` in the previous example, its input parameters are passed in `paramObject`, as required by the implementing ABL routine. Depending on the success of the `getOrderTotalAndStatus( )` call, the appropriate callback registered for the Promise executes, making it unnecessary to test its `success` parameter.

Thus, when the callback registered by the Promise's `done( )` method executes for a successful `getOrderTotalAndStatus( )` call, the order total is returned as the method return value in the JSDO-defined `_retVal` property, and the one output parameter with the order status is returned as the value of the ABL-defined `pcStatus` property, both of which are returned in the `response` property object. When the callback registered by the Promise's `fail( )` method executes for an unsuccessful `getOrderTotalAndStatus( )` call, the callback inspects the `response` property to return one or more possible error messages in an array of objects.

The following code fragment shows the `invoke( )` method calling a `getOrders( )` invocation method to return record objects from a Rollbase object that is related to the Rollbase resource object (`Customers`), with results similarly returned using a Promise object:

```
rbCustomers = new progress.data.JSDO( 'Customers' );

/* Initialize paramObject with a customer identity */
. . .

rbCustomers.invoke( "getOrders" , paramObject ).done(
    function( jsdo, success, request ) {

        var response = request.response;
        var tblOrders = response._retVal;
        /* process successful results */
        . . .

    }).fail(
    function( jsdo, success, request ) {
        if (request.response) {

            var errorMsg = request.response;
            /* handle error */

        }
    });
```

Note that for the call to `getOrders( )`, any input parameters are passed in `paramObject`, as required by the implementing Rollbase API to identify a customer to return record objects from the related `Orders` Rollbase object. Depending on the success of the `getOrders( )` call, the appropriate callback registered for the Promise executes.

Thus, when the callback registered by the Promise's `done( )` method executes for a successful `getOrders( )` call, a table object containing the record objects from the resource-related `Orders` Rollbase object is returned as the method return value in the JSDO-defined `_retVal` property, which is returned in the `response` property object. When the callback registered by the Promise's `fail( )` method executes for an unsuccessful `getOrders( )` call, the callback reads the `response` property to return any single error message as a `string` value.

See also:

addRecords( ) method on page 175, fill( ) method on page 220, invocation method on page 262, record property on page 305, request object on page 146, saveChanges( ) method on page 314

isAuthorized( ) method

Note: Applies to Progress Data Objects Version 4.3 or later.

Determines if the current JSDOSession object has authorized access to the web application specified by its serviceURI property setting.

This method also causes an offline or online event to fire if it detects that there has been a change in the JSDOSession object's online state.

This method is always executed asynchronously and returns results in callbacks that you register using methods of a Promise object returned as the method value.

Return type: jQuery Promise

Applies to: progress.data.JSDOSession class on page 124

Syntax

```
isAuthorized ( )
```

Promise method signatures

jQuery Promise objects define methods that you can call to register a callback function with a specific signature. The callback signatures depend on the method that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that isAuthorized( ) returns:

Syntax:

```
promise.done( function ( session , result , info ) )
promise.fail( function ( session , result , info ) )
promise.always( function ( session , result , info ) )
```

promise

A reference to the Promise object that is returned as the value of the isAuthorized( ) method. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDOSession class on page 124.

session

A reference to the JSDOSession object on which isAuthorized( ) was called.

result

One of the following string constant values depending on the JSDOSession authorization or online status identified by isAuthorized( ):

- **progress.data.Session.SUCCESS** — An authentication appropriate to the JSDOSession object's authenticationModel setting previously succeeded and is currently sufficient to give access to the web application specified by its serviceURI setting.
- **progress.data.Session.AUTHENTICATION_FAILURE** — An authentication appropriate to the JSDOSession object's authenticationModel setting previously succeeded, but the JSDOSession is no longer authorized to access its web application, for example, because the login session, established using HTTP Form authentication, has expired.
- **progress.data.Session.LOGIN_AUTHENTICATION_REQUIRED** — No successful authentication was completed for this JSDOSession object (there was no login session), or if

there was a login session established, the `JSDOSession` has terminated its access to its web application by logging out.

Note: This value is also returned when invoking the method after a page refresh if page refresh support is not enabled for the `JSDOSession`. For more information, see Additional information on page 270.

- **`progress.data.Session.GENERAL_FAILURE`** — An error resulted from trying to contact its web application. You might find more information by consulting the *`info`* callback parameter.

The authorization status (*`result`* parameter value) is determined by whether the `JSDOSession` object can contact its web application to see if it is allowed access. If a login session was never previously authenticated, the `JSDOSession` never even tries to contact its web application. This communication also depends on the `authenticationModel` setting passed to the object's constructor.

`info`

A JavaScript object that can have the following properties:

- **`xhr`** — A reference to the XMLHttpRequest object, if any, sent to the web server to make the authorization request to the web application. If no request was made, this property is undefined.
- **`offlineReason`** — A `string` constant that is set only if `isAuthorized( )` determines that the `JSDOsession` object is disconnected from its web application or its associated application server. The returned value indicates the reason for the `JSDOsession` offline state. Possible values include some of those returned by a call to the `ping( )` method, as follows:
  - **`progress.data.Session.DEVICE_OFFLINE`** — The device itself is offline. For example, it might be in airplane mode, or it might be unable to pick up a Wi-Fi or cell signal.
  - **`progress.data.Session.SERVER_OFFLINE`** — The web server is not available. For a Rollbase Data Object Service, this is the web server for the public or private cloud. For an OpenEdge Data Object Service, this is the Tomcat Java servlet container.
  - **`progress.data.Session.WEB_APPLICATION_OFFLINE`** — The server is running, but the Java web application that implements the Data Object Service is not deployed.

Additional information

The behavior of `isAuthorized( )` in a web app after a browser page refresh depends on whether page refresh support is enabled by passing an operative value for the `name` property in the `JSDOSession` object constructor.

For example, if page refresh support has **not** been enabled, regardless of the authentication model, after the user initiates a page refresh, but before a **new** login session has been established by calling the `login( )` method, the callback *`result`* parameter is always set to `progress.data.Session.LOGIN_AUTHENTICATION_REQUIRED`. For more information on page refresh support, see the description of the progress.data.JSDOSession class on page 124 constructor.

Note: If the `JSDOSession` authentication model is HTTP Basic, page refresh support is never enabled.

Example

The following code fragment shows how you can use `isAuthorized( )` to check the online state of a given `JSDOSession` object with page refresh support enabled in the `JSDOSession` constructor:

```
$(document).ready(function(){
    if (!ablSession) {
        // Create a session if it does not yet exist
        var ablSession = new progress.data.JSDOSession(
             {  serviceURI: "http://localhost:8810/FormFCS",
                authenticationModel: progress.data.Session.AUTH_TYPE_FORM,
                name: "ablSessionKey"   // enable page refresh support
             } );
    }

    // Check whether there had been a login and we are still authorized
    ablSession.isAuthorized()
    .done( function(session, result, info) {
        startApp(); // start web app
    })
    .fail( function(session, result, info) {
        switch (result) {
            case progress.data.Session.LOGIN_AUTHENTICATION_REQUIRED:
            case progress.data.Session.AUTHENTICATION_FAILURE:
                showLogin(); // login, then start web app
                break;
            case progress.data.Session.GENERAL_FAILURE:
                // possibly some sort of offline problem
                if (info.offlineReason) {
                    alert("Offline: " + info.offlineReason);
                }
                break;
        }
    });
```

This web app example initially verifies if the instance variable (`ablSession`) for a `JSDOSession` already references an instance and creates the instance if it does not already exist, with HTTP Form authentication and page refresh enabled.

It then immediately calls `isAuthorized( )` to determine if an existing login session is already established from a restored prior session state. If it is established and authorized to access its web application, the `done( )` callback executes calling `startApp()`, which is a method that continues web app execution using the restored JSDO login session. If it is not established or authorized (likely for a first-time execution or prior logout), the `fail( )` callback executes.

If the `fail( )` callback executes either because initial authentication for the `JSDOSession` has not been done or authorization for a restored `JSDOSession` failed, it calls `showLogin()`, which is a method that prompts the user for credentials to call the `login( )` method on the `JSDOSession` and then perhaps call `startApp()` (or the equivalent) if the new session login succeeds.

If the `fail( )` callback executes for some other reason, it displays any available offline status (`info.offlineReason`) that might be the cause of the failure. Although not shown, it can also inspect the returned XMLHttpRequest object (`info.xhr`) for more information on the failure.

See also:

name property (JSDOSession class) on page 291, login( ) method (JSDOSession class) on page 275, offline event on page 292, online event on page 294

jsdo property

An object reference to the JSDO that performed the operation returning the request object.

Data type: progress.data.JSDO class on page 110

Access: Read-only

Applies to: request object on page 146

The `jsdo` property is available for all JSDO events. This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe or register to JSDO events, any returned jQuery Promises, and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO. This object is also returned as the value of any JSDO invocation method that you execute synchronously.

See also:

fill( ) method on page 220, invocation method on page 262, invoke( ) method on page 264, saveChanges( ) method on page 314

JSDOs property

Returns an array of JSDOs that use the current `JSDOSession` or `Session` object to communicate with their Data Object Services.

Data type: JSDO array

Access: Read-only

Applies to: progress.data.JSDOSession class on page 124, progress.data.Session class on page 136

See also:

pingInterval property on page 303, services property on page 331

jsrecord property

A reference to the record object that was created, updated, or deleted by the current record-change operation.

Data type: progress.data.JSRecord class on page 133

Access: Read-only

Applies to: request object on page 146

The `jsrecord` property is available only for the following JSDO events:

- `afterCreate`
- `afterDelete`
- `afterUpdate`

- `beforeCreate`
- `beforeDelete`
- `beforeUpdate`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe both to JSDO events and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO.

See also:

saveChanges( ) method on page 314

jsrecords property

An array of references to record objects that are created, updated, or deleted on the server for a Submit operation (invoking `saveChanges(true)`) on a Data Object resource that supports before-imaging.

Data type: Array of progress.data.JSRecord class on page 133

Access: Read-only

Applies to: request object on page 146

For a before-image resource, such as an OpenEdge ProDataSet resource, if the record change on a given `JSRecord` object referenced in this array has executed with an error, the `data` property object of this record object contains a property named `"prods:rowState"` that has one of the following `string` values indicating the type of record change that was applied to the record:

- `"created"`
- `"modified"`
- `"deleted"`

To access this property (with a special character (`:`) in its name), you must reference it as an index reference on the `data` property object, as in this example, where `request` is the request object returned from the Submit operation:

```
for (var i = 0;i < request.jsrecords.length;i++) {
    var jsrecord = request.jsrecords[i];
    var jsrecError = jsrecord.getErrorString();
    if (jsrecError) {
        /* handle record-change error */
        console.log("Record change: "
                    + jsrecord.data["prods:rowState"]);
        console.log("Error: " + jsrecError);
    }
}
```

Note: The `"prods:rowState"` property is always available in this `data` property object **only** if the JSDO `autoApplyChanges` property is `false` when you invoke `saveChanges(true)`.

The `jsrecords` property is available only for the following JSDO events or in the request object returned to a jQuery Promise callback, and **only** after invoking `saveChanges(true)` on the JSDO:

- `afterSaveChanges`
- `beforeSaveChanges`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe both to JSDO events and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO.

See also:

saveChanges( ) method on page 314

lastSessionXHR property

Returns an object reference to the XMLHttpRequest object (XHR) that was most recently used by the `progress.data.Session` object to execute a `Session` object method.

The one exception is in the case of a successful invocation of the `logout( )` method, in which case `lastSessionXHR` is set to `null`.

Note: This does not include the XMLHttpRequest objects that a `Session` object helps to prepare for JSDO server requests (Data Object operations).

Data type: `Object`

Access: Read-only

Applies to: progress.data.Session class on page 136

It is possible for a `Session` object method to fail prior to sending a request to the web server. This is especially true of the `addCatalog( )` method. If a `Session` object method fails before sending the request, `lastSessionXHR` is the most recent XHR returned from a previous method call that did send a request to the web server. For example, if `login( )` fails with an authentication error on the web server, and the application follows with a call to `addCatalog( )`, `addCatalog( )` throws an error. However, `lastSessionXHR` then returns the XHR used for the unsuccessful `login( )` request instead of for the failed `addCatalog( )` call, because `addCatalog( )` never attempts to send its own request and therefore doesn't create an XHR.

In general, you can use the XHR returned by this property to find out more information about the results of a web server request than you can identify from the error code returned, or the error object thrown, by a given `Session` object method. One possible scenario is that the request to the server can succeed, but the body of the response, which should contain the catalog, contains data that cannot be parsed successfully as a Data Service Catalog.

Example

The following code fragment might provide more information about what has gone wrong when a possibly invalid Data Service Catalog is loaded for a session:

```
// create Session
pdsession = new progress.data.Session();

// log in
pdsession.login('http://testmach:8980/SportsMobileApp');

// load catalog

try {
  pdsession.addCatalog(
    "http://testmach:8980/SportsMobileApp/static/CustomerSvc.json");
}
catch(e) {
  var xhr = pdsession.lastSessionXHR;
  if ( xhr ) {
    if (xhr.status === 200 ) { // did HTTP request succeed?
      // probably something wrong with the catalog, dump it
      if (xhr.responseText) {
        console.log(xhr.responseText);
      }
    }
  }
}
```

See also:

addCatalog( ) method (Session class) on page 164, login( ) method (Session class) on page 280, logout( ) method (Session class) on page 288

login( ) method (JSDOSession class)

Note: Updated for Progress Data Objects Version 4.4.1 and later.

Starts a JSDO login session in a web application for the current `JSDOSession` object by sending an HTTP request with specified user credentials to the web application URI specified in the object's constructor.

This method throws an error if the method arguments are invalid or if the underlying communication layer throws an error.

On a successful login, the `JSDOSession` object sets its `connected` property to `true`. If the login fails, the object leaves its `connected` property set to `false`.

This method is always executed asynchronously and returns results in callbacks that you register using methods of a Promise object returned as the method value.

Note: In order to invoke this method successfully, jQuery Promises must be supported in your development environment. Otherwise, the method throws an exception.

Note: Before invoking this method, ensure that you set the `authenticationModel` configuration property in the constructor of the `JSDOSession` object correctly (see the notes on authentication models).

Note: If the browser or mobile device has already authenticated a JSDO login session, this method completes successfully.

Note: This method does not support proxy servers (servers that function as a security service).

Note: As a recommended alternative to using this method on an existing `JSDOSession` object, you can create a new, initialized `JSDOSession` using the `progress.data.getSession( )` stand-alone function. In one call, this function instantiates the object, invokes its `login( )` method with specified credentials, then invokes its `addCatalog( )` method to load a specified Data Service Catalog. For more information, see the getSession( ) stand-alone function on page 248 description.

Return type: jQuery Promise

Applies to: progress.data.JSDOSession class on page 124

Syntax

```
login ( [ username , password ] [ , parameter-object ] )
```

username

A `string` expression containing a user ID for the method to send to the web server for authentication.

Note: The `userName` property of the `JSDOSession` object returns the most recent value passed to this method for the current `JSDOSession` object.

password

A `string` expression containing a password for the method to send to the web server to authenticate the specified user.

parameter-object

An object that contains the following optional property:

- **`iOSBasicAuthTimeout`** — A `number` that specifies the time, in milliseconds, that the `login( )` method waits for a response before returning an error. This error might mean that the user entered invalid credentials. If you set this value to zero (0), no timeout is set, and `login( )` can wait indefinitely before returning an error. If you do not set the `iOSBasicAuthTimeout` property, `login( )` uses 4000 (4 seconds) as the default value.

  Note: Any non-zero timeout value (default or otherwise) for this *`parameter-object`* property operates **only** under certain conditions. Otherwise, any setting of this property has no effect. For more information, see the notes of the progress.data.JSDOSession class on page 124 description.

Note: The `login( )` method on the `progress.data.Session` class also takes a `serviceURI` property to specify the URI of the web application on which to start a JSDO login session and takes a `loginTarget` property to specify a protected web resource, other than the default, as a login target. The `login( )` method on `progress.data.JSDOSession` relies on the class constructor to specify the web application URI and always uses the default web resource, `/static/home.html`, as the login target.

Promise method signatures

jQuery Promise objects define methods that register a callback function with a specific signature. The callback signatures depend on the method that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that `login( )` returns:

Syntax:

```
promise.done( function ( session , result , info ) )
promise.fail( function ( session , result , info ) )
promise.always( function ( session , result , info ) )
```

promise

A reference to the Promise object that is returned as the value of the `login( )` method. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDOSession class on page 124.

session

A reference to the `JSDOSession` object on which `login( )` was called.

result

A constant indicating the overall result of the call that can have one of the following values:

- **`progress.data.Session.AUTHENTICATION_SUCCESS`** — The JSDO login session started successfully. You can use JSDOs to access any Data Object Services supported by the web application to which the `JSDOSession` object has logged in.
- **`progress.data.Session.AUTHENTICATION_FAILURE`** — JSDO login failed because of invalid user credentials (*`username`* or *`password`*).
- **`progress.data.Session.GENERAL_FAILURE`** — JSDO login failed because of a non-authentication failure.

  Note: This value can also be returned if invalid user credentials triggered a login timeout according to the value of the *`parameter-object`*`.iOSBasicAuthTimeout` property.

Note: It is not always necessary to test the value of *`result`* in a Promise method callback for the `login( )` method, especially if the callback is registered using *`promise`*`.done( )`, where the callback always executes with the same value (`progress.data.Session.AUTHENTICATION_SUCCESS`).

info

A JavaScript object that can have the following properties:

- **`errorObject`** — Any error object thrown as a result of sending a login request to the web server.

  Note: If this object is thrown because of a login timeout triggered according to the value of the *`parameter-object`*`.iOSBasicAuthTimeout` property, the `message` property of the error object is set to `"login timeout expired"`.

- **`xhr`** — A reference to the XMLHttpRequest object sent to the web server to start a JSDO login session.

You can also return the result of the most recent login attempt on the current `JSDOSession` object by reading its `loginResult` property. For a more specific status code returned in the HTTP response, you can also check the value of the `loginHttpStatus` property.

Note: You can log out from a web application and then log in again using the same `JSDOSession` object. The login will use the same `serviceURI` and `authenticationModel` settings originally passed to the constructor, but you must pass any required credentials each time `login( )` is called.

General web server interaction

The general web server interaction with and response to this method depends on the authentication model that the web server uses and how resources are accessed and protected. You configure the authentication model for each web application deployed to the Apache Tomcat and specify both the web application URI and its corresponding authentication model to the `JSDOSession` object constructor. For more information on the authentication models that a `JSDOSession` object supports, see the description of the constructor for the progress.data.JSDOSession class on page 124.

For more information on these authentication models and how to configure them for a web application, see the sections on configuring web server authentication models in the server documentation for your Data Object Service.

Caution: You must be sure that security is configured to complete authentication before the application requests resources in the Data Service Catalog. Although it is possible to configure application security so that only the Data Object resources in the Catalog require authentication, Progress does not recommend this approach. Instead, Progress recommends that you require authentication for application resources in addition to those defined in the Catalog, and require that the authentication occur prior to accessing any resources in the Catalog. Once the user is authenticated, the web server provides access to all other resources, including Catalog resources, according to the user's authorization settings.

Note: Unless the application design guarantees that the user will be prompted by the web browser or native device container to provide credentials before a `login( )` call occurs, Progress recommends (in some cases requires) that the mobile or web app pass the credentials as parameters to the `login( )` method. In addition, you must correctly pass the value of the `authenticationModel` configuration property to the `JSDOSession` object's constructor. Coding the mobile or web app in this way ensures that the proper credentials are submitted to the server and promotes a favorable user experience.

Example

The following code fragment calls the `login( )` method on the `JSDOsession` object, `empSession`. This example uses the callbacks registered by the Promise `done( )` and `fail( )` methods to check the result of the call along with any error object thrown as a result of the request, then assembles an appropriate message to display in an alert box. It also uses a `try-catch` block in case the `login( )` method throws an error object instead of sending the request to the server:

```
var msg;
var xhr;

try {
    empSession.login( userName, passWord ).done(
        function( session, result, info ) {
            msg = "Logged in successfully”;
        }).fail(
        function( session, result, info ) {
            if ( result
                 === progress.data.Session.AUTHENTICATION_FAILURE ) {
                msg = "Employee Login failed. Authentication error";
            }
            else if ( result
                      === progress.data.Session.GENERAL_FAILURE  ) {
                msg = "Employee Login failed. Unspecified error";
                if ( info.errorObject ) {
                    // Process error object thrown during login . . .
                }
                if ( info.xhr ) {
                    // Process XHR sent during login . . .
                }
            }
            xhr = info.xhr;
        });
}
catch(errObj) {
    msg = "Employee Login failed. Error attempting to call login";
    msg = msg + '\n' + errObj.message;
}

msg = msg +
          "\nloginResult: " + empSession.loginResult +
          "\nloginHttpStatus: " +  empSession.loginHttpStatus +
          "\nuserName: " +  empSession.userName +
          "\nLogin XHR: " +  xhr;

alert(msg);
```

See also:

addCatalog( ) method (JSDOSession class) on page 159, authenticationModel property (JSDOSession class) on page 201, connected property on page 216, getSession( ) stand-alone function on page 248, loginHttpStatus property on page 284, loginResult property on page 285, logout( ) method (JSDOSession class) on page 286, offline event on page 292, online event on page 294, serviceURI property on page 333, userName property on page 358

login( ) method (Session class)

Note: Deprecated in Progress Data Objects Version 4.4 and later. Please use the login( ) method (JSDOSession class) on page 275 instead.

Starts a JSDO login session on the current `Session` object by sending an HTTP request with specified user credentials to the URI of a specified web application.

On a successful login, the `Session` object sets its `connected` property to `true`. If the login fails, the object leaves its `connected` property set to `false`.

This method can be called synchronously or asynchronously, depending on parameters that you pass.

Note: Before invoking this method, ensure that you set the `authenticationModel` property on the `Session` object correctly (see the notes on authentication models).

Note: If the browser or mobile device has already authenticated a JSDO login session, this method completes successfully.

Note: This method does not support proxy servers (servers that function as a security service).

Return type: `number`

Applies to: progress.data.Session class on page 136

Syntax

```
login ( service-uri [ , username , password [ , login-target ] ] )
login ( parameter-object )
```

Note: If you call `login( )` passing *`service-uri`*, the method executes synchronously. If you call it passing *`parameter-object`*, the method executes synchronously or asynchronously, depending on the setting of *`parameter-object`*.

`service-uri`

A `string` expression containing the URI of the web application for which to start the JSDO login session. This web application must support one or more Data Object Services in order to create JSDOs for the service resources provided by the application. If HTTP Basic Authentication is in effect for the web application (see the notes on authentication models), this URI is appended with a string that identifies a protected resource against which to authenticate the login session (see *`login-target`*).

If the mobile app from which you are logging in is a hybrid app that will be installed to run directly in a native device container, or if it is a web app deployed to a different web server from the web application that hosts the JSDO login session, you must specify *service-uri* as an absolute URI to the Tomcat server domain or host and port, for example, `http://www.progress.com/SportsMobileApp`, or perhaps for testing, `http://testmach:8980/SportsMobileApp`.

If the mobile app from which you are logging in is a web app deployed to the same Apache Tomcat server as the web application that hosts the JSDO login session, you can specify *service-uri* as a relative URI, for example, `/SportsMobileApp`, which is relative to the deployment end point (Tomcat server domain or host and port).

Note: Once the `login( )` method executes, the value you pass for *service-uri* also sets the value of the `sessionURI` property on the current `Session` object, whether or not JSDO login completes successfully.

---

username

A `string` expression containing a user ID for the method to send to the web server for authentication.

Note: The `userName` property of the `Session` object returns the most recent value passed to this method for the current `Session` object.

---

password

A `string` expression containing a password for the method to send to the web server to authenticate the specified user.

login-target

A `string` expression that when appended to *service-uri* specifies a web application resource against which the specified user is authenticated. If you do not specify a value for *login-target*, the value is set to `"/static/home.html"` by default.

Note: The value returned by the `loginTarget` property of the `Session` object is either the value of the `login-target` parameter or the default (`"/static/home.html"`).

---

parameter-object

An object that has one or more of the following properties:

- **`serviceURI`** — (Required) Same value as the *service-uri* parameter.
- **`userName`** — (Optional) Same value as the *username* parameter.
- **`password`** — (Optional) Same value as the *password* parameter.
- **`loginTarget`** — (Optional) Same value as the *login-target* parameter.
- **`async`** — (Optional) A `boolean` that, if `true`, tells `login( )` to execute asynchronously. If `false` or absent, `login( )` executes synchronously.
- **`iOSBasicAuthTimeout`** — (Optional) A `number` that specifies the time, in milliseconds, that the `login( )` method waits for a response before returning an error. This error might mean that the user entered invalid credentials. If you set this value to zero (0), no timeout is set, and

`login( )` can wait indefinitely before returning an error. If you do not set the `iOSBasicAuthTimeout` property, `login( )` uses 4000 (4 seconds) as the default value.

Note: Any non-zero timeout value (default or otherwise) for this *`parameter-object`* property operates **only** under certain conditions, **including** when `login( )` is executed asynchronously. Otherwise, any setting of this property has no effect. For more information, see the notes of the progress.data.Session class on page 136 description.

When the method completes, it returns one of the following numeric constants to indicate the result:

- **`progress.data.Session.ASYNC_PENDING`** — The method is executing asynchronously. All asynchronous execution results (including any server errors) are returned in the `afterLogin` event that is fired on the `Session` object after the method completes. Before calling the method, you can subscribe one or more handlers to this event using the `subscribe( )` method on the `Session` object.
- **`progress.data.Session.SUCCESS`** — JSDO login session started successfully. After you have downloaded Data Service Catalogs using the `addCatalog( )` method for supported Data Object Services, you can create JSDOs for each Data Object resource provided by these services.
- **`progress.data.Session.AUTHENTICATION_FAILURE`** — JSDO login failed because of invalid user credentials (*`username`* or *`password`*).
- **`progress.data.Session.GENERAL_FAILURE`** — JSDO login failed because of a non-authentication failure.

It is also possible for this method to throw an `Error` object, in which case it does not return a value at all. You can also return the result for the most recent login attempt on the current `Session` object by reading the `loginResult` property. For a more specific status code returned in the HTTP response, you can check the value of the `loginHttpStatus` property. For more detailed information about any response (successful or unsuccessful) returned from the web server, you can also check the XMLHttpRequest object (XHR) returned by the `lastSessionXHR` property.

Note: When called asynchronously, any handler for the `afterLogin` event is passed the same information that is available for the `login( )` method when it is called synchronously, including the `Session` object that executed `login( )` (with the same property settings) and any `Error` object that was thrown processing the server response.

The general web server interaction with and response to this method depends on the authentication model that the web server uses and how resources are accessed and protected. You configure the authentication model for each web application deployed to the Apache Tomcat.

The JSDO supports the following authentication models, depending on the web application configuration:

- **`progress.data.Session.AUTH_TYPE_ANON`** — The web application supports Anonymous access. No authentication is required. This is the default value if you do not set it.
- **`progress.data.Session.AUTH_TYPE_BASIC`** — The web application supports HTTP Basic authentication and requires a valid username and password. To have the `Session` object manage access to the web application's resources for you, you need to pass these credentials in a call to the `Session` object's `login( )` method. Typically, you would require the user to enter their credentials into a login dialog provided by your mobile app, either using a form of your own design or using a template provided by Progress Software.
- **`progress.data.Session.AUTH_TYPE_FORM`** — The web application uses HTTP Form-based authentication. Like HTTP Basic, Form-based authentication requires user credentials for access to protected resources; the difference is that the web application itself sends a form to the client to get the credentials. However, when you have the `Session` object manage access to the web application's resources, you

handle Form-based authentication the same way that you handle Basic—get the user's credentials yourself and pass them to the `login( )` method. The `Session` intercepts the form sent by the web application and handles the authentication without that form being displayed.

If the web application requires authentication, you must set this value in the `Session` object's `authenticationModel` property correctly to ensure that users can log in.

For more information on these authentication models and how to configure them for a web application, see the sections on configuring web server authentication models in the server documentation for your Data Object Service.

Caution: You must be sure that security is configured to complete authentication before the application requests resources in the Data Service Catalog. Although it is possible to configure application security so that only the Data Object resources in the Catalog require authentication, Progress does not recommend this approach. Instead, Progress recommends that you require authentication for application resources in addition to those defined in the Catalog, and require that the authentication occur prior to accessing any resources in the Catalog. (**Note:** This is the purpose of the *`login-target`* parameter, either one you pass to the `login( )` method or its default.) Once the user is authenticated, the web server provides access to all other resources, including Catalog resources, according to the user's authorization settings.

Note: Unless the application design guarantees that the user will be prompted by the web browser or native device container to provide credentials before a `login( )` call occurs, Progress recommends (in some cases requires) that the mobile app pass the credentials as parameters to the `login( )` method. In addition, you must correctly set the value of the `Session` object's `authenticationModel` property. Coding the mobile app in this way ensures that the proper credentials are submitted to the server and promotes a favorable user experience.

Caution: To help ensure that HTTP Forms access to web applications works in certain web browsers, such as Firefox, when the web application is configured for Cross-Origin Resource Sharing (CORS), always call the `login( )` method asynchronously.

Example

The following code fragment calls the `login( )` method synchronously on the session, `empSession` by omitting the `async` property from its object parameter. For a similar example with the method called asynchronously, see the reference entry for the `afterLogin` event. This synchronous example uses `try` and `catch` blocks to check for either an expected return value, an invalid return value (as part of a test), or a thrown `Error` object with an unknown error, then assembles an appropriate message to display in an alert box:

```
var retValue;
var msg;

try {
    retValue = empSession.login( {
        serviceURI : "http://testmach:8980/SportsMobileApp",
        userName : uname,
        password : pw }  );

    if ( retValue === progress.data.Session.LOGIN_AUTHENTICATION_FAILURE ) {
        msg = "Employee Login failed. Authentication error";
    }
    else if ( retValue === progress.data.Session.LOGIN_GENERAL_FAILURE  ) {
        msg = "Employee Login failed. Unspecified error";
    }
    else  if ( retValue === progress.data.Session.LOGIN_SUCCESS  ) {
        msg = "Logged in successfully”;
    }
    else {
        msg = "TEST ERROR! UNEXPECTED loginResult" + msg;
    }
}
catch(errObj) {
    msg = "Employee Login failed. Error attempting to call login";
    msg = '\n' + errObj.message;
}

msg = msg +
          "\nloginResult: " + empSession.loginResult +
          "\nloginHttpStatus: " +  empSession.loginHttpStatus +
          "\nuserName: " +  empSession.userName +
          "\nlastSessionXHR: " +  empSession.lastSessionXHR;

alert(msg);
```

See also:

addCatalog( ) method (Session class) on page 164, afterLogin event on page 188, authenticationModel property (Session class) on page 201, connected property on page 216, lastSessionXHR property on page 274, loginHttpStatus property on page 284, loginResult property on page 285, loginTarget property on page 285, logout( ) method (Session class) on page 288, offline event on page 292, online event on page 294, serviceURI property on page 333, subscribe( ) method (Session class) on page 349, userName property on page 358

loginHttpStatus property

Returns the specific HTTP status code returned in the response from the most recent login attempt on the current `JSDOSession` or `Session` object.

Data type: `number`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 124, progress.data.Session class on page 136

See also:

login( ) method (JSDOSession class) on page 275, login( ) method (Session class) on page 280

loginResult property

Returns the return value of the `login( )` method, which is the basic result code for the most recent login attempt on the current `JSDOSession` or `Session` object.

Data type: `number`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 124, progress.data.Session class on page 136

Possible `loginResult` values include the following numeric constants:

- **`progress.data.Session.LOGIN_SUCCESS`** — JSDO login session started successfully.
- **`progress.data.Session.LOGIN_AUTHENTICATION_FAILURE`** — JSDO login failed because of invalid user credentials.
- **`progress.data.Session.LOGIN_GENERAL_FAILURE`** — JSDO login failed because of a non-authentication failure.

For a more specific status code returned in the HTTP response, you can check the value of the `loginHttpStatus` property.

The value of this property is `null` prior to the first login attempt and after a successful logout (until the next login attempt).

See also:

login( ) method (JSDOSession class) on page 275, login( ) method (Session class) on page 280, loginHttpStatus property on page 284

loginTarget property

Returns the string appended to the web application URI passed to the `login( )` method to form the URI of an application resource against which the user has been authenticated for the current login session.

By default, this appended string is `"/static/home.html"`.

Data type: `string`

Access: Read-only

Applies to: progress.data.Session class on page 136

You initially provide the Mobile Web application URI as a parameter to the `login( )` method. You can also pass a parameter to this method to specify a non-default value for the string appended to this URI.

See also:

login( ) method (Session class) on page 280, serviceURI property on page 333

logout( ) method (JSDOSession class)

Note: Updated for Progress Data Objects Version 4.4.1 and later.

Terminates the login session on the web application managed by the current `JSDOSession` object and leaves the object available to start a new login session with a call to its `login( )` method.

This method throws an error if the underlying communication layer throws an error.

On a successful logout, the `JSDOSession` object sets its `connected` property to `false`. If the logout fails, the object leaves its `connected` property set to `true`, unless the failure happened because the app cannot access the server.

Once `logout( )` is executed, no further communication (other than a `login( )` call) can occur between the mobile app and the server using this `JSDOSession` object. However, any catalogs loaded in the object remain available to create and maintain JSDOs, though these JSDOs cannot make requests to the server until a new login session is established for the object (including for anonymous access).

This method is always executed asynchronously and returns results in callbacks that you register using methods of a Promise object returned as the method value.

Note: In order to invoke this method successfully, jQuery Promises must be supported in your development environment. Otherwise, the method throws an exception.

Note: If you do not need to reuse the current `JSDOSession` instance, you can call `invalidate( )` (instead of `logout( )`) to permanently disable the instance and prevent any additional calls to its `login( )` method. If you want to permanently disable all current `JSDOSession` instances, you can also call `invalidateAllSessions( )` instead.

Return type: jQuery Promise

Applies to: progress.data.JSDOSession class on page 124

Syntax

```
logout ( )
```

Promise method signatures

jQuery Promise objects define methods that register a callback function with a specific signature. The callback signatures depend on the method that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that `logout( )` returns:

Syntax:

```
promise.done( function ( session , result , info ) )
promise.fail( function ( session , result , info ) )
promise.always( function ( session , result , info ) )
```

promise

A reference to the Promise object that is returned as the value of the `logout( )` method. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDOSession class on page 124.

session

A reference to the `JSDOSession` object on which `logout( )` was called.

result

A constant indicating the overall result of the call that can have one of the following values:

- **`progress.data.Session.SUCCESS`** — The session logout completed successfully.
- **`progress.data.Session.GENERAL_FAILURE`** — The session logout failed because of some error.

Note: It is not always necessary to test the value of *`result`* in a Promise method callback for the `logout( )` method, especially if the callback is registered using *`promise`*`.done( )` or *`promise`*`.fail( )`, where the callback always executes with the same value (`progress.data.Session.SUCCESS` and `progress.data.Session.GENERAL_FAILURE`, respectively).

---

info

A JavaScript object that can have the following properties:

- **`errorObject`** — An error object thrown while attempting to terminate the login session.
- **`xhr`** — A reference to the XMLHttpRequest object sent to the web server to terminate the login session.

Detailed logout behavior

When this method terminates the associated login session, the `JSDOSession` object can be re-used to start a new session using the same `serviceURI` and `authenticationModel` configuration settings originally passed to the object's constructor. The `JSDOSession` object's properties retain their values from the previous login session, with the following exceptions:

- `clientContextId` is reset to `null`.
- `loginHttpStatus` is reset to `null`.
- `loginResult` is reset to `null`.
- `userName` is reset to `null`.

Existing JSDOs and catalog information are not affected by a successful execution of `logout( )`. However, any attempt to call `addCatalog( )` or a JSDO method that requires contacting the server results in an error object being thrown.

Example

The following code fragment calls the `logout( )` method on the `JSDOsession` object, `session`. This example uses `try` and `catch` blocks in case `logout( )` throws an unexpected exception, and displays messages returned from Promise handlers accordingly:

```
var serviceURI = "http://oemobiledemo.progress.com/OEMobileDemoServicesForm/",
session = new progress.data.JSDOSession({
            "authenticationModel": progress.data.Session.AUTH_TYPE_FORM,
            "serviceURI": serviceURI
    });

try { // Login, add catalogs, create additional JSDO instances,
      // and do JSDO tasks for this session, ...then(...) ...
    session.logout()
    .then(function(session, result, info) {
        // Success handler of logout
        return "Employee session logged out successfully";
    }, function(session, result, info) {
        // Failure handler of logout
        if ( result === progress.data.Session.GENERAL_FAILURE ) {
            if ( info.errorObject ) {
                // Process error object thrown during logout...
            }
            if ( info.xhr ) {
                // Process XHR sent during logout...
            }
            return "Employee session logout failed. Unspecified error.";
        }
        else {
            return "Unexpected failed logout result";
        }
    })
    .then(function (msg) {
        console.log(msg);
    });
}
catch(ex) {
    console.log("There was an unexpected error from attempted logout: " + ex);
}
```

See also:

invalidate( ) method on page 256, invalidateAllSessions( ) stand-alone function on page 258, login( ) method (JSDOSession class) on page 275, offline event on page 292, online event on page 294

logout( ) method (Session class)

Note: Deprecated in Progress Data Objects Version 4.4 and later. Please use the logout( ) method (JSDOSession class) on page 286 instead.

Terminates the login session on the web application managed by the current `Session` object, and reinitializes most of the state information maintained by the object.

This method can be called synchronously or asynchronously, depending on parameters that you pass.

On a successful logout, the `Session` object sets its `connected` property to `false`. If the logout fails, the object leaves its `connected` property set to `true`, unless the failure happened because the app cannot access the server.

Once `logout( )` is executed, no further communication (other than a `login( )` call) can occur between the mobile app and the server until a new login session is established.

Return type: `undefined`

Applies to: progress.data.Session class on page 136

Syntax

```
logout ( )
login ( args )
```

Note: If you call `logout( )` with no parameters, the method executes synchronously. If you call it passing *`args`*, the method executes synchronously or asynchronously, depending on the setting of *`args`*.

`args`

An object that has one property:

- **`async`** — (Optional) A `boolean` that, if `true`, tells `logout( )` to execute asynchronously. If `false` or absent, `logout( )` executes synchronously. For an asynchronous call, the method results are returned in the `afterLogout` event that is fired on the `Session` object after the method completes. Before calling the method, you can subscribe one or more handlers to this event using the `subscribe( )` method on the `Session` object.

Note: If you include any other properties in *`args`*, `logout( )` ignores them.

When this method terminates the associated login session, the `Session` object can be re-used to start a new session. The `Session` object's properties retain their values, with the following exceptions:

- `clientContextId` is reset to `null`.
- `lastSessionXHR` is reset to `null`.
- `loginHttpStatus` is reset to `null`.
- `loginResult` is reset to `null`.
- `userName` is reset to `null`.

Existing JSDOs and catalog information are not affected by `logout( )`. However, any attempt to call `addCatalog( )` or a JSDO method that requires contacting the server results in an `Error` object being thrown.

For all errors encountered by `logout( )`, the method also throws an `Error` object. For detailed information about any unsuccessful `logout( )` response returned from the web server, you can check the XMLHttpRequest object (XHR) returned by the `lastSessionXHR` property. However, note that if the `logout( )` call is successful, `lastSessionXHR` is set to `null`.

Note: When called asynchronously, any handler for the `afterLogout` event is passed the same information that is available for the `logout( )` method when it is called synchronously, including the `Session` object that executed `logout( )` (with the same property settings) and any `Error` object that was thrown processing the server response.

Caution: To help ensure that HTTP Forms access to web applications works in certain web browsers, such as Firefox, when the web application is configured for Cross-Origin Resource Sharing (CORS), always call the `logout( )` method asynchronously.

Example

The following code fragment calls the `logout( )` method without parameters synchronously on the session, `empSession`. For a similar example with the method called asynchronously, see the reference entry for the `afterLogout` event. This synchronous example uses `try` and `catch` blocks in case `logout( )` throws an `Error` object, and displays a message if it does:

```
try {
    empSession.logout( );
}
catch(errObj) {
    var msg;

    msg = errObj ? '\n' + errObj.message : '';
    if ( empSession.lastSessionXHR === null ) {
        alert("logout succeeded");
        return;
    }
    alert("There was an error attempting log out." + msg);
}
```

See also:

afterLogout event on page 190, lastSessionXHR property on page 274, login( ) method (Session class) on page 280, subscribe( ) method (Session class) on page 349

name property (JSDO class)

Returns the name of the Data Object resource for which the current JSDO was created.

This value must match the name of a resource provided by the Data Object Service for which a login session has already been started.

Note: To set this property, you must pass its value to the JSDO constructor.

Data type: `string`

Access: Read-only

Applies to: progress.data.JSDO class on page 110

See also:

services property on page 331, progress.data.Session class on page 136

name property (JSDOSession class)

Note: Applies to Progress Data Objects Version 4.3 or later.

Returns the value of any `name` property to enable page refresh support that was passed to the constructor of the current `JSDOSession` object.

Data type: `string`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 124

See the description of the `JSDOSession` class constructor for more information on the setting and use of this property value.

See also:

getSession( ) stand-alone function on page 248, isAuthorized( ) method on page 268

objParam property

A reference to the object, if any, that was passed as an input parameter to the JSDO method that has returned the current request object.

If no object parameter was passed to the method, the `objParam` property is `undefined`.

Data type: `Object`

Access: Read-only

Applies to: request object on page 146

The `objParam` property is available for all JSDO events where an object parameter is passed to the JSDO method, including `saveChanges(true)`, where such object parameters are passed using a single Submit operation, or in the request object returned to a jQuery Promise callback.

However, this property **does not** apply to the following events:

- `afterSaveChanges`, when `saveChanges( )` or `saveChanges(false)` is called
- `beforeCreate`
- `beforeDelete`
- `beforeUpdate`
- `beforeSaveChanges`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe or register to JSDO events, any returned jQuery Promises, and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO. This object is also returned as the value of any JSDO invocation method that you execute synchronously.

See also:

fill( ) method on page 220, invocation method on page 262, invoke( ) method on page 264, saveChanges( ) method on page 314

offline event

Fires when the current `JSDOSession` or `Session` object detects that the device on which it is running has gone offline, or that the web application to which it has been connected is no longer available.

This event always fires when the device on which the `JSDOSession` or `Session` object is created goes offline (that is, the device is disconnected from the network). For this event to fire when the web application to which it has been connected is no longer available, the `JSDOSession` or `Session` object must have previously:

1. Been connected to the web application using the object's `login( )` method
2. Not been disconnected from the web application using the object's `logout( )` method

Applies to: progress.data.JSDOSession class on page 124, progress.data.Session class on page 136

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( session , off-line-reason , request )
```

session

A reference to the `JSDOSession` or `Session` object that has detected the offline condition.

off-line-reason

A `string` constant indicating the reason that the offline event has been fired. Possible constants include:

- **`progress.data.Session.APPSERVER_OFFLINE`** — (OpenEdge Data Object Services only) The other components necessary to run the service are available, but the associated OpenEdge application server is offline.
- **`progress.data.Session.DEVICE_OFFLINE`** — The device itself is offline. For example, it might be in airplane mode, or it might be unable to pick up a Wi-Fi or cell signal.
- **`progress.data.Session.SERVER_OFFLINE`** — The web server is not available. For a Rollbase Data Object Service, this is the web server for the public or private cloud. For an OpenEdge Data Object Service, this is the Tomcat Java servlet container.
- **`progress.data.Session.WEB_APPLICATION_OFFLINE`** — The server is running, but the Java web application that implements the Data Object Service is not deployed.

Note: The `progress.data.Session.DEVICE_OFFLINE` condition will fire the `offline` event on a `JSDOSession` or `Session` object even if the object has not yet invoked its `login( )` method.

You can use the string values of these constants directly to display or log messages, or test the values only and respond to the result in some other way.

request

If the offline condition was detected as a result of a request sent on behalf of a JSDO, this is a reference to the `request` object used to make the request. For more information, see the description of the request object on page 146. If the offline event is the result of the device going offline or because of a call to the `ping( )` method (either directly or because the `pingInterval` property is set greater than 0), this parameter is `null`.

Note: If the underlying database accessed by the Rollbase server or OpenEdge application server is unavailable, this will **not** cause the `offline` event to fire, nor will the fact that a particular Data Object Service contained by the hosting web application was not loaded when the application started.

Example

The following code fragment subscribes the function, `onSessionOffline`, to handle the `offline` event fired on the session, `mySession`:

```
/* subscribe to the offline event */
var mySession = new progress.data.JSDOSession(
        { serviceURI: "http://localhost/CustService",
          authenticationModel: progress.data.Session.AUTH_TYPE_FORM
        };
mySession.subscribe('offline', onSessionOffline );

/* some code that might cause mySession to detect that the
   application is offline */

. . .

function onSessionOffline( session , offlineReason , request ) {

  document.write(offLineReason);
  switch( offlineReason ) {
  case progress.data.Session.APPSERVER_OFFLINE:
    document.write("Check with the server system administrator");
    . . .
    break;
  case progress.data.Session.DEVICE_OFFLINE:
    document.write("Ensure that your device is online");
    . . .
    break;
  case progress.data.Session.SERVER_OFFLINE:
    document.write("Check with the server system administrator");
    . . .
    break;
  case progress.data.Session.WEB_APPLICATION_OFFLINE:
    document.write("Check with the server system administrator");
    . . .
    break;
  default:
    document.write("Check with the server system administrator");
  };

};
```

See also:

connected property on page 216, login( ) method (JSDOSession class) on page 275, login( ) method (Session class) on page 280, logout( ) method (JSDOSession class) on page 286, logout( ) method (Session class) on page 288, online event on page 294, ping( ) method (JSDOSession class) on page 297, ping( ) method (Session class) on page 300, subscribe( ) method (JSDOSession class) on page 348, subscribe( ) method (Session class) on page 349, unsubscribe( ) method (JSDOSession class) on page 355, unsubscribe( ) method (Session class) on page 356

online event

Fires when the current `JSDOSession` or `Session` object detects that the device on which it is running has gone online after it was previously offline, or that the web application to which it is connected is now available after it was previously unavailable.

This event always fires when the device on which the `JSDOSession` or `Session` object is created goes online after having been offline (that is, the device is reconnected to the network). For this event to fire when the web application to which it has been connected is now available after having been unavailable, the `JSDOSession` or `Session` object must have previously:

1. Been connected to the web application using the object's `login( )` method
2. Not been disconnected from the web application using the object's `logout( )` method
3. Detected an `offline` event as a result of sending a Data Object Service request to the web application or executing the `ping( )` method

Applies to: progress.data.JSDOSession class on page 124, progress.data.Session class on page 136

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( session , request )
```

session

A reference to the `JSDOSession` or `Session` object that has detected the online condition.

request

If the online condition was detected as a result of a request sent on behalf of a JSDO, this is a reference to the `request` object used to make the request. For more information, see the description of the request object on page 146. If the online event is the result of the device itself coming back online or because of a call to the `ping( )` method (either directly or because the `pingInterval` property is set greater than 0), this parameter is `null`.

This event can fire because:

- The device on which the app is running was previously offline (in airplane mode or could not connect to a Wi-Fi network), but is now online again.

Note: This condition will fire the `online` event on a `JSDOSession` or `Session` object even if the object has not yet invoked its `login( )` method or has already invoked its `logout( )` method .

- A `JSDOSession` or `Session` object that previously fired its `offline` event because of a failed attempt to contact a Data Object Service has now been used successfully to contact that same Data Object Service.

Example

The following code fragment subscribes the function, `onSessionOnline`, to handle the `online` event fired on the session, `mySession`:

```
/* subscribe to the online event */
var mySession = new progress.data.JSDOSession(
        { serviceURI: "http://localhost/CustService",
          authenticationModel: progress.data.Session.AUTH_TYPE_FORM
        };
mySession.subscribe('online', onSessionOnline );

/* some code that might cause mySession to detect that the
   application is online */

. . .

function onSessionOnline( session , request ) {

  document.write("Your best mobile app is back to work! "
                 + "You might be prompted to login again.");

};
```

See also:

connected property on page 216, login( ) method (JSDOSession class) on page 275, login( ) method (Session class) on page 280, logout( ) method (JSDOSession class) on page 286, logout( ) method (Session class) on page 288, offline event on page 292, ping( ) method (JSDOSession class) on page 297, ping( ) method (Session class) on page 300, subscribe( ) method (JSDOSession class) on page 348, subscribe( ) method (Session class) on page 349, unsubscribe( ) method (JSDOSession class) on page 355, unsubscribe( ) method (Session class) on page 356

onOpenRequest property

Returns the reference to a user-defined callback function that the `JSDOSession` or `Session` object executes to modify a request object before sending the request object to the server.

For example, this function might add a message header by modifying the XMLHttpRequest object used to send the request.

You do not typically use this property, because the `JSDOSession` or `Session` object properly handles preparation of the request object for normal circumstances.

Data type: `function`

Access: Readable/Writable

Applies to: progress.data.JSDOSession class on page 124, progress.data.Session class on page 136

By default, the value of the `onOpenRequest` property is `null`, meaning that the request object is sent without modification. If the value is set to a callback function, the function takes a single object parameter.

Syntax

This is the syntax for setting this property to a function reference:

```
mySession.onOpenRequest = funcRef
```

mySession

A reference to the `JSDOSession` or `Session` object for which the request object is to be modified before sending a request to the server.

funcRef

A reference to a JavaScript callback function that has the following signature:

Syntax:

```
function [ func-name ] ( param )
```

func-name

The optional name of a function you have defined external to the property assignment. Alternatively, you can specify *funcRef* as the entire inline function definition without *func-name*.

param

An `Object` that has the following properties:

- **xhr** — An object reference to the XMLHttpRequest object (XHR) used to send the request. The current request object can be modified by the function. When the callback is called, `XMLHttpRequest.open( )` will already have been called on the XHR , but the callback can call `open( )` again, overriding the effects of the first `open( )` call. When the callback function is used for a `login( )`, `addCatalog( )`, or `logout( )` call, although it should not be necessary and is not recommended, it is possible to replace the XHR entirely by creating a new object and assigning it as the value of the `xhr` property.
- **verb** — The HTTP operation (GET, PUT, etc.) to be performed by the request.
- **uri** — The URI to which the request is addressed.
- **session** — A reference to the `Session` object that invoked the callback.
- **formPreTest** — A `boolean` specifying whether the current `login( )` request is a preliminary request, used in cases of Form authentication, to determine whether the user is already logged in (`true`) or an actual login request (`false`).
- **async** — A `boolean` specifying whether the request is asynchronous (`true`) or synchronous (`false`).

Caution: For a `JSDOSession` object, if the callback function is used for a `login( )`, `addCatalog( )`, or `logout( )` call, and if it calls `XMLHttpRequest.open( )`, the `async` property value passed to that `open( )` call must be `true` (i.e., the `open( )` method must specify that the request is to be sent asynchronously).

Caution: For a `Session` object, if the callback function is used for a `login( )`, `addCatalog( )`, or `logout( )` call, and if it calls `XMLHttpRequest.open( )`, the `async` property value passed to that `open( )` call must match the `async` value that was passed to the `login( )`, `addCatalog( )`, or `logout( )` call.

If you assign a callback function as the value of `onOpenRequest`, it remains in effect for all requests for the duration of the session unless it is replaced by another function or is set to `null`.

Example

Be sure to reset the value of the property as necessary, as in the following example for the `JSDOSession` object, `myJSDOsession`:

```
myJSDOsession.onOpenRequest = function( params ) {
    params.xhr.setRequestHeader('Authorization', auth);
};

myJSDOsession.login(username, password).done(
    function(JSDOsession, result, info) {
        JSDOsession.onOpenRequest = null;
        . . .
}).fail(
    function(JSDOsession, result, info) {
        alert("Login failed");
});
```

See also:

request object on page 146, xhr property on page 358

ping( ) method (JSDOSession class)

Determines the online state of the current `JSDOSession` object from its ability to access the web application that it manages, and for an OpenEdge web application, from detecting if its associated application server is running.

This method also causes an `offline` or `online` event to fire if the ping detects that there has been a change in the `JSDOSession` object's online state.

This method throws an exception if the `JSDOSession` object has not logged into its web application or has since logged out.

This method is always executed asynchronously and returns results in callbacks that you register using methods of a Promise object returned as the method value.

Return type: jQuery Promise

Applies to: progress.data.JSDOSession class on page 124

Syntax

```
ping ( )
```

Promise method signatures

jQuery Promise objects define methods that you can call to register a callback function with a specific signature. The callback signatures depend on the method that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that `ping( )` returns:

Syntax:

```
promise.done( function ( session , result , info ) )
promise.fail( function ( session , result , info ) )
promise.always( function ( session , result , info ) )
```

promise

A reference to the Promise object that is returned as the value of the `ping( )` method. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDOSession class on page 124.

session

A reference to the `JSDOSession` object on which `ping( )` was called.

result

A `boolean` that indicates the online state of the `JSDOSession` object on which `ping( )` is called. If set to `true`, `ping( )` has determined that the current `JSDOSession` object is connected and logged into a web application (and its associated application server, for an OpenEdge Data Object Service), or if set to `false`, the object is disconnected from any web application (**or** its associated application server, for an OpenEdge Data Object Service).

info

A JavaScript object that can have the following properties:

- **`xhr`** — A reference to the XMLHttpRequest object sent to the web server to make the ping request to the web application.
- **`offlineReason`** — A `string` constant that `ping( )` sets only if it determines that the `JSDOsession` object is disconnected from any web application or its associated application server. The constant value indicates the reason for its offline state. Possible values are the same as those that can be passed to the `offline` event callback function as the value of its *`off-line-reason`* parameter, and include:
  - **`progress.data.Session.APPSERVER_OFFLINE`** — (OpenEdge Data Object Services only) The other components necessary to run the service are available, but the associated OpenEdge application server is offline.
  - **`progress.data.Session.DEVICE_OFFLINE`** — The device itself is offline. For example, it might be in airplane mode, or it might be unable to pick up a Wi-Fi or cell signal.

- **progress.data.Session.SERVER_OFFLINE** — The web server is not available. For a Rollbase Data Object Service, this is the web server for the public or private cloud. For an OpenEdge Data Object Service, this is the Tomcat Java servlet container.
- **progress.data.Session.WEB_APPLICATION_OFFLINE** — The server is running, but the Java web application that implements the Data Object Service is not deployed.

Detailed ping behavior

A `JSDOSession` object is considered to be in an online state if all of the following are true:

- The object can communicate with its web application. That is, the web server is running, the web application is started, and it has accessible Data Object Services.
- For OpenEdge Data Object Services only, the associated OpenEdge application server that the web application accesses is running and accessible.

When you execute `ping( )` on a `JSDOSession` instance, the method indicates the instance's current online status through its returned Promise object using the most commonly registered callbacks as follows:

- If the instance is online, the returned Promise object invokes any callback you register using the object's `done( )` method. The `done( )` callback's `result` parameter is always set to `true`.
- If the instance is offline, the returned Promise object invokes any callback you register using the object's `fail( )` method. The `fail( )` callback's `result` parameter is always set to `false`.
- The returned Promise object invokes any callback you register using the object's `always( )` method with the value of the callback's `result` parameter indicating the instance's online status as described for the Promise method signatures above.

Note: You can have `ping( )` executed automatically by setting the `pingInterval` property on the `JSDOSession` object, which specifies the time interval between invocations of `ping( )`. However, automatic invocations do not return Promise objects, but only fire `offline` and `online` events appropriately when a change in the `JSDOSession` instance's online status is detected.

Note: For OpenEdge Data Object Services, an OpenEdge application server supports `ping( )` using an OpenEdge-defined ABL class, `OpenEdge.Rest.Admin.AppServerStatus`. This class responds to a REST ping service call to its `ServerStatus( )` method, which indicates that the application server is available when the method returns successfully. You can also define a version of this method in your own user-defined ABL class that returns a custom string value when `ping( )` returns successfully, and you can retrieve this value from the `xhr` object reference returned by `ping( )`. For more information, see the sections on constructing and debugging REST requests in the administration documentation for the OpenEdge AppServer or the Progress Application Server for OpenEdge.

Example

The following code fragment shows how you can use `ping( )` to check the online state of a given `JSDOSession` object after logging in:

```
var mySession = new progress.data.JSDOSession ( { . . . } );

/* These session events can fire on device online
   status changes even prior to logging in */
mySession.subscribe('offline', onSessionOffline );
mySession.subscribe('online', onSessionOnline );

. . .

mySession.login().done( // Anonymous login
    function( session, result, info ) {

        . . .

        ping().done(
            function( session, result, info ) {
                console.log("Session ping result: Online");
                }
            }).fail(
            function( session, result, info ) {
                console.log("Session ping result: Offline -- "
                            + info.offlineReason);
                // Process info.xhr, if necessary, for more information
            });

        . . .

    }).fail( /* Process failed login . . . */ );

. . .

function onSessionOffline( pSession , pOfflineReason , pRequest ) {
  console.log("The session status has changed to: Offline -- "
              + pOfflineReason);
};

function onSessionOnline( pSession , pRequest ) {
  console.log("The session status has changed to: Online");
};
```

See also:

connected property on page 216, offline event on page 292, online event on page 294, pingInterval property on page 303

ping( ) method (Session class)

Note: Deprecated in Progress Data Objects Version 4.4 and later. Please use the ping( ) method (JSDOSession class) on page 297 instead.

Determines the online state of the current `Session` object from its ability to access the web application that it manages, and for an OpenEdge web application, from detecting if its associated application server is running.

This method also causes an `offline` or `online` event to fire if the ping detects that there has been a change in the `Session` object's online state.

This method throws an exception if the `Session` object has not logged into its web application or has since logged out.

You can call this method either synchronously or asynchronously.

Return type: `boolean`

Applies to: progress.data.Session class on page 136

Syntax

```
ping ( [ parameter-object ] ) )
```

parameter-object

An object with any or all of the following properties:

- **async** — A `boolean`. If set to `true`, `ping( )` executes asynchronously. If set to `false`, `ping( )` executes synchronously. If you do not pass this property setting, `ping( )` executes asynchronously.
- **onCompleteFn** — A callback function that is called only if `ping( )` executes asynchronously. This function is called after the `ping( )` method receives a response from the server (or times out), regardless of the online status returned:

  Syntax:

  ```
  function [ func-name ] ( completion-object )
  ```

  Where *func-name* is the name of a callback function that you define external to *parameter-object*, and *completion-object* is an object parameter that `ping( )` passes to the function with these properties:

  - **pingResult** — A `boolean` that indicates the session online status that `ping( )` returns. If set to `true`, `ping( )` determined that the current `Session` object is connected and logged in to a web application (and its application server, for an OpenEdge Data Object Service), or if set to `false`, the session is disconnected from any web application (**or** its application server, for an OpenEdge Data Object Service).
  - **xhr** — The XMLHttpRequest object that `ping( )` used to make the request.
  - **offlineReason** — A string constant that `ping( )` sets only if it determines that the session is disconnected from any web application. The constant value indicates the reason for its offline status. Possible values are the same as those that can be passed to the `offline` event callback function as the value of its *off-line-reason* parameter, as shown below.

- **xhr** — If you call `ping( )` synchronously, this is a property that you can initially set to any value (including `null` or `undefined`), but which `ping( )` resets to a reference to the XMLHttpRequest object that `ping( )` used to make the request. If you do not include an initial setting for this property in *parameter-object*, you cannot access this XMLHttpRequest object for synchronous `ping( )` call.

- **offlineReason** — If you call `ping( )` synchronously, this is a property that you can initially set to any value (including `null` or `undefined`), but which `ping( )` resets to a string constant value if it determines that the session is disconnected from any web application. This constant value indicates the reason for its offline status. Possible values are the same as those that can be passed to the `offline` event callback function as the value of its *`off-line-reason`* parameter, as shown below.

  Caution: If you do not include an initial setting for this property in *`parameter-object`*, any `offlineReason` value returned by `ping( )` is unavailable for a synchronous call.

The possible values for the `offlineReason` properties described above include:

- **progress.data.Session.APPSERVER_OFFLINE** — (OpenEdge Data Object Services only) The other components necessary to run the service are available, but the associated OpenEdge application server is offline.
- **progress.data.Session.DEVICE_OFFLINE** — The device itself is offline. For example, it might be in airplane mode, or it might be unable to pick up a Wi-Fi or cell signal.
- **progress.data.Session.SERVER_OFFLINE** — The web server is not available. For a Rollbase Data Object Service, this is the web server for the public or private cloud. For an OpenEdge Data Object Service, this is the Tomcat Java servlet container.
- **progress.data.Session.WEB_APPLICATION_OFFLINE** — The server is running, but the Java web application that implements the Data Object Service is not deployed.

Detailed ping behavior

The `ping( )` method always returns the overall session online status as a `boolean` value: `true` if the session's web application is fully available (online and with accessible Data Object Services) and `false` if the web application is not available (offline or with inaccessible Data Object Services). If you call `ping( )` synchronously, it returns this online status as its method return value. If you call `ping( )` asynchronously, the method always returns with a value of `false`, but passes its online status as the value of the `pingResult` property it passes to its `onCompleteFn` callback.

A `Session` object is considered to be in an online state if all of the following are true:

- The object can communicate with its web application. That is, the web server is running, the web application is started, and it has accessible Data Object Services.
- For OpenEdge Data Object Services only, the associated OpenEdge application server that the web application accesses is running and accessible.

Note: You can have `ping( )` executed automatically by setting the `pingInterval` property on the `Session` object, which specifies the time interval between invocations of `ping( )`.

Note: For OpenEdge Data Object Services, an OpenEdge application server supports `ping( )` using an OpenEdge-defined ABL class, `OpenEdge.Rest.Admin.AppServerStatus`. This class responds to a REST ping service call to its `ServerStatus( )` method, which indicates that the application server is available when the method returns successfully. You can also define a version of this method in your own user-defined ABL class that returns a custom string value when `ping( )` returns successfully, and you can retrieve this value from the `xhr` object reference returned by `ping( )`. For more information, see the sections on constructing and debugging REST requests in the administration documentation for the OpenEdge AppServer or the Progress Application Server for OpenEdge.

Example

The following code fragment shows how you can use `ping( )` to check the online status of a given `Session` object after logging in:

```
var mySession = new progress.data.Session;

/* These session events can fire on device status changes even before logging in */
mySession.subscribe('offline', onSessionOffline );
mySession.subscribe('online', onSessionOnline );

. . .

/* Code to set webapplURI, username, and password variables and log in . . . */
mySession.authenticationModel = progress.data.Session.AUTH_TYPE_FORM;
var username = . . . ;
var password = . . . ;
var webApplicationURI = . . . ;
mySession.login( webApplicationURI, username, password );

. . .

var pingParm = { async : false, offlineReason : null };

if ( !ping(pingParm) ) {
  document.write("Session ping result: " + pingParm.offlineReason);
}
else {
  document.write("Session ping result: Online");
}

. . .

function onSessionOffline( pSession , pOfflineReason , pRequest ) {
  document.write("The session status has changed to: " + pOfflineReason);
};

function onSessionOnline( pSession , pRequest ) {
  document.write("The session status has changed to: Online");
};
```

See also:

connected property on page 216, offline event on page 292, online event on page 294, pingInterval property on page 303

pingInterval property

A `number` that specifies the duration, in milliseconds, between one automatic execution of the current `JSDOSession` or `Session` object's `ping( )` method and the next.

Setting this property to a value greater than zero (0) causes the `JSDOSession` or `Session` object to begin executing its `ping( )` method, and when execution completes, to repeatedly execute the method after the specified delay. If you set its value to zero (0), no further execution of `ping( )` occurs after any current execution completes. The default value is zero (0).

Data type: `number`

Access: Readable/Writable

Applies to: progress.data.JSDOSession class on page 124, progress.data.Session class on page 136

You can set `pingInterval` to start the automatic execution of `ping( )` any time after you create the `JSDOSession` or `Session` object. However, `ping( )` does not begin executing until and unless you have successfully invoked the object's `login( )` method to start a JSDO login session.

Note that when you call the `ping( )` method directly, you have several options for how to call it to get the results. You do not have these options and you cannot get results directly from each automatic execution of `ping( )` that begins from a setting of `pingInterval`. The effects from this automatic execution are limited to causing the `JSDOSession` or `Session` object to fire its `offline` or `online` event, and to change the value of its `connected` property, when a given `ping( )` execution detects a change in the object's online status.

See also:

connected property on page 216, JSDOs property on page 272, login( ) method (JSDOSession class) on page 275, login( ) method (Session class) on page 280, offline event on page 292, online event on page 294, ping( ) method (JSDOSession class) on page 297, ping( ) method (Session class) on page 300

readLocal( ) method

Clears out the data in JSDO memory and replaces it with all the data stored in a specified local storage area, including any pending changes and before-image data, if they exist.

Return type: `boolean`

Applies to: progress.data.JSDO class on page 110

Working record: After execution, no working records are set for the tables of the JSDO.

Syntax

```
readLocal ( [ storage-name ] )
```

storage-name

The name of the local storage area whose data is to replace the data in JSDO memory. If *storage-name* is not specified, blank, or `null`, the name of the default storage area is used. The name of this default area is `jsdo_`*serviceName*`_`*resourceName*, where *serviceName* is the name of the Data Object Service that supports the JSDO instance, and *resourceName* is the name of the resource (table, dataset, etc.) for which the JSDO instance is created.

This method returns `true` if it successfully reads the data from the local storage area; it then replaces JSDO memory with this data. If the storage area has no data (is empty), this clears JSDO memory instead of replacing it with any data, and the method also returns `true`. If *storage-name* does not exist, but otherwise encounters no errors, the method ignores (does not clear) JSDO memory and returns `false`. If the method does encounter errors (for example, with the data in the specified storage area), it also leaves JSDO memory unchanged and throws an exception.

You can call the JSDO `saveChanges( )`, `acceptChanges( )`, or `rejectChanges( )` method after calling this method, and any changes read into JSDO memory from local storage are handled appropriately.

Example

The following code fragment replaces the data in JSDO memory with all the data currently stored in the default storage area:

```
dataSet = new progress.data.JSDO( 'dsStaticData' );
dataSet.fill();
dataSet.saveLocal();
.
.
.
dataSet.readLocal();
```

See also:

acceptChanges( ) method on page 153, fill( ) method on page 220, rejectChanges( ) method on page 305, saveChanges( ) method on page 314, saveLocal( ) method on page 330

record property

A property on a JSDO table reference that references a `JSRecord` object with the data for the working record of a table referenced in JSDO memory.

If no working record is set for the referenced table, this property is `undefined`.

Data type: progress.data.JSRecord class on page 133

Access: Read-only

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

The table reference that provides this property can either be the value of a property on the JSDO with the name of a referenced table in JSDO memory or a reference to the JSDO itself if the JSDO references only a single table.

The field values and any additional properties for the working record are provided by the object returned by the `data` property in the `JSRecord` object that is returned by the `record` property.

See also:

data property on page 217

rejectChanges( ) method

Rejects changes to the data in JSDO memory for the specified table reference or for all table references of the specified JSDO.

If the method succeeds, it returns `true`. Otherwise, it returns `false`.

Note: This method is most useful when the JSDO `autoApplyChanges` property is set to `false`. In this case, you typically invoke this method **after** calling the `saveChanges( )` method in order to cancel a series of changes that have failed on the server. If the `autoApplyChanges` property is `true`, the JSDO automatically accepts or rejects changes for the specified table reference, or for all table references of the specified JSDO, based on the success of the corresponding record-change operations on the server resource.

Note: Rejecting all pending changes in JSDO memory—or even pending changes for a single table reference—because only some were unsuccessful on the server might be too broad an action for your application. If so, consider using `rejectRowChanges( )` to reject changes a single table record at a time. For more information, see the description of `rejectRowChanges( )` method.

Return type: `boolean`

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Working record: The working record is set depending on the changes rejected.

Syntax

```
jsdo-ref.rejectChanges ( )
jsdo-ref.table-ref.rejectChanges ( )
```

jsdo-ref

A reference to the JSDO. If you call the method on *jsdo-ref*, the method rejects changes for all table references in the JSDO.

table-ref

A table reference on the JSDO. If you call the method on *table-ref*, the method rejects changes for the specified table reference.

When you reject changes on a table reference, this method backs out all pending changes to the record objects in the specified table in JSDO memory, and uses any before-image data to return each record to its original data values before the pending changes were made. When you reject changes on the JSDO reference, the method backs out all pending changes to the record objects in all tables of the JSDO, and uses any before-image data to return each record to its original data values before the pending changes were made. As the specified changes are rejected, the method also empties any associated before-image data, clears all associated error message settings, and removes the associated pending record change indications from JSDO memory.

Note: Regardless if you call `rejectChanges( )`, any error message settings that result from Data Object resource operations invoked by the most recent execution of the `fill( )` or `saveChanges( )` method remain available for return by the `getErrors( )` method until the next execution of either `fill( )` or `saveChanges( )`. For more information, see the getErrors( ) method on page 233 description.

This method is especially useful for a resource that supports before-imaging (such as an OpenEdge ProDataSet) and handles the results of all record changes sent using a Submit operation as part of a single server transaction that undoes all the record changes in response to any single record-change error. When the Submit operation returns with an error, calling this method ensures that JSDO memory is synchronized with all the record changes that were undone as part of the server transaction.

Note: After this method rejects changes, and if you have set up automatic sorting using the `autoSort` property, all the record objects for affected table references are sorted accordingly. If the sorting is done using sort fields, any `string` fields are compared according to the value of the `caseSensitive` property.

Caution: If you have already successfully applied these changes on the server using the `saveChanges( )` method, **do not** invoke this method if you want the affected client data to be consistent with the corresponding data on the server.

Example

The following code fragment shows a JSDO created so it **does not** automatically accept or reject changes to data in JSDO memory after a call to the `saveChanges( )` method. Instead, it subscribes a handler for the JSDO `afterSaveChanges` event to determine if all changes to the `eCustomer` table in JSDO memory should be accepted or rejected based on the success of all resource Create, Update, and Delete operations on the server. To change the data for a record, a jQuery event is also defined on an update button to update the corresponding `eCustomer` record in JSDO memory with the current field values entered in a customer detail form (`#custdetail`):

```
dataSet = new progress.data.JSDO( { name: 'dsCustomerOrder',
                                                    autoApplyChanges : false } );
dataSet.eCustomer.subscribe('afterSaveChanges', onAfterSaveCustomers, this);

$('#btnUpdate').bind('click', function(event) {
  var jsrecord = dataSet.eCustomer.findById($('#custdetail #id').val());
  if (jsrecord) {jsrecord.assign();};
});

// Similar controls might be defined to delete and create eCustomer records...

$('#btnSave').bind('click', function(event) {
  dataSet.saveChanges();
});

function onAfterSaveCustomers(jsdo, success, request) {
  if (success)  {
    jsdo.eCustomer.acceptChanges();
    // Additional actions associated with accepting the pending changes...
  }
  else  {
    jsdo.eCustomer.rejectChanges();
    // Additional actions associated with rejecting the pending changes...
  }
}
```

When the update button is clicked, the event handler uses the `findById( )` method to find the original record (`jsrecord`) with the matching internal record ID (`#id`) and invokes the `assign( )` method on `jsrecord` with an empty parameter list to update its fields in `eCustomer` with any new values entered into the form. You might define similar events and controls to delete `eCustomer` records and add new `eCustomer` records.

An additional jQuery event also defines a save button that when clicked invokes the `saveChanges( )` method to apply all pending changes in JSDO memory to the server. After the method completes, and all results have been returned to the client from the server, the JSDO `afterSaveChanges` event fires, and if any resource operations on the server were **not** successful, the handler calls `rejectChanges( )` to reject all pending `eCustomer` changes in JSDO memory.

Note: This example shows the default invocation of `saveChanges( )`, which invokes each resource record-change operation, one record at a time, across the network. For a resource that supports before-imaging (such as an OpenEdge ProDataSet), you can also have `saveChanges( )` send all pending record change operations across the network in a single Submit operation. For more information and an example, see the description of the `saveChanges( )` method.

See also:

acceptChanges( ) method on page 153, autoApplyChanges property on page 202, autoSort property on page 203, caseSensitive property on page 213, rejectRowChanges( ) method on page 308, saveChanges( ) method on page 314

rejectRowChanges( ) method

Rejects changes to the data in JSDO memory for a specified record object.

This can be the working record of a table reference or the record specified by a `JSRecord` object reference. If the method succeeds, it returns `true`. Otherwise, it returns `false`.

Note: This method is most useful when the JSDO `autoApplyChanges` property is set to `false`. In this case, you might invoke this method in the callback for the corresponding JSDO event fired in response to an unsuccessful record-change operation on the server resource that was invoked by executing the `saveChanges( )` method. If the `autoApplyChanges` property is `true`, the JSDO automatically accepts or rejects changes to the record object based on the success of the corresponding resource operation on the server.

Return type: `boolean`

Applies to: progress.data.JSRecord class on page 133, progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Working record: The working record is set depending on the changes rejected.

Syntax

```
jsrecord-ref.rejectRowChanges ( )
jsdo-ref.rejectRowChanges ( )
jsdo-ref.table-ref.rejectRowChanges ( )
```

jsrecord-ref

A reference to a `JSRecord` object for a table reference in JSDO memory.

You can obtain a `JSRecord` object by:

- Invoking a JSDO method that returns record objects from a JSDO table reference (`find( )`, `findById( )`, or `foreach( )`)
- Accessing the `record` property on a JSDO table reference that already has a working record.
- Accessing the *`record`* parameter passed to the callback of a JSDO `afterCreate`, `afterDelete`, or `afterDelete` event.

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference, and that table reference has a working record.

table-ref

A table reference on the JSDO that has a working record.

When you reject changes on a specified record object, this method makes the record reflect all pending changes to it in JSDO memory. As the specified changes are rejected, the method also empties any associated before-image data, clears any associated error message setting, and removes the associated pending change indications from JSDO memory.

Note: Regardless if you call `rejectRowChanges( )`, any error message settings that result from Data Object resource operations invoked by the most recent execution of the `fill( )` or `saveChanges( )` method remain available for return by the `getErrors( )` method until the next execution of either `fill( )` or `saveChanges( )`. For more information, see the getErrors( ) method on page 233 description.

Note: After this method rejects changes on a record, and if you have set up automatic sorting using the `autoSort` property, all the record objects for the affected table reference are sorted accordingly. If the sorting is done using sort fields, any `string` fields are compared according to the value of the `caseSensitive` property.

Caution: If you have successfully applied these JSDO changes to the server using the `saveChanges( )` method, **do not** invoke this method if you want the affected client data to be consistent with the corresponding data on the server.

Example

The following code fragment shows a JSDO created so it **does not** automatically accept or reject changes to data in JSDO memory after a call to the `saveChanges( )` method. Instead, it subscribes a single handler for each of the `afterDelete`, `afterCreate`, and `afterUpdate`, events to determine if changes to any `eCustomer` table record in JSDO memory should be accepted or rejected based on the success of the corresponding resource operation on the server. To change the data for a record, a jQuery event is also defined on a save button to update the corresponding `eCustomer` record in JSDO memory with the current field values entered in a customer detail form (`#custdetail`):

```
dataSet = new progress.data.JSDO( { name: 'dsCustomerOrder',
                                                  autoApplyChanges : false } );
dataSet.eCustomer.subscribe('afterDelete', onAfterCustomerChange, this);
dataSet.eCustomer.subscribe('afterCreate', onAfterCustomerChange, this);
dataSet.eCustomer.subscribe('afterUpdate', onAfterCustomerChange, this);

$('#btnSave').bind('click', function(event) {
  var jsrecord = dataSet.eCustomer.findById($('#custdetail #id').val());
  if (jsrecord) {jsrecord.assign();};
  dataSet.saveChanges();
});

// Similar controls might be defined to delete and create eCustomer records...

function onAfterCustomerChange(jsdo, record, success, request) {
  if (success)  {
    record.acceptRowChanges();
  }
  else
  {
    record.rejectRowChanges();
    // Perform other actions associated with rejecting this record change
  }
}
```

When the button is clicked, the event handler uses the `findById( )` method to find the original record with the matching internal record ID (`#id`) and invokes the `assign( )` method on `jsrecord` with an empty parameter list to update its fields in `eCustomer` with any new values entered into the form. It then calls the `saveChanges( )` method to invoke the resource Update operation to apply these record changes to the server. You might define similar events and controls to delete the `eCustomer` record or add a new `eCustomer` record.

After each resource operation for a changed `eCustomer` record completes, results of the operation are returned to the client from the server, and the appropriate event fires. If the operation was **not** successful, the handler calls `rejectRowChanges( )` to reject the record change associated with the event in JSDO memory. An advantage of using an event to manually reject a record change is that you can perform other actions associated with rejecting this particular change, such as displaying an alert to the user that identifies the error that caused the rejection.

Note: This example shows the default invocation of `saveChanges( )`, which invokes each resource record-change operation, one record at a time, across the network. For a resource that supports before-imaging (such as an OpenEdge ProDataSet), you can also have `saveChanges( )` send all pending record change operations across the network in a single Submit operation. For more information and an example, see the description of the `saveChanges( )` method.

See also:

acceptRowChanges( ) method on page 156, autoApplyChanges property on page 202, autoSort property on page 203, caseSensitive property on page 213, saveChanges( ) method on page 314, rejectChanges( ) method on page 305

remove( ) method

Deletes the specified table record referenced in JSDO memory.

The specified record can either be the working record of a referenced table or any record provided by a `JSRecord` object.

To synchronize the change on the server, call the `saveChanges( )` method.

Return type: `boolean`

Applies to: progress.data.JSRecord class on page 133, progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Working record: After execution, any prior working record setting for the affected table, and for any of its child tables, is no longer set.

Syntax

```
jsrecord-ref.remove ( )
jsdo-ref.remove ( )
jsdo-ref.table-ref.remove ( )
```

jsrecord-ref

A reference to a `JSRecord` object for a table record in JSDO memory.

You can obtain a `JSRecord` object by:

- Invoking a JSDO method that returns record objects from a JSDO table reference (`find( )`, `findById( )`, or `foreach( )`)
- Accessing the `record` property on a JSDO table reference that already has a working record.
- Accessing the *`record`* parameter passed to the callback of a JSDO `afterCreate`, `afterDelete`, or `afterDelete` event.
- Accessing each record object provided by the `jsrecords` property on the *`request`* object parameter passed to the callback of a JSDO `afterSaveChanges` event, or passed to the callback of any Promise object returned from the `saveChanges( )` method. The `jsrecords` property is only available on completion of a Submit operation (`saveChanges(true)`) on a resource that supports before-imaging, and **not** if the resource supports before-imaging without Submit.

jsdo-ref

A reference to the JSDO. You can call the method on *`jsdo-ref`* if the JSDO has only a single table reference, and that table reference has a working record.

table-ref

A table reference on the JSDO that has a working record.

Note: This method does not trigger an automatic sort and has no effect on any existing sort order established for the table reference. However, if there is a sort order that depends on the presence or absence of the record object you are removing, and you want to establish a new sort order with this record object absent, you must manually sort the remaining record objects using the `sort( )` method by passing it the same sort function that you used to establish the previous sort order.

Example

The following code fragment shows a jQuery event defined on a delete button to delete the record displayed in a customer detail form from the `eCustomer` table referenced in JSDO memory:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );

$('#btnDelete').bind('click', function(event) {
  var jsrecord = dataSet.eCustomer.findById($('#custdetail #id').val());
  if (jsrecord) {jsrecord.remove();};
  dataSet.saveChanges();
});
```

The form has been previously displayed with values from the same record. When the button is clicked, the event handler uses the `findById( )` method to find the original record with the matching internal record ID (`jsrecord`) and invokes the `remove( )` method on `jsrecord` to delete the record from `eCustomer`.

See also:

data property on page 217, saveChanges( ) method on page 314, sort( ) method on page 344

response property

Returns an object or `string` containing data and status information from an operation invoked on a Data Object resource.

Note: As detailed here, you can inspect the object or `string` value of this property to return certain types of error information from a resource operation, depending on the operation, the error origination, and whether or not the resource supports before-imaging. As of Progress Data Objects 4.3 and later, the JSDO provides a `getErrors( )` method that returns **all** types of error information for a specified table reference associated with a resource operation in a consistent manner, regardless of the operation, the error origination, and whether or not the resource supports before-imaging. For more information on this general error handling mechanism, see the getErrors( ) method on page 233 description.

Data type: `Object` or `string`

Access: Read-only

Applies to: request object on page 146

If a resource's Create, Read, Update, Delete (CRUD), or Submit operation returns successfully and the response is valid JSON that can be converted to a JavaScript object, the `response` property contains a reference to the resource's data that is returned from the server. If the server response is not valid JSON, the `response` property is `undefined`.

If a resource's Invoke operation returns successfully and has no return value or output parameters, the property is `null`. If the Invoke operation has a return value, you can read it as the value of the object's `_retVal` property. If the operation has output parameters, you can read these parameters as the values of object properties whose case-sensitive names and data types match the names and data types of the output parameters specified for the operation on the server.

If any CUD or Submit operation returns an error for an OpenEdge resource, the `response` property error object always contains the following properties for a resource that does **not** support before-imaging and **can** contain these properties for a resource that **supports** before-imaging:

- **`_retVal`** — A `string` with the value of any ABL `RETURN ERROR` string or `ReturnValue` property for a thrown `AppError` object
- **`_errors`** — An array of JavaScript objects, each of which contains two properties: `_errorMsg` with the ABL error message string and `_errorNum` with the error number, for one of possibly many ABL-returned errors

---

Note: This `response` property object can also return an error object with `_retVal` and `_errors` properties for an unhandled application server exception instead of an ABL application error.

If a given OpenEdge resource **does** support before-imaging, and the `response` property does not reference an error object for an unsuccessful result, information for an individual record-change errors (from CUD or Submit operations) is typically returned as part of the before-image data for each record object associated with the error, which you can return using the `getErrorString( )` method. For `after*` events of individual CUD operations, you can return this associated record object using the `jsrecord` property of the request object. In the callback for an `afterSaveChanges` event, or in any callback for a Promise object response to a Submit operation (`saveChanges(true)`) on a before-image resource, you can return the before-image data for every JSDO record object sent to the server using the `jsrecords` property of the request object.

If a CRUD operation returns an error on a Rollbase resource, the `response` property references a single `string` value (not an `Object`) with the error information.

Access to information on individual record-change error results depends on if the resource operation is a Create, Update, Delete (CUD), or a Submit operation and if the resource supports before-imaging. For more information, see the section on error handling in the description of the `saveChanges( )` method. See also the notes in this description of the `response` property.

The `response` property is available only for the following JSDO events or in the request object returned to a jQuery Promise callback:

- `afterCreate`
- `afterDelete`
- `afterFill`
- `afterInvoke`
- `afterSaveChanges`
- `afterUpdate`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe or register to JSDO events, any returned jQuery Promises, and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO. This object is also returned as the value of any JSDO invocation method that you execute synchronously.

Note: Error information returned with this property is most useful, and in one case is only possible, when the `autoApplyChanges` property is `false`, where you can individually reject or accept JSDO memory changes associated with each resource operation. When `autoApplyChanges` is `true`, the JSDO automatically rejects any JSDO memory changes associated with operation errors, but clears any error conditions and information associated with the operations only after the final `after*` event has been fired and handled (if handled) for the operation.

Note: Any any errors that result from a Data Object resource operation invoked by the most recent execution of the `fill( )` or `saveChanges( )` method remain available for return by the `getErrors( )` method until the next execution of `fill( )` or `saveChanges( )`, regardless of the setting of `autoApplyChanges`.

Note: If a given resource does not support before-imaging, any information for an individual record-change error from a CUD operation is typically returned in the `response` property, as described above. In the callback for an `afterSaveChanges` event, or in any callback for a Promise object, in response to executing `saveChanges(false)` (invoking CUD operations), you can also inspect the returned `response` property to find any error information, as described above.

See also:

autoApplyChanges property on page 202, fill( ) method on page 220, getErrors( ) method on page 233, getErrorString( ) method on page 239, invocation method on page 262, invoke( ) method on page 264, jsrecord property on page 272, jsrecords property on page 273, saveChanges( ) method on page 314, xhr property on page 358

saveChanges( ) method

Synchronizes to the server all record changes (creates, updates, and deletes) pending in JSDO memory for the current Data Object resource since the last call to the `fill( )` or `saveChanges( )` method, or since any prior changes have been otherwise accepted or rejected.

This method also causes an `offline` or `online` event to fire if it detects that there has been a change in the JSDO login session's online state.

This method always executes asynchronously and returns results (either or both) in subscribed JSDO event callbacks or in callbacks that you register using methods of a Promise object returned as the method value. A Promise object is always returned as the method value if jQuery Promises (or the exact equivalent) are supported in your development environment. Otherwise, this method returns `undefined`.

Return type: jQuery Promise or `undefined`

Applies to: progress.data.JSDO class on page 110

Working record: After execution, any prior working record settings for the tables of the JSDO are no longer set.

Syntax

```
saveChanges ( [ use-submit ] )
```

use-submit

An optional boolean parameter that when false (or not specified), tells saveChanges( ) to individually invoke all pending resource Create, Update, and Delete (CUD) operations, which are sent one record at a time across the network. You can use this option with a JSDO that accesses either a single-table or multi-table resource, with or without before-image support. Results for each record change are returned across the network to the JSDO after the operation on that record completes. The JSDO provides standard mechanisms to handle the results from each standard CUD operation (see the section on returning and handling results, below).

When true, this parameter tells saveChanges( ) to invoke a single Submit operation on the resource that handles any pending record-change (CUD) operations in a single network request. You can use this option **only** with a JSDO that accesses a single OpenEdge ProDataSet resource that supports before-imaging. (Rollbase resources do not support Submit.)

Note: With either type of operation (CUD or Submit) on an OpenEdge resource, the behavior of the operation on the server depends entirely on the Business Entity that implements the resource. To synchronize JSDO memory appropriately with the resource implementation, you must set the JSDO autoApplyChanges property and call the JSDO accept*Changes( ) and reject*Changes( ) as described later in this topic. The behavior of CUD operations on a Rollbase resource is provided by a standard Rollbase server API and works the same way for any Rollbase resource.

Promise method callback signatures

jQuery Promise objects define methods that register a callback function with a specific signature. The callback signatures depend on the method that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that saveChanges( ) returns:

Syntax:

```
promise.done( function ( jsdo , success , request ) )
promise.fail( function ( jsdo , success , request ) )
promise.always( function ( jsdo , success , request ) )
```

promise

A reference to the Promise object that is returned as the value of the saveChanges( ) method. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDO class on page 110.

jsdo

A reference to the JSDO that invoked the saveChanges( ) method (record-change operations) on its resource. For more information, see the description of the jsdo property on page 272 of the request object.

success

A boolean that is true if all record-change operations invoked on the resource by saveChanges( ) were successful. For more information, see the description of the success property on page 350 of the request object.

Note: It is not always necessary to test the value of *success* in a Promise method callback for the saveChanges( ) method, especially if the callback is registered using *promise*.done( ) and *promise*.fail( ), where the callback executes according to this value.

request

A reference to the request object returned after the saveChanges( ) method completed execution and returned all results from its record-change operations on the server. For more information, see the description of the request object on page 146.

Default CUD operation execution: one changed record at a time

Without *use-submit*, the saveChanges( ) method invokes the pending CUD operations in JSDO memory one record at a time across the network, and in the following general order of operation type:

1. **"Delete"** — All record deletions are applied.
2. **"Create"** — The creation of all new records is applied.
3. **"Update"** — Updates are applied to all modified records.

The sending of changes for multiple operations on the **same** record is optimized so the fewest possible changes are sent to the server. For example, if a record is first updated, then deleted in JSDO memory, only the deletion is sent to the server. However, note that all the changes to the record are applied to JSDO memory in a pending manner in case the deletion fails on the server.

If the JSDO is accessing an OpenEdge ProDataSet resource that is configured to support before-imaging, the JSDO also sends before-image data, along with each changed record, across the network to the server.

Note: Without *use-submit*, this method performs no batching of resource CUD operations. That is, the Delete operation is invoked over the network for each deleted record, followed by the Create operation for each created record, and finally by the Update operation for each updated record. So, even for a ProDataSet resource that supports before-imaging, each CUD operation executes over the network only one record at a time and cannot be part of a multi-record transaction.

Submit operation execution: multiple changed records at a time

If the JSDO is accessing an OpenEdge ProDataSet resource that supports before-imaging and the Submit operation, with *use-submit* specified as true, this method sends all pending Delete, Create, and Update record changes to the server in a single Submit operation. This Submit operation can thus manage a single, multi-record server transaction. For more information on the Data Object Submit operation, see Using JSDOs to create mobile and web clients on page 37.

In this case, all the record changes, along with the before-image data, are sent in the ProDataSet to the server, to be processed by a single ABL routine that implements the Submit operation. This allows all associated server CUD operations to be part of a single server transaction in which all pending record changes are attempted, with all results returned in the ProDataSet in a single network response, including any error information for each record change.

If you specify *use-submit* as true for a ProDataSet resource, the implementing Business Entity must be created with before tables for every temp-table that the ProDataSet manages and configured to support both before-image data and invocation of the Submit operation. Otherwise, this method throws an exception.

If you specify *use-submit* as `true` for a temp-table resource, this method throws an exception.

Returning and handling results

This method returns results asynchronously in two different ways, and in the following order, depending on the development environment used to build the mobile app:

1. **Using JSDO named events for all environments** — The following events fire before and after the `saveChanges( )` method executes, respectively, with results handled by callback functions that you subscribe as documented for each event:
   a. beforeSaveChanges event on page 210
   b. beforeDelete event on page 207
   c. afterDelete event on page 183
   d. beforeCreate event on page 206
   e. afterCreate event on page 181
   f. beforeUpdate event on page 211
   g. afterUpdate event on page 195
   h. afterSaveChanges event on page 192

2. **Using a Promise object returned for environments that support jQuery Promises** — Any callbacks that you register using Promise object methods all execute both **after** the `saveChanges( )` method itself and **after** all subscribed JSDO `after*` event callbacks complete execution. Note that the signatures of all Promise method callbacks match the signature of the `afterSaveChanges` event callback function so you can specify an existing `afterSaveChanges` event callback as the callback function that you register using any Promise method.

Because the callbacks that you register with any returned Promise methods execute only after all subscribed `after*` event callbacks complete, you can invoke logic in registered Promise method callbacks that can modify any processing done by the event callbacks.

Note that your programming requirements for any `afterCreate`, `afterUpdate`, `afterDelete` (CUD operation) event callback, and for any `afterSaveChanges` event or Promise method callback can be affected by your setting of the JSDO `autoApplyChanges` property.

Behavior of event and Promise callbacks when autoApplyChanges is true

If you set the `autoApplyChanges` property to `true` (the default setting) before you invoke `saveChanges(false)`, and a corresponding record-change (CUD) operation succeeds on the server, the JSDO automatically accepts and synchronizes the change in JSDO memory.

If any CUD operation fails, the operation is automatically undone and the JSDO rejects the change by reverting the applicable record in JSDO memory to its last-saved state. Specifically:

- If a Create operation fails, the record is removed from JSDO memory.
- If an Update operation fails, the record reverts to the state it was in immediately preceding the `assign( )` method invocation that led to the failure.
- If a Delete operation fails, the record is restored to JSDO memory in its last-saved state. This state does not reflect any unsaved `assign( )` method invocations that may have occurred before the `remove( )` call.

When the JSDO synchronizes JSDO memory for CUD operations, it uses any before-image data (if available) in each response from invoking the `saveChanges( )` method.

If you invoke a Submit operation (`saveChanges(true)`) on an OpenEdge ProDataSet resource with before-image support, JSDO memory is automatically updated, with changes accepted or rejected based on the success or failure of each record create, update, and delete included in the Submit operation request. If the Submit returns with a network or server error that prevented the operation from being invoked on the server, all changes in JSDO memory associated with the Submit are rejected.

Note: For a Submit operation on a before-image resource, if you want all such record changes rejected if even one record change returns an error, set `autoApplyChanges` to `false` and explicitly reject all record changes based on the returned error condition. For more information on behavior when `autoApplyChanges` is `false`, see Behavior of event and Promise callbacks when autoApplyChanges is false on page 318 in this `saveChanges( )` method description.

When `autoApplyChanges` is `true`, the JSDO automatically clears any error conditions set for the affected record changes only **after** the record changes have all been rejected and undone and **after** all registered `after*` event and Promise method callbacks have executed. If any callback executes with unsuccessful results, you can call the JSDO `getErrors( )` method on a table reference to return any errors from the resource operation or operations associated with the specified JSDO table. For more information, see Error handling on page 319 in this `saveChanges( )` method description.

Note: The errors from CUD or Submit operations that you return using `getErrors( )` remain available for return until the next invocation of `fill( )` or `saveChanges( )`, regardless of the `autoApplyChanges` setting. For more information, see the getErrors( ) method on page 233 description.

Note: You can manually inspect error conditions and information as part of `after*` event and *`promise`*`.fail( )` callback execution when `autoApplyChanges` is `true`. However, checking for such error information might not be as useful as when `autoApplyChanges` is `false`, because all JSDO memory changes associated with record-change errors are automatically undone.

Caution: Use care when taking any action within event or Promise method callbacks that might interfere with the automatic acceptance or rejection of pending record changes. If you want to manually manage the handling of pending record changes in response to resource operations in these callbacks, consider setting `autoApplyChanges` to `false` before invoking `saveChanges( )`.

Behavior of event and Promise callbacks when autoApplyChanges is false

If you set the `autoApplyChanges` property to `false` before you invoke `saveChanges( )`, you must use one of the following methods to manually accept or reject any record changes in order to synchronize JSDO memory with operation results from the server. Use the method that is appropriate for the resource operation and the JSDO event or Promise method callback where you manage these changes:

- `acceptChanges( )`
- `acceptRowChanges( )`
- `rejectChanges( )`
- `rejectRowChanges( )`

Depending on the success of the particular resource operation and the JSDO `after*` event or Promise method callback in which you respond to operation results, you can check returned request object properties to determine what `accept*( )` or `reject*( )` method to call. If any callback executes with unsuccessful results, you can call the JSDO `getErrors( )` method on a table reference to return any errors from the resource operation or operations associated with the specified JSDO table.

Note: The errors from CUD or Submit operations that you return using `getErrors( )` remain available for return until the next invocation of `fill( )` or `saveChanges( )`, regardless of the `autoApplyChanges` setting. For more information, see the getErrors( ) method on page 233 description.

If you are handling results for one or more record-change (CUD) operations in an `afterSaveChanges` event callback, or in a Promise method callback, in response to calling `saveChanges(false)` (without using Submit), you can use the `batch` property of the request object to evaluate the results of each CUD operation invoked by `saveChanges( )`.

If you are handling results of a Submit operation (`saveChanges(true)`) on an OpenEdge ProDataSet resource with before-imaging, in an `afterSaveChanges` event or Promise method callback you can use the returned `jsrecords` property of the request object to help evaluate the results of each attempted record-change.

If you want to evaluate the results of any single record-change operation invoked by `saveChanges(false)` (a CUD operation with or without before-image support), or `saveChanges(true)` (a Submit operation) with before-image support, you can inspect the request object returned in an appropriate `after*` event callback that you have registered for each CUD operation, where you can identify the associated record change using the `jsrecord` property.

Note that the `acceptChanges( )` and `rejectChanges( )` methods might not be as useful for synchronizing JSDO memory with the server as the corresponding `acceptRowChanges( )` and `rejectRowChanges( )` methods, which you can invoke selectively in response to the results of individual record-change operations. The `acceptChanges( )` and `rejectChanges( )` methods are most useful for accepting or rejecting all record changes in JSDO memory based on the results of a single server transaction (invoked using Submit with before-image support) that either succeeds and applies all server record changes or fails if even one record change fails and undoes all server record changes as part of undoing the transaction. For more information, see the reference description for each method.

For more information on handling errors from failed record-change operations, see Error handling on page 319 in this `saveChanges( )` method description.

Error handling

You can use two mechanisms for retrieving information about any failure (error) in the most recently executed `saveChanges( )` call and the record changes (with or without before-image support) that might be discarded as a result:

- Call the `getErrors( )` method on any or all JSDO table references involved in the operation. For more information, see the getErrors( ) method on page 233 description.

  Note: Updated to support all possible types of error returns in Progress Data Objects 4.3 or later.

- Access properties on the *`request`* parameter object returned by an event or Promise method callback that is executed in response to the operation. Use this mechanism to return error information for specific cases, for example, to return related data for an error using the `jsrecord` or `jsrecords` property. For more information, see the request object on page 146 description.

An error has occurred when the *`success`* parameter of the callback or the `success` property of the returned request object is `false`. (Note that for certain Promise method callbacks, such as those registered by *`promise`*`.done( )` and especially *`promise`*`.fail( )` for error handling, there is no need to check the value of the *`success`* parameter or `success` property passed to the callback, because the Promise method callback itself executes based on this value.)

The request object properties available for retrieving error information depend on whether the Data Object resource supports before-image (BI) data and on what `after*` event or Promise method callback is currently executing for the resource operation, or operations, that you are handling. Note that regardless of resource support for BI data, or the currently executing callback, the `getErrors( )` method **always** returns all error information associated with the most recently invoked CUD or Submit operations invoked as part of the most recent `saveChanges( )` call.

Note: These error handling options are most useful when the `autoApplyChanges` property is `false`. When `autoApplyChanges` is `true`, the method automatically accepts or rejects record changes as previously described for this property setting, and clears all associated error conditions and information either after the final `after*` event fires and executes its final subscribed callback for a given operation or after the final callback registered by a Promise method executes, whichever one executes last. Note that **even after** the errors are cleared, the `getErrors( )` method continues to return the errors associated with the most recently executed `saveChanges( )` call until the next `fill( )` or `saveChanges( )` call.

Thus, for a resource that:

- ***Does not support*** **before-image data with individual CUD record-change operations by `saveChanges(false)`** — To retrieve discarded changes and error information returned as part of the response to an:
  - **`after*` event fired for each CUD operation request invoked on a single record of data only** — **Either** call `getErrors( )` **or** inspect the `response` and `xhr` properties of the callback *`request`* parameter object for any error information. Inspect the callback *`record`* parameter or the `jsrecord` property (the record object) of the returned request object for information on operation data.
  - **`afterSaveChanges` event fired or a *`promise`*`.fail( )` method callback executed after all individual CUD operations on one or more records of data complete** — Call `getErrors( )` and inspect the `batch` property of the callback *`request`* parameter object to identify and retrieve any additional information and data returned for each CUD operation executed as part of invoking `saveChanges(false)`.
- ***Supports*** **before-image data with individual CUD record-change operations executed by `saveChanges(false)` (without using Submit)** — To retrieve discarded changes and error information returned as part of a before-image response to an:
  - **`after*` event fired for a specific CUD operation on a single record of data** — **Either** call `getErrors( )` and inspect the callback *`record`* parameter or the `jsrecord` property (the record object) of the returned request object for information on the operation data, **or** inspect the `response` and `xhr` properties of the callback *`request`* parameter object for any error information. If you inspect the `response` and `xhr` properties and no error information is found, inspect the callback *`record`* parameter or the `jsrecord` property (the record object) of the returned request object. As part of this inspection, check the return value of the `getErrorString( )` method called on the record object; a value other than `null` or `undefined` returns an error response for the corresponding before-image CUD operation executed on the server.
  - **`afterSaveChanges` event fired or a *`promise`*`.fail( )` method callback executed for CUD operations on one or more records of data** — Call `getErrors( )` and inspect the `batch` property of the callback *`request`* parameter object to identify and retrieve any additional information and data returned for each CUD operation executed as part of invoking `saveChanges(false)`.
- ***Supports*** **before-image data with a single Submit operation on one or more records of data executed by `saveChanges(true)`** — To retrieve discarded changes and error information returned as part of a before-image response from an:

- **`after*` event fired for each record-change (CUD) as part of the single Submit operation — Either** call `getErrors( )` and inspect the callback *`record`* parameter or the `jsrecord` property (the record object) of the returned request object for information on the operation data, **or** inspect the `response` and `xhr` properties of the callback *`request`* parameter object for any error information on the specific record change. If you inspect the `response` and `xhr` properties and no error information is found, inspect the callback *`record`* parameter or `jsrecord` property (the record object) of the request object returned. As part of this inspection, check the return value of the `getErrorString( )` method called on the record object; a value other than `null` or `undefined` returns an error response for this record change as part of the Submit operation executed on the server.
- **`afterSaveChanges` event fired or a *`promise`*`.fail( )` method (or equivalent) callback executed for the single Submit operation — Either** call `getErrors( )` **or** inspect the `response` and `xhr` properties of the callback *`request`* parameter object for any error information on the entire Submit operation.

  If you call `getErrors( )` to return Submit operation errors, using any associated record ID information, you can also search the record objects returned by the `jsrecords` property of the callback *`request`* parameter object to identify corresponding record changes and inspect their before-image data.

  If you inspect the `response` and `xhr` properties, and no error information is found, inspect the `jsrecords` property to access **all** the record objects that were targets of the Submit operation and their corresponding record changes and before-image data. As part of this inspection, check the return value of the `getErrorString( )` method called on each record object; a value other than `null` or `undefined` returns an error response for this record change as part of the Submit executed on the server.

Examples

The following examples show the calling of `saveChanges( )` for various types of Data Object resources, with and without before-image support, and using different callback mechanisms to handle the results.

The following code fragment shows a JSDO created for an OpenEdge ProDataSet resource with before-imaging and its `autoApplyChanges` property set to `false` so it **does not** automatically accept or reject changes to data in JSDO memory after a call to the `saveChanges( )` method. Instead, it subscribes a single callback for each of the `afterDelete`, `afterCreate`, and `afterUpdate`, events to determine if changes to any `eCustomer` table record in JSDO memory should be accepted or rejected based on the success of the corresponding resource operation on the server. To change the data for a record, a jQuery event is defined on an update button to update the corresponding `eCustomer` record in JSDO memory with the current field values entered in a customer detail form (`#custdetail`):

Table 32: Example: saveChanges( ) using Submit with before-imaging that accepts/rejects changes based on CUD events

```
dataSet = new progress.data.JSDO( { name: 'dsCustomerOrder',
                                                autoApplyChanges : false } );
dataSet.eCustomer.subscribe('afterDelete', onAfterCustomerChange, this);
dataSet.eCustomer.subscribe('afterCreate', onAfterCustomerChange, this);
dataSet.eCustomer.subscribe('afterUpdate', onAfterCustomerChange, this);

$('#btnUpdate').bind('click', function(event) {
  var jsrecord = dataSet.eCustomer.findById($('#custdetail #id').val());
  if (jsrecord) {jsrecord.assign();};
});

// Similar controls might be defined to delete and create eCustomer records...

$('#btnCommit').bind('click', function(event) {
  dataSet.saveChanges(true); // Invokes the resource Submit operation
});
```

```
function onAfterCustomerChange(jsdo, record, success, request) {
  if (success)  {
    record.acceptRowChanges();
    // Perform other actions associated with accepting this record change
  }
  else
  {
    record.rejectRowChanges();
    // Perform other actions associated with rejecting this record change
  }
}
```

When the button is clicked, the event handler uses the `findById( )` method to find the original record with the matching internal record ID (`#id`) and invokes the `assign( )` method on `jsrecord` with an empty parameter list to update its fields in `eCustomer` with any new values entered into the form. You might define similar events and controls to delete the `eCustomer` record or add a new `eCustomer` record.

A jQuery event also defines a commit button that when clicked invokes the `saveChanges( )` method, passing `true` as the *`use-submit`* parameter value, to apply all pending changes in JSDO memory on the server in a single network request. Using this parameter, all pending record deletes, creates, and updates, including before-image data, are sent to the server in a single ProDataSet as input to a Submit operation. This operation processes all the changes for each record delete, create, or update on the server, storing the results in the same ProDataSet, including any errors, for output to the client in a single network response.

After the method completes, and all results have been returned to the client from the server, the appropriate event for each resource Delete, Create, or Update operation fires even though multiple changes can be sent using a single Submit operation. If the operation **was** successful, the event handler calls `acceptRowChanges( )` to accept the `eCustomer` record change associated with the event in JSDO memory. If the operation **was not** successful, the event handler calls `rejectRowChanges( )` to reject the `eCustomer` record change. An advantage of using an event to manually accept or reject a record change is that you can perform other actions associated with this particular change, such as creating a local log that describes the change or reports the error.

The following examples show similar processing of results on an OpenEdge resource, first using an `afterSaveChanges` event callback with and without before-imaging, then using equivalent Promise method callbacks for the resource without before-imaging. The final example shows Promise callback handling of a Rollbase resource. The instantiation of the JSDO login session and the JSDO referenced by `myjsdo` is assumed.

The following code fragment subscribes the callback function, `onAfterSaveChanges`, to handle the `afterSaveChanges` event fired on the JSDO, `myjsdo` after invoking the Submit operation for an OpenEdge ProDataSet resource with a `ttCustomer` table resource and before-image support:

Table 33: Example: saveChanges( ) using Submit on an OpenEdge ProDataSet resource and handled using the afterSaveChanges event

```
/* subscribe to event */
myjsdo.subscribe( 'afterSaveChanges', onAfterSaveChanges );

/* some code that initiates multiple CUD operations before
   sending them to the server */
var newrec = myjsdo.add();

. . .

var jsrecord = myjsdo.findById(myid);
if (jsrecord) {jsrecord.remove();};

/* call to saveChanges() using Submit with event handling
```

```
   and with autoApplyChanges set to true by default */
myjsdo.saveChanges(true);

function onAfterSaveChanges( jsdo , success , request ) {
    var lenErrors,
        errors,
        errorType;

    if (success) {
        /* all record changes invoked by saveChanges()Submit succeeded */
        /* for example, accept all changes and redisplay records in the JSDO table */
        jsdo.foreach( function(jsrecord) {
            /* reference the record/field as jsrecord.data.<fieldName> . . . */

        });
    }
    else { // success = false
        /* handle Submit operation errors where either the server request
           or one or more record changes might have failed */
        console.log("Operation: Submit");
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) { /* Each error */
            console.log(JSON.stringify(errors[idxError]));
        } /* for each error message */
    }
};
```

In the above example, the `autoApplyChanges` property is set to `true` by default to allow `saveChanges( )` to save or reject all changes automatically based on the success of the Submit operation. If the method completes successfully, the JSDO simply accepts all changes in JSDO memory and re-displays the latest data from the records in the JSDO. If it completes unsuccessfully, the JSDO accepts or rejects each record change in JSDO memory based on the success of each submitted record change **after** the subscribed event callback has executed.

Note: For a Submit operation, you can determine the success of each record change in the `onAfterSaveChanges` callback based on the result of calling the JSDO `getErrorString( )` method on each record object that you return from the value of the `request.jsrecords` array property.

Also, if the method completes unsuccessfully, the event callback accesses any server exception information by calling the `getErrors( )` method on the single table in the JSDO (`jsdo` parameter). It concatenates any errors found and logs them to the console, possibly in preparation for having the user re-enter their changes. In this case, the error object returned by `getErrors( )` for each error is converted to a string in the form of a JSON object using the standard `JSON.stringify( )` method.

Note: These messages are accessible by `getErrors( )` until the next call to `fill( )` or `saveChanges( )`.

The following code fragment subscribes the callback function, `onAfterSaveChanges`, to handle the `afterSaveChanges` event fired on the JSDO, `myjsdo`, for an OpenEdge single-table resource, `ttCustomer`, without before-imaging:

Table 34: Example: saveChanges( ) on an OpenEdge resource without before-imaging using the afterSaveChanges event

```
/* subscribe to event */
myjsdo.subscribe( 'afterSaveChanges', onAfterSaveChanges );
```

```
/* some code that initiates multiple CUD operations before
   sending them to the server */
var newrec = myjsdo.add();

. . .

var jsrecord = myjsdo.findById(myid);
if (jsrecord) {jsrecord.remove();};

/* call to saveChanges() and event handling */
myjsdo.autoApplyChanges = false;
myjsdo.saveChanges();

function onAfterSaveChanges( jsdo , success , request ) {

    var operations = request.batch.operations,
        len = operations.length, /* number of resource operations invoked */
        lenErrors,
        errors,
        errorType;

    if (success) {
        /* all resource operations invoked by saveChanges() succeeded */
        /* for example, redisplay records in the JSDO table */
        jsdo.acceptChanges();
        jsdo.foreach( function(jsrecord) {
            /* reference the record/field as jsrecord.data.<fieldName> . . . */

        });
    }
    else {
        /* one or more resource operations invoked by saveChanges() failed */
        /* handle all operation error messages */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.DATA_ERROR:
                    errorType = "Server Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
                case default:
                    errorType = null; // Unexpected errorType value
                    break;
            }
            if (errorType) { /* log all error text
                console.log("ERROR: " + errorType + errors[idxError].error);
                if (errors[idxError].id) { /* error with record object */
                    console.log("RECORD ID: " + errors[idxError].id);
                    /* possibly log the data values for record with this ID */
                }
                if (errors[idxError].responseText) {
                    console.log("HTTP FULL TEXT: "
                                + errors[idxError].responseText);
                }
            }
            else { /* unexpected errorType */
                console.log("UNEXPECTED ERROR TYPE: "
```

```
                                + errors[idxError].type);
            }
        } /* for each error message */

        /* accept or reject record changes based on each operation success */
        for(var idx = 0; idx < len; idx++) {
            var operationEntry = operations[idx];
            if (operationEntry.success) {
                operationEntry.jsrecord.acceptRowChanges();
            }
            else {
                operationEntry.jsrecord.rejectRowChanges();
            }
        } /*for each CUD operation */

    }
};
```

In the above example, the `autoApplyChanges` property is dynamically set to `false` prior to calling `saveChanges( )` to allow custom handling of the results. If the method completes successfully, the callback simply accepts all changes in JSDO memory and re-displays the latest data from the records in the JSDO.

If the method completes unsuccessfully, the callback logs the error messages (`jsdo.ttCustomer.getErrors()`) returned for all the operations invoked by `saveChanges( )` that failed. In this case, the property values from each error object returned by `getErrors( )` are formatted and logged based on the error type for a more readable and informative output than a simple string representation of the JSON object. Finally, it accesses the `batch` property of the request object returned for the `afterSaveChanges` event in order to inspect the request object for each resource operation (`operations[idx]`) that was invoked by `saveChanges( )`. It then accepts or rejects the record changes for each operation record (`operationEntry.jsrecord`), depending on the operation success.

Note: These messages are accessible by `getErrors( )` until the next call to `fill( )` or `saveChanges( )`.

Note: For a Delete operation error, no data is available in `operationEntry.jsrecord` because the operation record has already been cleared from JSDO local memory. However, after executing `operationEntry.jsrecord.rejectRowChanges( )` to undo the operation on the JSDO, the record is restored to JSDO local memory as expected.

The following code fragment registers `done( )` and `fail( )` method callbacks on the Promise returned by a `saveChanges( )` call on a JSDO, `myjsdo`, which provides identical handling of operation results for the same OpenEdge resource handled in the previous example using the `afterSaveChanges` event callback:

Table 35: Example: saveChanges( ) on an OpenEdge resource using a Promise object

```
/* some code that initiates multiple CUD operations before
   sending them to the server */
var newrec = myjsdo.add();

. . .

var jsrecord = myjsdo.findById(myid);
if (jsrecord) {jsrecord.remove();};

/* call to saveChanges() with inline-coded Promise callback handling */
myjsdo.autoApplyChanges = false;
myjsdo.saveChanges().done(
    function( jsdo, success, request ) {
```

```
        /* all resource operations invoked by saveChanges() succeeded */
        /* for example, redisplay records in the JSDO table */
        jsdo.acceptChanges();
        jsdo.foreach( function(jsrecord) {
            /* reference the record/field as jsrecord.data.<field-ref> . . . */

        });
    }).fail(
    function( jsdo, success, request ) {
        /* one or more resource operations invoked by saveChanges() failed */
        var operations = request.batch.operations,
            len = operations.length, /* number of resource operations invoked */
            lenErrors,
            errors,
            errorType;

        /* handle all operation error messages */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.DATA_ERROR:
                    errorType = "Server Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
                case default:
                    errorType = null; // Unexpected errorType value
                    break;
            }
            if (errorType) { /* log all error text
                console.log("ERROR: " + errorType + errors[idxError].error);
                if (errors[idxError].id) { /* error with record object */
                    console.log("RECORD ID: " + errors[idxError].id);
                    /* possibly log the data values for record with this ID */
                }
                if (errors[idxError].responseText) {
                    console.log("HTTP FULL TEXT: "
                                + errors[idxError].responseText);
                }
            }
            else { /* unexpected errorType */
                console.log("UNEXPECTED ERROR TYPE: "
                            + errors[idxError].type);
            }
        } /* for each error message */

        /* accept or reject record changes based on each operation success */
        for(var idx = 0; idx < len; idx++) {
            var operationEntry = operations[idx];
            if (operationEntry.success) {
                operationEntry.jsrecord.acceptRowChanges();
            }
            else {
                operationEntry.jsrecord.rejectRowChanges();
            }
        } /*for each CUD operation */

    });
```

In the above example, the same processing occurs for successful and unsuccessful execution of `saveChanges( )`, but there is no need to test the `success` parameter of the `done( )` and `fail( )` Promise method callbacks, because they each execute in response to this value.

Note: For a Delete operation error, no data is available in `operationEntry.jsrecord` because the operation record has already been cleared from JSDO local memory. However, after executing `operationEntry.jsrecord.rejectRowChanges( )` to undo the operation on the JSDO, the record is restored to JSDO local memory as expected.

However, if your code already uses the `afterSaveChanges` event implementation, and you want to quickly change it to use Promises, you can simply register the original `onAfterSaveChanges` function as the callback for the `always( )` Promise method, as in the following code fragment:

Table 36: Example: saveChanges( ) using a Promise object that registers an afterSaveChanges callback

```
/* some code that initiates multiple CUD operations before
   sending them to the server */
var newrec = myjsdo.add();

. . .

var jsrecord = myjsdo.findById(myid);
if (jsrecord) {jsrecord.remove();};

/* call to saveChanges() with Promise callback handling using
   the original afterSaveChanges event handler */
myjsdo.autoApplyChanges = false;
myjsdo.saveChanges().always(onAfterSaveChanges);

function onAfterSaveChanges( jsdo , success , request ) {

    var operations = request.batch.operations,
        len = operations.length, /* number of resource operations invoked */
        lenErrors,
        errors,
        errorType;

    if (success) {
        /* all resource operations invoked by saveChanges() succeeded */
        /* for example, redisplay records in the JSDO table */
        jsdo.acceptChanges();
        jsdo.foreach( function(jsrecord) {
            /* reference the record/field as jsrecord.data.<fieldName> . . . */

        });
    }
    else {
        /* one or more resource operations invoked by saveChanges() failed */
        /* handle all operation error messages */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.DATA_ERROR:
                    errorType = "Server Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
```

```
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
                case default:
                    errorType = null; // Unexpected errorType value
                    break;
            }
            if (errorType) { /* log all error text
                console.log("ERROR: " + errorType + errors[idxError].error);
                if (errors[idxError].id) { /* error with record object */
                    console.log("RECORD ID: " + errors[idxError].id);
                    /* possibly log the data values for record with this ID */
                }
                if (errors[idxError].responseText) {
                    console.log("HTTP FULL TEXT: "
                                + errors[idxError].responseText);
                }
            }
            else { /* unexpected errorType */
                console.log("UNEXPECTED ERROR TYPE: "
                            + errors[idxError].type);
            }
        } /* for each error message */

        /* accept or reject record changes based on each operation success */
        for(var idx = 0; idx < len; idx++) {
            var operationEntry = operations[idx];
            if (operationEntry.success) {
                operationEntry.jsrecord.acceptRowChanges();
            }
            else {
                operationEntry.jsrecord.rejectRowChanges();
            }
        } /*for each CUD operation */

    }
};
```

Note: For a Delete operation error, no data is available in `operationEntry.jsrecord` because the operation record has already been cleared from JSDO local memory. However, after executing `operationEntry.jsrecord.rejectRowChanges( )` to undo the operation on the JSDO, the record is restored to JSDO local memory as expected.

Finally, the following code fragment registers `done( )` and `fail( )` method callbacks on the Promise returned by a `saveChanges( )` call on a JSDO, `myjsdo`, that is already instantiated for a Rollbase object resource:

Table 37: Example: saveChanges( ) on a Rollbase resource using a Promise object

```
/* some code that initiates multiple CUD operations before
   sending them to the server */
var newrec = myjsdo.add();

. . .

var jsrecord = myjsdo.findById(myid);
if (jsrecord) {jsrecord.remove();};

/* call to saveChanges() with Promise callback handling */
myjsdo.autoApplyChanges = false;
myjsdo.saveChanges().done(
    function( jsdo, success, request ) {
        /* all resource operations invoked by saveChanges() succeeded */
```

```
        /* for example, redisplay records in the JSDO table */
        jsdo.acceptChanges();
        jsdo.foreach( function(jsrecord) {
            /* reference the record/field as jsrecord.data.<field-ref> . . . */

        });
    }).fail(
    function( jsdo, success, request ) {
        /* one or more resource operations invoked by saveChanges() failed */
        var operations = request.batch.operations,
            len = operations.length, /* number of resource operations invoked */
            lenErrors,
            errors,
            errorType;

        /* handle all operation error messages */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.ERROR:
                    errorType = "Rollbase Error: ";
                    break;
                case default:
                    errorType = null; // Unexpected errorType value
                    break;
            }
            if (errorType) { /* log all error text
                console.log("ERROR: " + errorType + errors[idxError].error);
                if (errors[idxError].id) { /* error with record object */
                    console.log("RECORD ID: " + errors[idxError].id);
                    /* possibly log the data values for record with this ID */
                }
                if (errors[idxError].responseText) {
                    console.log("HTTP FULL TEXT: "
                                + errors[idxError].responseText);
                }
            }
            else { /* unexpected errorType */
                console.log("UNEXPECTED ERROR TYPE: "
                            + errors[idxError].type);
            }
        } /* for each error message */

        /* accept or reject record changes based on each operation success */
        for(var idx = 0; idx < len; idx++) {
            var operationEntry = operations[idx];
            if (operationEntry.success) {
                operationEntry.jsrecord.acceptRowChanges();
            }
            else {
                operationEntry.jsrecord.rejectRowChanges();
            }
        } /*for each CUD operation */

    });
```

For a Rollbase resource, any error results from a resource CUD operation are returned by `getErrors( )` as error type `progress.data.JSDO.ERROR` with the `errors[idxError].error` property set to the returned `string` value. This includes results from both CUD operation errors and server errors.

Note: For a Delete operation error, no data is available in `operationEntry.jsrecord` because the operation record has already been cleared from JSDO local memory. However, after executing `operationEntry.jsrecord.rejectRowChanges( )` to undo the operation on the JSDO, the record is restored to JSDO local memory as expected.

See also:

acceptChanges( ) method on page 153, acceptRowChanges( ) method on page 156, autoApplyChanges property on page 202, batch property on page 205, data property on page 217, fill( ) method on page 220, getErrors( ) method on page 233, getErrorString( ) method on page 239, invocation method on page 262, jsrecord property on page 272, jsrecords property on page 273, rejectChanges( ) method on page 305, rejectRowChanges( ) method on page 308, response property on page 312, success property on page 350

saveLocal( ) method

Saves JSDO memory to a specified local storage area, including pending changes and any before-image data, according to a specified data mode.

Return type: `undefined`

Applies to: progress.data.JSDO class on page 110

Syntax

```
saveLocal ( [ storage-name [ , data-mode ] ] )
saveLocal ( data-mode )
```

storage-name

The name of the local storage area in which to save the specified data from JSDO memory. If *storage-name* is not specified, blank, or `null`, the name of the default storage area is used. The name of this default area is `jsdo_`*serviceName*`_`*resourceName*, where *serviceName* is the name of the Data Object Service that supports the JSDO instance, and *resourceName* is the name of the resource (table, dataset, etc.) for which the JSDO instance is created.

data-mode

A JSDO class constant that specifies the data in JSDO memory to be saved to local storage. Each data mode initially clears the specified local storage area of all its data, then replaces it with the data from JSDO memory as specified. The possible values include:

- **`progress.data.JSDO.ALL_DATA`** — Replaces the data in the storage area with **all** the data from JSDO memory, including pending changes and any before-image data. This is the default data mode.
- **`progress.data.JSDO.CHANGES_ONLY`** — Replaces the data in the storage area with **only** the pending changes from JSDO memory, including any before-image data.

If this method encounters any errors, it leaves the specified storage area unchanged and throws an exception.

This method supports any schema that the JSDO supports.

Note: If you want to save JSDO memory to local storage after the JSDO `saveChanges( )` method fails in response to an offline condition, be sure to set the `autoApplyChanges` property on the JSDO to `false` before calling this method for the first time. You can then continue to save JSDO memory to protect against a local session failure as it accumulates further offline changes until the mobile app goes back online and `saveChanges( )` succeeds in saving the changes to the server.

You can also use this method to routinely cache static data, such as state and rate tables, that might not change often, allowing for faster startup of the mobile app. One way to do this is to define a JSDO for a resource that accesses only static data, and invoke this method after refreshing JSDO memory using the `fill( )` method. When caching data in general, be sure to save JSDO memory when it is in a consistent state, such as immediately after successful invocation of the JSDO `fill( )` or `saveChanges( )` method (in the relatively rare case where routinely static data is being updated).

Example

The following code fragment caches JSDO memory for a JSDO defined with static data immediately after it is loaded into JSDO memory:

```
dataSet = new progress.data.JSDO( 'dsStaticData' );
dataSet.fill();
dataSet.saveLocal();
```

In this case, all the data in JSDO memory is cached to the default storage area.

See also:

acceptChanges( ) method on page 153, autoApplyChanges property on page 202, fill( ) method on page 220, readLocal( ) method on page 304, rejectChanges( ) method on page 305, saveChanges( ) method on page 314

services property

Returns an array of objects that identifies the Data Object Services that have been loaded for the current `JSDOSession` or `Session` object and its web application.

Data type: Array of `Object`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 124, progress.data.Session class on page 136

You load Data Object Services for a `JSDOSession` or `Session` object by loading the corresponding Data Service Catalogs using the object's `addCatalog( )` method. You can load these Catalogs either before or after you log into a web application using the object's `login( )` method.

Each object in the array returned by this session property contains two properties:

- **`name`** — The name of a Data Object Service that is loaded for the specified session.
- **`uri`** — The URI for the service. If the address of the service in the Catalog is an absolute URI, this value is that URI. If the service address is relative, this value is the relative address concatenated to the value of the `JSDOSession` or `Session` object's `serviceURI` property, which contains the web application URI used by the object's `login( )` method.

Note: To return a corresponding list of URIs for the loaded Data Service Catalogs, read the `catalogURIs` property.

Example

Given two Catalogs, `CustomerSvc.json` and `ItemSvc.json`, that specify their services this way:

- **`"CustomerSvc"`** service with this URI: **`"/rest/CustomerSvc"`**
- **`"ItemSvc"`** service with this URI: **`"http://itemhost:8080/SportsApp/rest/ItemSvc"`**

The following code fragment produces the output that follows:

```
// create Session
pdsession = new progress.data.JSDOSession('http://custhost:8080/SportsApp');

window.loginView = kendo.observable({
    submit: function() {

        // load 2 unprotected Catalogs
        pdsession.addCatalog([
              "http://custhost:8080/SportsApp/static/CustomerSvc.json",
              "http://itemhost:8080/SportsApp/static/ItemSvc.json" ]
            ).done(
            function(JSDOsession, result, info) {

                // log in anonymously
                JSDOSession.login(
                    ).done(
                    function(JSDOSession, result, info) {

                        /* Use services property to print services
                           loaded by this Session object */
                        for (var i=0; i < JSDOSession.services.length; i++) {
                            console.log( JSDOSession.services[i].name + "   "
                                         + JSDOSession.services[i].uri );
                        }
                        // Stuff with widgets and DataSources . . .
                    }).fail( // log in
                    function(JSDOSession, result, info) { . . .
                    });
            }).fail( // Add Catalogs
            function(JSDOSession, result, details) { . . .
            });
    }
});
```

Output from the preceding code fragment:

```
CustomerSvc   http://custhost:8080/SportsApp/rest/CustomerSvc
ItemSvc   http://itemhost:8080/SportsApp/rest/ItemSvc
```

See also:

addCatalog( ) method (JSDOSession class) on page 159, addCatalog( ) method (Session class) on page 164, catalogURIs property on page 215, login( ) method (JSDOSession class) on page 275, login( ) method (Session class) on page 280, serviceURI property on page 333

serviceURI property

Returns the URI to the web application that has been passed as an option to the class constructor for the current `JSDOSession` object or that has been passed as a parameter to the most recent call to the `login( )` method on the current `Session` object, whether or not the most recent call to `login( )` succeeded.

Data type: `string`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 124, progress.data.Session class on page 136

See also:

getSession( ) stand-alone function on page 248, login( ) method (JSDOSession class) on page 275, login( ) method (Session class) on page 280

setDetailPage( ) method

Specifies the HTML page that contains the form in which item details are displayed.

Return type: `null`

Applies to: progress.ui.UIHelper class on page 142

Syntax

```
uihelper-ref.setDetailPage( object )
uihelper-ref.table-ref.setDetailPage( object )
```

uihelper-ref

A reference to a `UIHelper` instance. You can call the method on *uihelper-ref* if the JSDO associated with the instance has only a single table reference.

table-ref

A table reference on the JSDO associated with the `UIHelper` instance, and that table reference has the records to be displayed in the form.

object

A JavaScript object with the following properties:

- **`name`** — A `string` value set to the `id` attribute of the HTML `div` (or other) element that contains the form page.
- **`fieldTemplate`** — (Optional) A `string` value set to an HTML string specifying the fields and labels (and their format) that override, for the specified form, the default field template settings or any field template settings specified by the `setFieldTemplate( )` method.

Example

The following code fragment sets the form page to the HTML element with an `id` attribute value of `'cust-detail-page'` and a single form field:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );
uihelper = new progress.ui.UIHelper({ jsdo: dataSet });
uiHelper.eCustomer.setDetailPage({
    name: 'cust-detail-page',
    fieldTemplate: '<input id="{__name__}"></input>'
});
```

See also:

getFormFields( ) method on page 240, getFormRecord( ) method on page 241, setFieldTemplate( ) method on page 334

setFieldTemplate( ) method

A class method that specifies a new default field format for all detail forms created with `UIHelper` instances in the current JavaScript session.

These settings can be overridden for specific forms by the optional `fieldTemplate` property of the JavaScript object passed to the `setDetailPage( )` method.

Return type: `null`

Applies to: progress.ui.UIHelper class on page 142

Syntax

```
progress.ui.UIHelper.setFieldTemplate( field-template )
```

field-template

A `string` value set to an HTML string that specifies new default field formats for detail forms used by `UIHelper` instances in the current JavaScript session.

It is not necessary to call `setFieldTemplate( )` unless one of the following is true:

- You are not using a jQuery Mobile environment.
- You are using a jQuery Mobile environment, but you want to change the default settings provided with the `progress.ui.UIHelper` class (see below).

The `progress.ui.UIHelper` class provides a default form template for use in all jQuery Mobile environments.

Example

The following code fragment shows the `setFieldTemplate( )` method setting the installed default field template values for the `progress.ui.UIHelper` class. Notice that the parameter is a single string with the field values for the template. In this particular case, it is a long string:

```
progress.ui.UIHelper.setFieldTemplate( '<div data-role="fieldcontain">
    <label for="{__name__}">{__label__}</label>
    <input id="{__name__}" name="{__name__}"
           placeholder="" value="" type="text" />
  </div>' );
```

As shown in the preceding example with the class default setting, the template string uses the following substitution parameters:

- **`{__name__}`** — The name of the field, as defined by the schema
- **`{__label__}`** — The title property of the field, as defined by the schema

See also:

setDetailPage( ) method on page 333

setItemTemplate( ) method

A class method that specifies a new default item format for all list views created with `UIHelper` instances in the current JavaScript session.

These settings can be overridden for specific list views by the `setListView( )` method. In addition, the `__format__` parameter can be overridden for specific items by the `addItem( )` method.

Return type: `null`

Applies to: progress.ui.UIHelper class on page 142

Syntax

```
progress.ui.UIHelper.setItemTemplate( item-template )
```

`item-template`

A `string` value set to an HTML string that specifies new default item formats for list views used by `UIHelper` instances in the current JavaScript session.

It is not necessary to call `setItemTemplate( )` unless one of the following is true:

- You are not using a jQuery Mobile environment.
- You are using a jQuery Mobile environment, but you want to change the default settings provided with the `progress.ui.UIHelper` class (see below).

The `progress.ui.UIHelper` class provides a default list view template for use in all jQuery Mobile environments.

Example

The following code fragment shows the `setItemTemplate( )` method setting the installed default item template values for the `progress.ui.UIHelper` class Notice that the parameter is a single string with the field values for the template. In this particular case, it is a long string:

```
progress.ui.UIHelper.setItemTemplate(
  '<li data-theme="c" data-id="{__id__}">
       <a href="#{__page__}"class="ui-link" data-transition="slide">
          {__format__}</a>
   </li>' );
```

As shown in the preceding example with the class default setting, the template string uses the following substitution parameters:

- **`{__id__}`** — The internal ID of the record
- **`{__page__}`** — The `name` attribute of the object pased as a parameter to `setDetailPage( )`, which defines the form for each individual list item
- **`{__format__}`** — The `format` property of the object pased as a parameter to `setListView( )` or (optionally) to `addItem( )`, which identifies the fields to be included for each item in the list view

See also:

addItem( ) method on page 168, setListView( ) method on page 336

setListView( ) method

Defines a list view for a referenced JSDO table.

Return type: `null`

Applies to: progress.ui.UIHelper class on page 142, table reference property (UIHelper class) on page 353

Syntax

```
uihelper-ref.setListView( object )
uihelper-ref.table-ref.setListView( object )
```

`uihelper-ref`

A reference to a `UIHelper` instance. You can call the method on *`uihelper-ref`* if the JSDO associated with the instance has only a single table reference.

`table-ref`

A table reference on the JSDO associated with the `UIHelper` instance, and that table reference has the records to be displayed in the list view.

`object`

A JavaScript object that contains one or more of the following properties:

- **name** — A `string` set to the `id` attribute of the HTML `div` (or other) element that contains the list view.
- **format** — (Optional) A `string` that specifies the fields shown for each item and the line breaks between them. If the `format` property is omitted, the `UIHelper` instance uses the list item in the HTML ListView as a template.
- **autoLink** — A `boolean` that specifies whether events are automatically generated to display the corresponding form on the detail page when an item is clicked.
- **itemTemplate** — A `string` value set to an HTML string that overrides, for the specified list view, the item format in the default item template or any item format specified by the `setItemTemplate( )` method.

Example

The following code fragment sets the list view page to the HTML element with an `id` attribute value of `'listview'` with a list item formatted to display five specified fields and supported with events generated to display the corresponding detail page when the list item is clicked:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );
uihelper = new progress.ui.UIHelper({ jsdo: dataSet });
uihelper.eCustomer.setListView({
     name:'listview',
     format: '{CustNum}<br>{Name}<br>{Phone}<br>{Address}<br>{State}',
     autoLink: true
});
```

See also:

setItemTemplate( ) method on page 335, showListView( ) method on page 343

setProperties( ) method

Note: Applies to Progress Data Objects Version 4.3 or later.

Replaces all user-defined properties in the current JSDO instance with the user-defined properties defined in the specified object.

Return type: `null`

Applies to: progress.data.JSDO class on page 110

Syntax

```
setProperties ( propsObject )
```

propsObject

An `Object` containing a comma-separated list of *`name`*, *`value`* pairs that define the complete set of user-defined properties in the current JSDO, where *`name`* and *`value`* define a single property as

defined for the `setProperty( )` method. The properties defined in *`propsObject`* entirely replace all other user-defined properties previously defined in the JSDO. You can also remove all existing user-defined properties from the JSDO by specifying `{}` as the value of *`propsObject`*.

Note: Any user-defined property you create with *`name`* set to `"server.count"` has a reserved usage in JSDO plugins. For more information, see the description of the `responseMapping` function that you can define using the addPlugin( ) method on page 171.

Example

The following code fragment sets the `"prop1"` user-defined property in `jsdoCustOrders` to the value `"12345"`. It then calls `setProperties( )` to replace all of the existing user-defined properties in `jsdoCustOrders` with definitions for the two user-defined properties, `"prop2"` and `"prop3"` and writes the current values of `"prop1"`, `"prop2"`, and `"prop3"` to the console log:

```
jsdoCustOrders = new progress.data.JSDO(. . .);

jsdoCustOrders.setProperty("prop1", "12345");
jsdoCustOrders.setProperties({"prop2", 100, "prop3", "ABCD"});
console.log("prop1: " + jsdoCustOrders.getProperty("prop1"));
console.log("prop2: " + jsdoCustOrders.getProperty("prop2"));
console.log("prop3: " + jsdoCustOrders.getProperty("prop3"));
```

The output from the `console.log` statements in this example appear as follows:

```
prop1: undefined
prop2: 100
prop3: ABCD
```

Note that the value returned and written out for `"prop1"` is `undefined`, because a user-defined property with that name is no longer defined in the JSDO.

See also:

getProperties( ) method on page 245, setProperty( ) method on page 338

setProperty( ) method

Note: Applies to Progress Data Objects Version 4.3 or later.

Sets the value of the specified JSDO user-defined property.

Return type: `null`

Applies to: progress.data.JSDO class on page 110

Syntax

```
setProperty ( name [ , value ] )
```

name

The name of a user-defined property to define in the current JSDO.

value

The value, if any, to set for the user-defined *name* property in the JSDO.

This method either defines a new user-defined property or updates an existing user-defined property definition in the JSDO. If *value* is not defined (or set to `undefined`), any existing *name* property definition is removed from the JSDO.

Note: Any user-defined property you create with *name* set to `"server.count"` has a reserved usage in JSDO plugins. For more information, see the description of the `responseMapping` function that you can define using the addPlugin( ) method on page 171.

Example

The following code fragment sets the `"prop1"` and `"prop2"` user-defined properties in `jsdoCustOrders` to the values, `"12345"` and `100`, respectively:

```
jsdoCustOrders = new progress.data.JSDO(. . .);

jsdoCustOrders.setProperty("prop1", "12345");
jsdoCustOrders.setProperty("prop2", 100);
```

See also:

addPlugin( ) method on page 171, getProperty( ) method on page 246, setProperties( ) method on page 337

setSortFields( ) method

Specifies or clears the record fields on which to automatically sort the record objects for a table reference after you have set its `autoSort` property to `true` (the default).

This method enables or disables automatic sorting based on record fields only for supported JSDO operations. See the description of the `autoSort` property for more information.

Return type: `null`

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Syntax

```
jsdo-ref.setSortFields ( sort-fields )
jsdo-ref.table-ref.setSortFields ( sort-fields )
```

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference.

table-ref

A table reference on the JSDO.

sort-fields

An array of `string` values set to the names of record fields on which to sort the record objects, with an optional indication of the sort order for each field. This array can have the following syntax:

Syntax:

```
[ "field-name [ :sort-order ]" [ , "field-name [ :sort-order ]" ] ... ]
```

field-name

The name of a field in the record objects of the specified table reference. Any *field-name* must already exist in the JSDO schema and must have a scalar value (cannot be an array field).

sort-order

An indication of the sort order for the field, which can have one of the following case-insensitive values:

- `ASC` — Sorts ascending.
- `ASCENDING` — Sorts ascending.
- `DESC` — Sorts descending.
- `DESCENDING` — Sorts descending.

The default sort order is ascending.

When the automatic sort occurs, the record objects are sorted and grouped by each successive *field-name* in the array, according to its JavaScript data type and specified *sort-order*. Fields are compared using the >, <, and = JavaScript operators. All `string` fields can be compared with or without case sensitivity depending on the `caseSensitive` property setting. However, note that date fields are compared as dates, even though they are represented as strings in JavaScript.

If you set the *sort-fields* parameter to `null`, or you specify an empty array, the method clears all sort fields. Automatic sorting for the table reference can then occur only if there is an existing sort function setting using the `setSortFn( )` method.

Note: If you set a sort function for the table reference using `setSortFn( )` in addition to using this method to set sort fields, the sort function takes precedence.

Example

In the following code fragment, assuming the `autoSort` property is set to `true` on `dsCustomer.eCustomer` (the default), after the `fill( )` method initializes JSDO memory, the record objects for `eCustomer` are sorted by the `Country` field ascending, then by the `State` field within `Country` ascending, then by the `Balance` field within `State` descending. At a later point, the `foreach( )` method then loops through these record objects, starting with the first record in `eCustomer` sort order:

```
dsCustomer = new progress.data.JSDO( { name: 'dsCustomer' });
dsCustomer.eCustomer.setSortFields( [ "Country", "State", "Balance:DESC" ] );
dsCustomer.fill();
. . .
dsCustomer.eCustomer.foreach( function( customer ){ . . . } );
```

See also:

autoSort property on page 203, caseSensitive property on page 213, setSortFn( ) method on page 341, sort( ) method on page 344

setSortFn( ) method

Specifies or clears a user-defined sort function with which to automatically sort the record objects for a table reference after you have set its `autoSort` property to `true` (the default).

This method enables or disables automatic sorting based on a sort function only for supported JSDO operations. See the description of the `autoSort` property for more information.

Return type: `null`

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Syntax

```
jsdo-ref.setSortFn ( funcRef )
jsdo-ref.table-ref.setSortFn ( funcRef )
```

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference.

table-ref

A table reference on the JSDO.

funcRef

A reference to a JavaScript sort function that compares two record objects for the sort and returns a `number` value. This function must have following signature:

Syntax:

```
function [ func-name ] ( jsrecord-ref1 , jsrecord-ref2 )
```

Where *func-name* is the name of a function that you define external to the `setSortFn( )` method parameter list, and *jsrecord-ref1* and *jsrecord-ref2* are two `JSRecord` objects that the function compares from the specified table reference. You can then pass *func-name* to the `setSortFn( )` method as the *funcRef* parameter. Alternatively, you can specify *funcRef* as the entire inline function definition without *func-name*.

Your *funcRef* code determines the criteria by which one of the two input record objects follows the other in the sort order, and returns one of the following values depending on the result:

- **1** — The *jsrecord-ref1* object follows (is "greater than") the *jsrecord-ref2* object in the sort order.
- **-1** — The *jsrecord-ref1* object precedes (is "less than") the *jsrecord-ref2* object in the sort order.
- **0** — The two record objects occupy the same position (are "equal") in the sort order.

When the JSDO invokes an automatic sort, and a sort function is set using this method, the sort uses this function to determine the sort order for every pair of records that it tests as it iterates through the record objects of the specified table reference.

If you set the *funcRef* parameter to `null`, the method clears any sort function definition. Automatic sorting for the table reference can then occur only if there are one or more existing sort fields set using the `setSortFields( )` method.

Note: Any default JavaScript comparisons that you make with `string` fields in *funcRef* are case sensitive according to JavaScript rules and ignore the setting of the `caseSensitive` property.

Note: If you set sort fields for the table reference using `setSortFields( )` in addition to using this method to set a sort function, the sort function takes precedence.

Examples

In the following code fragment, assuming the `autoSort` property is set to `true` on `dsCustomer.eCustomer` (the default), after the `fill( )` method initializes JSDO memory, the record objects for `eCustomer` are automatically sorted using the results of the external user-defined function, `sortOnNameCSensitive( )`, whose reference is passed to the `setSortFn( )` method. In this case, the function compares the case-sensitive values of the `Name` fields from each pair of `eCustomer` record objects selected by the sort. At a later point, the `foreach( )` method then loops through these record objects, starting with the first record in `eCustomer` sort order:

```
dsCustomer = new progress.data.JSDO( { name: 'dsCustomer' });
dsCustomer.setSortFn ( sortOnNameCSensitive );
dsCustomer.fill();
. . .
dsCustomer.eCustomer.foreach( function( customer ){ . . . } );

function sortOnNameCSensitive ( rec1 , rec2 ) {
  if (rec1.data.Name > rec2.data.Name)
    return 1;
  else if (rec1.data.Name < rec2.data.Name)
    return -1;
  else
    return 0;
  }
```

If you want to compare the `Name` field in this function using a case-insensitive test, you can use the JavaScript `toUpperCase( )` function in the user-defined function. For example, in `sortOnNameCInsensitive( )`, as follows:

```
dsCustomer = new progress.data.JSDO( { name: 'dsCustomer' });
dsCustomer.setSortFn ( sortOnNameCInsensitive );
dsCustomer.fill();
. . .
dsCustomer.eCustomer.foreach( function( customer ){ . . . } );

function sortOnNameCInsensitive ( rec1 , rec2 ) {
  if (rec1.data.Name.toUpperCase() > rec2.data.Name.toUpperCase())
    return 1;
  else if (rec1.data.Name.toUpperCase() < rec2.data.Name.toUpperCase())
    return -1;
  else
    return 0;
  }
```

See also:

autoSort property on page 203, caseSensitive property on page 213, setSortFields( ) method on page 339, sort( ) method on page 344

showListView( ) method

Displays the referenced table's list view on the mobile app device or browser page.

Return type: `null`

Applies to: progress.ui.UIHelper class on page 142, table reference property (UIHelper class) on page 353

Syntax

```
uihelper-ref.showListView( )
uihelper-ref.table-ref.showListView( )
```

uihelper-ref

A reference to a `UIHelper` instance. You can call the method on *uihelper-ref* if the JSDO associated with the instance has only a single table reference.

table-ref

A table reference on the JSDO associated with the `UIHelper` instance, and that table reference has the records to be displayed in the list view.

See also:

clearItems( ) method on page 215, setListView( ) method on page 336

sort( ) method

Sorts the existing record objects for a table reference in JSDO memory using either specified sort fields or a specified user-defined sort function.

Return type: `null`

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Syntax

```
jsdo-ref.sort ( { sort-fields | funcRef } )
jsdo-ref.table-ref.sort ( { sort-fields | funcRef } )
```

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference.

table-ref

A table reference on the JSDO.

sort-fields

An array of `string` values set to the names of record fields on which to sort the record objects, with an optional indication of the sort order for each field. This array can have the following syntax:

Syntax:

```
[ "field-name[:sort-order]" [ , "field-name[:sort-order]" ] ... ]
```

field-name

The name of a field in the record objects of the specified table reference. Any *field-name* must already exist in the JSDO schema and must have a scalar value (cannot be an array field).

sort-order

An indication of the sort order for the field, which can have one of the following case-insensitive values:

- `ASC` — Sorts ascending.
- `ASCENDING` — Sorts ascending.
- `DESC` — Sorts descending.
- `DESCENDING` — Sorts descending.

The default sort order is ascending.

When the sort occurs, the record objects are sorted and grouped by each successive *field-name* in the array, according to its JavaScript data type and specified *sort-order*. Fields are compared using the >, <, and = JavaScript operators. All `string` fields can be compared with or without case sensitivity depending on the `caseSensitive` property setting. However, note that date fields are compared as dates, even though they are represented as strings in JavaScript.

funcRef

A reference to a JavaScript sort function that compares two record objects for the sort and returns a `number` value. This function must have following signature:

Syntax:

```
function [ func-name ] ( jsrecord-ref1 , jsrecord-ref2 )
```

Where *func-name* is the name of a function that you define external to the `sort( )` method parameter list, and *jsrecord-ref1* and *jsrecord-ref2* are two `JSRecord` objects that the function compares from the specified table reference. You can then pass *func-name* to the `sort( )` method as the *funcRef* parameter. Alternatively, you can specify *funcRef* as the entire inline function definition without *func-name*.

Your *funcRef* code determines the criteria by which one of the two input record objects follows the other in the sort order, and returns one of the following values depending on the result:

- `1` — The *jsrecord-ref1* object follows (is "greater than") the *jsrecord-ref2* object in the sort order.
- `-1` — The *jsrecord-ref1* object precedes (is "less than") the *jsrecord-ref2* object in the sort order.
- `0` — The two record objects occupy the same position (are "equal") in the sort order.

When you invoke the `sort( )` method with a sort function, the sort uses this function to determine the sort order for every pair of records that it tests as it iterates through the record objects of the specified table reference.

Note: Any default JavaScript comparisons that you make with `string` fields in *`funcRef`* are case sensitive according to JavaScript rules and ignore the setting of the `caseSensitive` property.

Caution: Because the `sort( )` method executes in JavaScript on the client side, sorting a large set of record objects can take a significant amount of time and make the UI appear to be locked. You might set a wait or progress indicator just prior to invoking the sort to alert the user that the app is working.

Examples

In the following code fragment, the `fill( )` method initializes JSDO memory with `eCustomer` record objects from the server in order of the table primary key (the default). The `sort( )` method later sorts the record objects for `eCustomer` by the `Country` field ascending, then by the `State` field within `Country` ascending, then by the `Balance` field within `State` descending. The `foreach( )` function then loops through these record objects in the new `eCustomer` sort order:

```
dsCustomer = new progress.data.JSDO( { name: 'dsCustomer' });
dsCustomer.fill();
. . .
dsCustomer.sort( [ "Country", "State", "Balance:DESC" ] );
dsCustomer.eCustomer.foreach( function( customer ){ . . . } );
```

In the following code fragment, the `fill( )` method initializes JSDO memory with `eCustomer` record objects from the server in order of the table primary key (the default). The `sort( )` method later sorts the record objects for `eCustomer` using the results of an inline function definition, which in this case compares the case-sensitive values of the `Name` fields from each pair of `eCustomer` record objects selected by the sort. The `foreach( )` method then loops through these record objects in the new `eCustomer` sort order:

```
dsCustomer = new progress.data.JSDO( { name: 'dsCustomer' });
dsCustomer.fill();
. . .
dsCustomer.sort( function( rec1 , rec2 ) {
  if (rec1.data.Name > rec2.data.Name)
    return 1;
  else if (rec1.data.Name < rec2.data.Name)
    return -1;
  else
    return 0;
  } );
dsCustomer.eCustomer.foreach( function( customer ){ . . . } );
```

If you want to compare the `Name` fields using a case-insensitive test, you can use the JavaScript `toUpperCase( )` function in the inline function definition, as follows:

```
dsCustomer = new progress.data.JSDO( { name: 'dsCustomer' });
dsCustomer.fill();
. . .
dsCustomer.sort( function( rec1 , rec2 ) {
  if (rec1.data.Name.toUpperCase() > rec2.data.Name.toUpperCase())
    return 1;
  else if (rec1.data.Name.toUpperCase() < rec2.data.Name.toUpperCase())
    return -1;
  else
    return 0;
  } );
dsCustomer.eCustomer.foreach( function( customer ){ . . . } );
```

See also:

autoSort property on page 203, caseSensitive property on page 213, setSortFields( ) method on page 339, setSortFn( ) method on page 341

subscribe( ) method (JSDO class)

Subscribes a given event callback function to a named event of the current JSDO or table reference.

For more information on these events, see the reference entry for progress.data.JSDO class on page 110.

Return type: `null`

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Syntax

```
jsdo-ref.subscribe ( event-name [ , op-name ] ,
                     event-handler [ , scope ] )
jsdo-ref.table-ref.subscribe ( event-name [ , op-name ] ,
                               event-handler [ , scope ] )
```

jsdo-ref

A reference to the JSDO. If you call the method on *jsdo-ref*, you can subscribe the event handler to any event that fires on the JSDO and all its table references.

table-ref

A table reference on the JSDO. If you call the method on *table-ref*, you can subscribe the event handler to an event that fires only on the table reference.

event-name

A `string` that specifies the name of an event to which you subscribe an event handler. See the reference entry for progress.data.JSDO class on page 110 for a list of available JSDO events.

If you call the `subscribe( )` method on *`table-ref`*, you can subscribe the event handler **only** to the following events:

- `afterCreate`
- `afterDelete`
- `afterUpdate`
- `beforeCreate`
- `beforeDelete`
- `beforeUpdate`

`op-name`

A `string` that specifies the name of a JSDO invocation method, a call to which causes the event to fire. This parameter is required in cases where *`event-name`* is `beforeInvoke` or `afterInvoke`. Use it **only** with these event names, and only when subscribing on the JSDO (not on a *`table-ref`*). The value of `op-name` is the same as the `fnName` property on the request object.

`event-handler`

A reference to an event handler function that is called when the specified event fires.

`scope`

An optional object reference that defines the execution scope of the event handler function called when the event fires. If the `scope` property is omitted, the execution scope is the global object (usually the browser or device window).

See also:

invocation method on page 262, progress.data.JSDO class on page 110, unsubscribe( ) method (JSDO class) on page 353

subscribe( ) method (JSDOSession class)

Subscribes a given event callback function to an event of the current `JSDOSession` object.

This method throws an exception if the specified event is not supported by the `JSDOSession` class.

Return type: `undefined`

Applies to: progress.data.JSDOSession class on page 124

Syntax

```
subscribe ( event-name , event-handler [ , scope ] )
```

event-name

A `string` that specifies the name of an event on a `JSDOSession` object to which you subscribe an event handler. See the reference entry for the progress.data.JSDOSession class on page 124 for a list of available events.

event-handler

A reference to an event handler function that is called when the specified event fires.

scope

An optional object reference that defines the execution scope of the event handler function called when the event fires. If the `scope` property is omitted, the execution scope is the global object (usually the browser or device window).

The `subscribe( )` method throws an error object if *event-name* does not identify an event supported by the `JSDOSession` class (the look up is not case sensitive), or if an argument is not of the correct type.

See also:

unsubscribe( ) method (JSDOSession class) on page 355

subscribe( ) method (Session class)

Note: Deprecated in Progress Data Objects Version 4.4 and later. Please use the subscribe( ) method (JSDOSession class) on page 348 instead.

Subscribes a given event callback function to an event of the current `Session` object.

Return type: `null`

Applies to: progress.data.Session class on page 136

Syntax

```
subscribe ( event-name , event-handler [ , scope ] )
```

event-name

A `string` that specifies the name of an event on a `Session` object to which you subscribe an event handler. See the reference entry for the progress.data.Session class on page 136 for a list of available events.

event-handler

A reference to an event handler function that is called when the specified event fires.

scope

An optional object reference that defines the execution scope of the event handler function called when the event fires. If the `scope` property is omitted, the execution scope is the global object (usually the browser or device window).

The `subscribe( )` method throws an error object if *`event-name`* does not identify an event supported by the `Session` object (the lookup is case insensitive), or if an argument is not of the correct type.

See also:

unsubscribe( ) method (Session class) on page 356

success property

A `boolean` that when set to `true` indicates that the Data Object resource operation was successfully executed.

This property is set from the HTTP status code returned from the server. A successful resource operation returns an HTTP status code in the range of 200 - 299, with one exception. An unsuccessful resource operation causes a value outside this range to be returned for the HTTP status code and sets this property to `false`. In addition, a record-change operation (Create, Update, Delete, or Submit) on before-image data that returns an HTTP status code of 200 OK, and returns a record object with a before-image error message is **also** unsuccessful, and sets this property to `false`.

Data type: `boolean`

Access: Read-only

Applies to: request object on page 146

The `success` property is available only for the following JSDO events or in the request object returned to a jQuery Promise callback:

- `afterCreate`
- `afterDelete`
- `afterFill`
- `afterInvoke`
- `afterSaveChanges`
- `afterUpdate`

In the case of an `afterSaveChanges` event, the `success` property is `true` only if **all** record-change operations that are invoked by the `saveChanges( )` method were successfully executed.

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe or register to JSDO events, any returned jQuery Promises, and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO. This object is also returned as the value of any JSDO invocation method that you execute synchronously.

Note: For an OpenEdge resource, when the server routine that implements a resource operation raises an unhandled error, this causes an HTTP status code of 500 and any included error information to be returned from the server. This can occur when either an ABL routine raises an application error or the ABL virtual machine (AVM) raises a system error, and the error is thrown out of the top-level server routine.

See also:

fill( ) method on page 220, invocation method on page 262, invoke( ) method on page 264, saveChanges( ) method on page 314

table reference property (JSDO class)

An object reference property on a JSDO that has the name of a corresponding table in the Data Object resource for which the current JSDO was created.

Its value is a reference (*table reference*) to the table object in JSDO memory. This table object provides access to a working record, if defined. If the server Data Object provides access to a multi-table resource, such as an OpenEdge ProDataSet, the JSDO provides one table reference for every table in the multi-table resource.

Data type: Table object reference in JSDO memory

Access: Read-only

Applies to: progress.data.JSDO class on page 110

In JSDO syntax descriptions, wherever a table reference can be used, for example, in *jsdo-ref*.*table-ref*, where *jsdo-ref* represents a reference to a JSDO instance, *table-ref* represents the name of a property containing the table reference. Every referenced table object, in turn, provides the following properties:

- **`record` property** — A reference to a `JSRecord` object (*record object*), which provides the data for the working record of the table in its own `data` property. This `data` property provides access to the field values of the working record as corresponding field reference properties (see the following bullet). If no working record is defined for the table object, its `record` property is `null`.
- **field reference property** — Also referred to as a *field reference*, a property on the table object that has the name and data type of a field (as defined in the table schema) and the value for that field in the working record. In JSDO syntax descriptions, wherever a field reference can appear, for example, in *jsdo-ref*.*table-ref*.*field-ref*, *field-ref* represents a property with the name of a corresponding table field containing the field reference.

  A JSDO table object provides one such field reference for each field defined in the corresponding resource table. If no working record is defined for a given table object, all of its field references are `null`. However, you can reference the current value for any *field-ref* on the `data` property of any `JSRecord` object reference that you have previously saved for a working record or that you otherwise return during a search of JSDO memory, for example, using the JSDO `foreach( )` method.

  The JSDO also provides support for accessing array fields, which can be defined in an OpenEdge resource temp-table. You can reference individual array elements of any *field-ref* defined as an array using any of the following mechanisms:

  - **Standard JavaScript subscript reference** — Such as an integer subscript reference, *field-ref*`[`*index*`]`, where *index* is a zero (0)-based integer expression that indexes the array.
  - **JSDO array-element reference** — Which specifies each element of the array as a separate field reference property with a unique name specified according to the format, *field-ref_integer*, where *field-ref* is the name of the original array field, *integer* is a one (1)-based integer that qualifies the field name, and where *field-ref* and *integer* are separated by a single underscore (_). Each

field-ref_integer property contains the same value as the corresponding *field-ref* array element, *field-ref*[*index*], where *index* is equal to *integer* - 1. (**Note:** The *integer* name qualifier is one-based to match OpenEdge array field subscripts, which are one-based in OpenEdgeAdvanced Business Language (ABL).)

Note: A given JSDO array-element reference can conflict with a scalar field that happens to have the same *field-ref_integer* name. If this happens for an OpenEdge resource table, you can resolve the conflict by assigning a different scalar field name to be generated in the Catalog for the corresponding ABL temp-table field definition. To assign a different generated field name, specify the SERIALIZE-NAME option for the affected scalar field in the temp-table definition. For more information, see the description of the DEFINE TEMP-TABLE statement in *OpenEdge Development: ABL Reference*.

You can therefore read field values in the working record of a given table reference using corresponding field references **either** on the data property of the JSRecord object returned by the record property of the table reference or directly on the table reference itself. However, although you can assign values directly to fields referenced on this data property, doing so does not trigger any automatic sorting of the data in JSDO memory, or any saving of the updated field value to the Data Object resource on the server. Instead, to write field values, Progress **strongly recommends** that you **either** call the JSDO assign( ) method on a *jsdo-ref*.*table-ref* with a working record (or on **any** valid JSRecord object reference), **or** assign values directly to any *jsdo-ref*.*table-ref*.*field-ref* with a working record.

Caution: Again, **never write directly** (by assigning a value) to a *field-ref* on the record.data property (or on the data property of any JSRecord object reference); use *field-ref* on this data property **only to read** the current field value. Writing field values directly on data.*field-ref* of a JSRecord object reference does **not** mark the record for update when you subsequently call the saveChanges( ) method, nor does it re-sort the updated record in JSDO memory according to any order you have established using the autoSort property. To mark a record for update and automatically re-sort the record according to the autoSort property, you **must** set a field value **either** by calling the assign( ) method on a record object **or** by assigning the value directly to the *jsdo-ref*.*table-ref*.*field-ref* of a working record, as described above.

Note: The record property with a working record does provide an alternative way to read a table field that happens to have the same name as any JSDO property (or method) that you can access (or invoke) directly on a table reference with a working record (**except**, of course, a table field with the name, record). That is, not every JSDO property (or method) that is available on a table reference is also available on the record.data property itself.

Examples

For example, the following code fragment shows two different ways to read the CustNum field of a record added to a Customer table provided by a Data Object resource ('CustomerOrderDS'):

```
var dataSet = new Progress.data.JSDO( 'CustomerOrderDS' );
dataSet.Customer.add();
alert(dataSet.Customer.record.data.CustNum);
alert(dataSet.Customer.CustNum);
```

Both calls to the alert( ) function access the same CustNum field in the working record of the Customer table created using the add( ) method.

In the following code fragment, `MonthQuota` is an array field with 12 elements (one for each month of the year) in the `Salesrep` table, and shows two different ways to read an element of the `MonthQuota` array from a record added to a `Salesrep` table provided by an OpenEdge Data Object resource (`SalesDS`):

```
var dataSet = new Progress.data.JSDO( 'SalesDS' );
var idx = 0;
dataSet.Salesrep.add();
alert(dataSet.Salesrep.record.data.MonthQuota[idx]);
alert(dataSet.Salesrep.MonthQuota_1);
```

Both calls to the `alert( )` function access the same `MonthQuota` array element in the working record of the `Salesrep` table created using the `add( )` method.

Note: Both of the previous examples assume the names of tables and fields in the OpenEdge sports2000 sample database. However, note also that where `MonthQuota` is presented as an array field for this example, that same field in the OpenEdge-installed instance of sports2000 is defined instead as a scalar integer field.

For more information on accessing the working record of a table reference, see the notes in the section on progress.data.JSDO class on page 110.

See also:

autoSort property on page 203, data property on page 217, progress.data.JSRecord class on page 133, record property on page 305

table reference property (UIHelper class)

An object reference property on a `UIHelper` instance that corresponds to a table of the JSDO with which the `UIHelper` instance is associated.

You can use this table reference to access methods of the `progress.ui.UIHelper` class.

Data type: Table object reference in JSDO memory

Access: Read-only

Applies to: progress.ui.UIHelper class on page 142

Note: A table reference (*`table-ref`* in syntax) on a `UIHelper` object does not have a `record` property. Therefore you cannot use this table reference to directly access record field values, but only to invoke methods on the `UIHelper` object.

See also:

table reference property (JSDO class) on page 351

unsubscribe( ) method (JSDO class)

Unsubscribes a given event callback function from a named event of the current JSDO or table reference.

For more information on these events, see the refernce entry for progress.data.JSDO class on page 110.

Return type: null

Applies to: progress.data.JSDO class on page 110, table reference property (JSDO class) on page 351

Syntax

```
jsdo-ref.unsubscribe ( event-name [ , op-name ] ,
                       event-handler [ , scope ] )
jsdo-ref.table-ref.unsubscribe ( event-name [ , op-name ] ,
                                 event-handler [ , scope ] )
```

jsdo-ref

A reference to the JSDO. If you call the method on *jsdo-ref*, you can unsubscribe the event handler from any event that fires on the JSDO and all its table references.

table-ref

A table reference on the JSDO. If you call the method on *table-ref*, you can unsubscribe the event handler from an event that fires only on the table reference.

event-name

The name of an event to which you unsubscribe an event handler. See the refernce entry for progress.data.JSDO class on page 110 for a list of available JSDO events.

If you call the unsubscribe( ) method on *table-ref*, you can unsubscribe the event handler **only** from the following events:

- afterCreate
- afterDelete
- afterUpdate
- beforeCreate
- beforeDelete
- beforeUpdate

op-name

The name of a JSDO invocation method, a call to which causes the event to fire. This parameter is required in cases where *event-name* is beforeInvoke or afterInvoke. Use it **only** with these event names, and only when unsubscribing on the JSDO (not on a *table-ref*). To work, the *op-name* value must match the corresponding *op-name* parameter in a preceding subscribe( ) method call.

event-handler

A reference to an event handler function that is called when the specified event fires.

scope

An optional object reference that defines the execution scope of the event handler function called when the event fires. Specifying the scope is optional in the event subscription. If the event subscription **does** specify an execution scope, you must specify a matching *scope* parameter when you call the `unsubscribe( )` method to cancel the event subscription.

See also:

invocation method on page 262, progress.data.JSDO class on page 110, subscribe( ) method (JSDO class) on page 347, unsubscribeAll( ) method on page 356

unsubscribe( ) method (JSDOSession class)

Unsubscribes a given event callback function from an event of the current `JSDOSession` object.

This method throws an exception if the specified event is not supported by the `JSDOSession` class.

Return type: `undefined`

Applies to: progress.data.JSDOSession class on page 124

Syntax

```
unsubscribe ( event-name , event-handler [ , scope ] )
```

event-name

The name of a `JSDOSession` object event to which you unsubscribe an event handler. See the reference entry for the progress.data.JSDOSession class on page 124 for a list of available events.

event-handler

A reference to an event handler function that is to be removed from the list of callbacks that are called when the specified event fires.

scope

An optional object reference that defines the execution scope of the event handler function. Specifying the scope is optional in the event subscription. If the event subscription **does** specify an execution scope, you must specify a matching *scope* parameter when you call the `unsubscribe( )` method to cancel the event subscription.

The `unsubscribe( )` method throws an error object if *event-name* does not identify an event supported by the `JSDOSession` class (the look up is not case sensitive), if an argument is not of the correct type, or if there is no handler that was subscribed for that event with the same *scope* as passed to `unsubscribe( )`.

See also:

progress.data.JSDOSession class on page 124, subscribe( ) method (JSDOSession class) on page 348

unsubscribe( ) method (Session class)

Note: Deprecated in Progress Data Objects Version 4.4 and later. Please use the unsubscribe( ) method (JSDOSession class) on page 355 instead.

Unsubscribes a given event callback function from an event of the current `Session` object.

Return type: `null`

Applies to: progress.data.Session class on page 136

Syntax

```
unsubscribe ( event-name , event-handler [ , scope ] )
```

event-name

The name of a `Session` object event to which you unsubscribe an event handler. See the refernce entry for progress.data.Session class on page 136 for a list of available events.

event-handler

A reference to an event handler function that is called when the specified event fires.

scope

An optional object reference that defines the execution scope of the event handler function called when the event fires. Specifying the scope is optional in the event subscription. If the event subscription **does** specify an execution scope, you must specify a matching *scope* parameter when you call the `unsubscribe( )` method to cancel the event subscription.

The `unsubscribe( )` method throws an error object if *event-name* does not identify an event supported by the `Session` object (the lookup is case insensitive), if an argument is not of the correct type, or if there is a mismatch between the value of the *scope* argument and the scope specified in the corresponding `subscribe( )` method call.

See also:

progress.data.Session class on page 136, subscribe( ) method (Session class) on page 349, unsubscribeAll( ) method on page 356

unsubscribeAll( ) method

Unsubscribes all event callback functions from a single named event of the current JSDO, `JSDOSession` or `Session` object, or unsubscribes all event callback functions from all events of the current JSDO, `JSDOSession`, or `Session` object.

Return type: `null`

Applies to: progress.data.JSDO class on page 110, progress.data.JSDOSession class on page 124, progress.data.Session class on page 136

Syntax

```
unsubscribeAll ( [ event-name ] )
```

event-name

A `string` that if specified, is the name of an event on the current object from which to unsubscribe all event handlers. If not specified, the method unsubscribes all event handlers from all events of the current object. See the reference entry for the progress.data.JSDO class on page 110, progress.data.JSDOSession class on page 124 or the progress.data.Session class on page 136 for a list of available events.

For a `JSDOSession` or `Session` object, the `unsubscribeAll( )` method throws an error object if *event-name* does not identify an event supported by the `progress.data.JSDOSession` or `progress.data.Session` class (the lookup is case insensitive), or if *event-name* is not a `string`. For a JSDO instance, the method ignores these conditions.

See also:

unsubscribe( ) method (JSDO class) on page 353, unsubscribe( ) method (JSDOSession class) on page 355, unsubscribe( ) method (Session class) on page 356

useRelationships property

A `boolean` that specifies whether JSDO methods that operate on table references in JSDO memory recognize and honor *data-relations* defined in the schema (that is, work only on records of a child table that are related to the working record of a parent table).

Data type: `boolean`

Access: Readable/Writable

Applies to: progress.data.JSDO class on page 110

When set to `true`, methods, such as `add( )`, `find( )`, and `foreach( )`, that have default behavior for related table references respect these relationships when operating on related tables. When set to `false`, these methods operate on all table references as if they have no relationships. The default value is `true`.

Note: Data-relations defined for OpenEdge Data Object resources (ProDataSets) that use the `RECURSIVE` key word have no effect and are not supported by the JSDO.

See also:

add( ) method on page 158, find( ) method on page 227, foreach( ) method on page 231

userName property

Returns the username passed as a parameter to the most recent call to the `login( )` method on the current `JSDOSession` or `Session` object.

Data type: `string`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 124, progress.data.Session class on page 136

This value is returned, whether or not the most recent call to `login( )` succeeded.

See also:

getSession( ) stand-alone function on page 248, login( ) method (JSDOSession class) on page 275, login( ) method (Session class) on page 280

xhr property

A reference to the XMLHttpRequest object used to make an operation request on a resource of a Data Object Service.

In the case of an asynchronous call, this property may not be available until after the XMLHttpRequest object is created.

Note: You can use the XMLHttpRequest object referenced by this property to return error information that originates from the web server that hosts the Data Object Service handling the operation request. As of Progress Data Objects 4.3 and later, the JSDO provides a `getErrors( )` method that returns all error information for an operation request, including error information that originates from the hosting web server. For more information, see the getErrors( ) method on page 233 description.

Data type: `Object`

Access: Read-only

Applies to: request object on page 146

The `xhr` property is available for the following events or in the request object returned to a jQuery Promise callback:

- `afterFill`
- `afterInvoke`

The `xhr` property is available for the following events after calling the JSDO `saveChanges( )` method either with an empty parameter list or with the single parameter value of `false`:

- `afterCreate`
- `afterDelete`
- `afterUpdate`

The `xhr` property is available **only** for the following event or in the request object returned to a jQuery Promise callback after calling `saveChanges(true)` on a JSDO resource that supports a Data Object Submit operation:

- `afterSaveChanges`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe or register to JSDO events, any returned jQuery Promises, and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO. This object is also returned as the value of any JSDO invocation method that you execute synchronously.

See also:

fill( ) method on page 220, getErrors( ) method on page 233, invocation method on page 262, invoke( ) method on page 264, response property on page 312, saveChanges( ) method on page 314

8

Data type mappings for Data Object Services

Each type of Data Object Service maps the data types used and understood by its resources to JSON types for transport over the network, and the data are then converted to corresponding JavaScript data types when loaded into JSDO memory. A similar process happens in reverse when the JSDO sends data back over the network to the Data Object Service. The following topics provide an overview of JavaScript data type support in the JSDO and the supported mappings to the data types of OpenEdge ABL and Rollbase object resources.

For details, see the following topics:

- JavaScript data type overview
- OpenEdge ABL to JavaScript data type mappings
- Rollbase object to JavaScript data type mappings

JavaScript data type overview

JavaScript supports five primitive data types, as shown in the following table.

Table 38: JavaScript primitive data types

Data type	Description	Examples
`string`	A string of characters enclosed in double or single quotes.	`"jump rope"`

`number`	An unquoted numeric value, which can include an exponent using scientific notation.	`17`
		`54.35`
		`0.9582e-42`

`boolean`	The unquoted, lowercase, literal value `true` or `false`.	`true`
`null`	The unquoted, lowercase, literal value, `null`.	`null`
`undefined`	The unquoted, lowercase, literal value, `undefined`.	`undefined`

The data type of a primitive value is determined by the format of the value:

- A string of characters surrounded by quotes indicates the value is a `string`.
- A string of numeric characters without surrounding quotes and with an optional decimal point, an optional negative sign, or an optional exponent indicates a `number`.
- The string `true` or `false` without surrounding quotes indicates a `boolean`.
- The string `null` without surrounding quotes indicates a literal `null` value that typically represents a variable, object property, or function return value that is not assigned to any other data type value.
- The string `undefined` without surrounding quotes also indicates a literal `undefined` value that typically represents a variable, object property, or function return value that is undefined.

In addition to these standard primitive data types, there are some commonly-used, but non-standard data types for certain values that are not officially supported in JavaScript. For these non-standard types, specially formatted strings represent values for which there is no standard primitive JavaScript data type.

The following table shows non-standard JavaScript data types that OpenEdge Data Object resources support.

Table 39: OpenEdge-supported non-standard JavaScript data types

Non-standard JavaScript data type	Representation
Date	A `string` in the ISO 8601 format, `"yyyy-mm-ddThh:mm:ss.sss+hh:mm"`. JavaScript does support a `Date` object for working with dates and times. However, all dates and times returned from an OpenEdge application server to a JSDO are stored as a `string` in the ISO 8601 format.
Binary data	A `string` consisting of the Base64 encoded equivalent of the binary data, which in OpenEdge can be represented in an ABL `BLOB`, `MEMPTR`, or `ROWID` data type.

For more information on the OpenEdge usage of the non-standard JavaScript data types in the previous table, see OpenEdge ABL to JavaScript data type mappings on page 363.

JavaScript also supports two complex data types, used to aggregate values of all JavaScript data types, including both primitive and complex data, as shown in the following table:

Table 40: JavaScript complex data types

Data type	Description	Examples

`Object`	A comma-delimited list of named values (*properties*), either primitive or complex, enclosed in braces (`{}`). The property names can either be literal values or quoted strings.	`{ myString : "jump rope",`
		`  'myNum' : 17,`
		`  'myBool' : false }`

`Array`	A comma-delimited list of unnamed values, either primitive or complex, enclosed in brackets (`[]`)	`[ "jump rope", 17, false ]`

Note: JavaScript also supports standard objects with the same type names as the primitive data types written by convention using initial upper case, for example, a `Number` object. These objects serve as wrappers for the corresponding primitive types and provide additional operations on these primitive types.

OpenEdge ABL to JavaScript data type mappings

A JSDO communicates with an OpenEdge application server through a Data Object Service to marshal ABL data for parameters (and return values) of ABL routines that pass the following ABL data types: ProDataSets, temp-tables, arrays, and primitive values. The JSDO provides this ABL data to the mobile app programmer as JavaScript data. For more information on ABL data types, see the reference entry on data types in *OpenEdge Development: ABL Reference*. For more information on the format and use of JavaScript data types, you can review sources on the web, such as http://www.w3schools.com/js/js_datatypes.asp. For more information on JSDOs and how they communicate with an OpenEdge application server through a Data Object Service, see progress.data.JSDO class on page 110.

The following table shows the mappings between the supported ABL data types in an OpenEdge Data Object resource and the JavaScript data types of corresponding table fields and invocation method parameter values in a JSDO. These data type conversions happen in the server-side Data Object when the JSDO invokes a resource operation that exchanges data between the JSDO and its OpenEdge resource using the `fill( )`, `saveChanges( )`, or any invocation method.

Note: An ABL data item or Invoke routine output parameter that is set to the Unknown value (`?`) maps to the JavaScript `null` value when sent from the OpenEdge Data Object resource to the JSDO, and a JSDO data item or invocation method input parameter value that is set to `null` maps to the ABL Unknown value (`?`) when sent from the JSDO to the OpenEdge Data Object resource. The ABL `BLOB` and `CLOB` are only allowed as fields of temp-table parameters, and their respective equivalents, `MEMPTR` and `LONGCHAR`, are only allowed as scalar parameters or as elements of `Array` parameters.

Table 41: OpenEdge ABL data type mappings

ABL data type	JavaScript data type
`BLOB`[1]	`string` (Base64 encoded)
`CHARACTER`	`string`
`CLOB`[1]	`string`
`COM-HANDLE`	`number`
`DATASET`[2]	An `Object` that maps to a ProDataSet and contains one or more `Object` instances, each of which maps to an ABL temp-table in the ProDataSet (see `TEMP-TABLE`)
`DATE`[3]	`string` (ISO 8601 formatted string of the form `"yyyy-mm-dd"`)
`DATETIME`[3]	`string` (ISO 8601 formatted string of the form `"yyyy-mm-ddThh:mm:ss.sss"`)
`DATETIME-TZ`[3]	`string` (ISO 8601 formatted string of the form `"yyyy-mm-ddThh:mm:ss.sss+hh:mm"`)
`DECIMAL`	`number`
`primitive` `EXTENT`[2]	Where *`primitive`* is an ABL primitive data type (not a `DATASET` or `TEMP-TABLE`), maps to an `Array` of the JavaScript primitive data type that maps to the corresponding ABL primitive data type
`HANDLE`	`number`
`INT64`	`number`
`INTEGER`	`number`
`LOGICAL`	`boolean` (`true` or `false`)
`LONGCHAR`[4]	`string`
`MEMPTR`[4]	`string` (Base64 encoded)
`RAW`	Not supported
`RECID`	Not supported

[1] In temp-tables only.

[2] ABL ProDataSets, temp-tables, and arrays map to JavaScript objects and arrays using a structure that is identical to the OpenEdge-supported mapping to JSON objects and arrays. JSON (JavaScript Object Notation) is a character representation of JavaScript data types that is often used as a lightweight alternative to XML. For more information on ABL support for JSON, including the JSON representation of these ABL data types, see *OpenEdge Development: Working with JSON*.

[3] See Date and time conversions on page 365.

[4] As scalar parameters or elements of Array parameters only.

ABL data type	JavaScript data type
ROWID	string (Base64 encoded)
TEMP-TABLE[2]	An Object that contains a single Array of Object instances, where each Object in the Array maps to a record in the corresponding temp-table

Date and time conversions

In the JSDO, all ABL DATE, DATETIME, and DATETIME-TZ data types are mapped to a string in the ISO 8601 format shown in the previous table. This format supports all the necessary functionality for these data types (date, time, and time zone data). However, if you read and convert (**Output**) this string from the JSDO to a JavaScript Date object, or write (**Input**) the value of a JavaScript Date object to the JSDO as a similarly ISO 8601-formatted string, the JSDO processes the value differently based on the original ABL data type in the OpenEdge resource, as shown in the following table:

Table 42: JavaScript Date object conversions to and from ISO 8601 strings

ABL data type	Process
DATE	**Output:** Client time zone is used. Time is set to 0 (midnight). **Input:** Time and time zone information is ignored.
DATETIME	**Output:** Client time zone is used. **Input:** Time zone information is ignored
DATETIME-TZ	**Output and Input:** Date, time, and time zone is preserved.

Note: This same processing occurs whether for reading and writing data on the OpenEdge server as part of Data Object Read (**Output**), Create (**Input**), and Update (**Input**) operations, or for reading and writing data on the server through the parameters (**Output** and **Input**) and return values (**Output**) of Invoke operations.

Note: This date and time conversion supports Kendo UI clients that access the JSDO as a remote data service using the JSDO dialect of the Kendo UI DataSource, which works with JSDO date and time data as JavaScript Date objects. For more information, see Using the JSDO dialect of the Kendo UI DataSource on page 17.

Rollbase object to JavaScript data type mappings

A JSDO communicates with a a Rollbase server through a Data Object Service to marshal Rollbase field data for parameters (and return values) of the Rollbase APIs that operate on Rollbase objects. The JSDO provides this Rollbase field data to the mobile app programmer as JavaScript data. For more information on Rollbase field types, see the Rollbase reference to Field Types. For more information on the format and use of JavaScript data types, you can review sources on the web, such as http://www.w3schools.com/js/js_datatypes.asp. For more information on JSDOs and how they communicate with a Rollbase server through a Data Object Service, see progress.data.JSDO class on page 110.

The following tables describe the mappings between the field types of different categories of Rollbase object fields and the JavaScript data types of corresponding table fields in a JSDO. These data type conversions happen in the server-side Data Object when the JSDO invokes a resource operation that exchanges data between the JSDO and its Rollbase resource using the `fill( )`, `saveChanges( )`, or any invocation method.

Note: Any unmapped Rollbase fields default to a JavaScript `string`.

Table 43: Rollbase Basic field type mappings

Rollbase field type	JavaScript data type
Base Currency	`number`
Checkbox	`boolean`
Currency	`number`
Date	`string`
Date/Time	`string`
Decimal	`number`
Email	`string`
Group of Checkboxes	`string`
Integer	`number`
Password	`string`
Percent	`number`
Phone Number	`string`
Picklist	`string`
Picklist(Multi-Select)	`string`
Radio Buttons	`number`
Reference	`number`
Text	`string`
Text Area	`string`
Time	`number`
URL	`string`

Table 44: Rollbase Advanced field type mappings

Rollbase field type	JavaScript data type
Auto-number	`string`
Dependent Picklist	`number`
Document Template	`string`
Email Template	`string`
Expression	`string`
File Upload	`string`
Formula	`string`
Image Upload	`string`
Integration Link	`string`
Related Field	`string`
Roll-up Summary	`string`
Shared Image	`string`
Template	`string`
Version Number	`number`

Table 45: Rollbase Portal field type mappings

Rollbase field type	JavaScript data type
Captcha Image	`string`
Hidden Input	`string`
IP Address	`string`

Table 46: Rollbase System field type mappings (read-only)

Rollbase field type	JavaScript data type
Comments	`string`
iCal	`string`
LDF Filter	`string`

Rollbase field type	JavaScript data type
Organization Data	`string`
Parent Object	`string`
Tag	`string`
Time Zone	`string`
User Role	`string`
vCard	`string`