Progress®



Develop Progress Data Objects

OpenEdge®

Copyright

© 2021 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.

DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Defrag This, Deliver More Than Expected, DevReach (and design), Icenium, Inspec, Ipswitch, iMacros, Kendo UI, Kinvey, MessageWay, MOVEit, NativeChat, NativeScript, OpenEdge, Powered by Chef, Powered by Progress, Progress, Progress Software Developers Network, SequeLink, Sitefinity (and Design), Sitefinity, Sitefinity (and design), SpeedScript, Stylus Studio, Stylized Design (Arrow/3D Box logo), Styleized Design (C Chef logo), Stylized Design of Samurai, TeamPulse, Telerik, Telerik (and design), Test Studio, WebSpeed, WhatsConfigured, WhatsConnected, WhatsUp, and WS_FTP 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, Chef Automate, Chef Compliance, Chef Desktop, Chef Habitat, Chef WorkStation, Corticon.js, Corticon Rules, Data Access, DataDirect Autonomous REST Connector, DataDirect Spy, DevCraft, Fiddler, Fiddler Everywhere, FiddlerCap, FiddlerCore, FiddlerScript, Hybrid Data Pipeline, iMail, JustAssembly, JustDecompile, JustMock, KendoReact, NativeScript Sidekick, OpenAccess, PASOE, Pro2, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, Push Jobs, SafeSpaceVR, Sitefinity Cloud, Sitefinity CMS, Sitefinity Digital Experience Cloud, Sitefinity Feather, Sitefinity Insight, Sitefinity Thunder, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, Supermarket, SupportLink, Unite UX, 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.

July 2021

Product version: Progress OpenEdge 12.4

Updated: 2021/06/28

Table of Contents

Preface....7

Overview of Progress Data Objects, Services, and Catalogs....9

Run-time architecture and data access....10
Additional development options....11

Using the JSDO dialect of the Kendo UI DataSource 15

Features of the JSDO dialect....16
Sample Kendo UI DataSource instantiations using the JSDO dialect....19
Sample instantiation for an OpenEdge Data Object Service....19

Use Progress Data Source for NativeScript and Angular....25

Use JSDO and the Data Source Modules in an existing NativeScript application....31

Use JSDO to create mobile and web clients....33

JSDO overview....34
JSDO classes and objects....34
How a JSDO maps to a Data Object resource....35
How JSDO memory works....36
Methods of the JSDO and JSRecord classes....39
Asynchronous execution....43
Properties of JSDO....49
Requirements to use JSDO....51
Enable cookies....52
Create and manage access to a JSDO instance....52
CRUD and Submit operations....54
CRUD examples....57
Server-side implementation....67
Access custom Invoke operations....69
Asynchronous vs. synchronous method execution....69
Invoke operation example....70
Manage JSDO login sessions....75
Requirements for creating a JSDO login session....75
Use default web pages to support client app login....77
Handle changes in session online status....78

Use protected web resources....80
JSDOSession Management Model....81

Preface

Purpose

This manual describes Progress Data Objects concepts, procedures, and utilities that enable you to create and employ Progress Data Objects.

Audience

This manual is designed as a guide for developers working with Progress Data Objects.

Documentation conventions

See Documentation Conventions for an explanation of the terminology, format, and typographical conventions used throughout the OpenEdge content library.

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 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/.

Progress Data Objects supports two basic app development options: developing mobile apps using the remot http client access and developing both mobile and web apps using other development tools.

For details, see the following topics:

- Run-time architecture and data access
- Additional 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.

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** — 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** — 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** — Represents a single record of data associated with a Data Object resource that is stored in the memory of the JSDO instance.
- **progress.ui.UIHelper class** — Allows you to dynamically bind JSDO data to a JavaScript UI using certain supported UI frameworks.
- **request object** — 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 Progress Data Object Services include those supported by an OpenEdge application server. OpenEdge application servers are 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.

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 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 Use JSDO to create mobile and web clients on page 33.

Additional development options

The following options can be used to support and enhance your applications and data objects:

OpenEdge Data Object Services

Before you can build and test client apps to access Progress Data Objects using JSDO, you must obtain access to 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*.

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 the Product Overview and Bundle Support for Kendo UI Components.

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. You can 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 15.

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 Use JSDO to create mobile and web clients on page 33

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.

To learn 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.

Other options

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 Use JSDO to create mobile and web clients on page 33.

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: 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.

- **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`**, 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 19 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`**, 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) 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.

  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 Data Object Services. Specific configuration features of the JSDO dialect are highlighted in bold.

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.

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. For more information, see the description of the JSDO fill( ) method.

  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`. 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`, the respective feature is ignored by both the server and the JSDO dialect of the Kendo UI DataSource.

- **Set 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"`.
- **Set 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.
- **Set 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 remot http client access in *OpenEdge Development: Web Services*. 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: 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.

- **Set 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 16.
- **Set 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). 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	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, 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.

- **Handle the parameter (e) and associated JSDO errors returned by the DataSource `error` event handler function** — 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 19, 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 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.

3

Use Progress Data Source for NativeScript and Angular

Description

The Progress Data Source is an object that connects an Angular application with a Progress Data Object Service. It allows you to perform CRUD operations on a Data Object Service in TypeScript. It is available as an npm package on http://www.npmjs.com.

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 4: 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 demonstrates how to instantiate a DataSource in TypeScript:

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

let jsdo:progress.data.JSDO,
    session:progress.data.JSDOSession,
    dataSource: DataSource;

progress.data.getSession({
    serviceURI: uri,
    catalogURI: catUri,
    authenticationModel: "anonymous"
}).then((object) => {
    session = object.jsdosession;
    jsdo = new progress.data.JSDO({
        name: "Customer"
    });

    dataSource = new DataSource({
        jsdo: jsdo,
        tableRef: "ttCustomer"
    });
});
```

The following example demonstrates how to read data:

```
// This is a read with a filter, the filter is not required
dataSource.read({
    filter: {
        field: 'State',
        operator: 'eq',
        value: 'HI'
    }
}).subscribe((myData: DataResult) => {
    // You can also loop through myData.data instead of calling dataSource.getData()
    dataSource.getData().forEach(element => {
        // print out each customers CustNum
        console.log(element.CustNum);
    })
});
```

The following example demonstrates how to create a new record:

```
dataSource.create({
    CustNum: 100000,
    Name: 'testName',
    State: 'MA',
    Balance: 0
});
// You have to call saveChanges for the changes to propagate to the backend
dataSource.saveChanges().subscribe((myData) => {
    // process my data here
    myData.data.forEach((elem) => console.log(elem));
}, (error: Error ) => {
    // process error here
});
```

The following example demonstrates how to update a record:

```
dataSource.getData().forEach(element => {
    if (element.CustNum === 3000) {
        element.Name = "TestUser Foo"
        dataSource.update(element);
    }
});
// You have to call saveChanges for the changes to propagate to the backend
dataSource.saveChanges().subscribe((myData) => {
    // process data here
    myData.data.forEach((elem) => console.log(elem));
}, (error: Error ) => {
    // process error here
});
```

The following example demonstrates how to remove a record:

```
dataSource.getData().forEach(element => {
    if (element.CustNum === 9000) {
        dataSource.remove(element);
    }
});
// You have to call saveChanges for the changes to propagate to the backend
dataSource.saveChanges().subscribe((myData) => {
    // process data here
}, (error: Error ) => {
    // process error here
});
```

Note: `saveChanges()` only needs to be called when you are finished with your local changes. You do not need to call it for each individual operation.

4

Use JSDO and the Data Source Modules in an existing NativeScript application

Setup:

This setup uses the NativeScript Cars app to demonstrate this functionality.

1. Run the following command to create the master template: `tns create my-master-detail-ng --template tns-template-master-detail-ng`.
2. Use the following command to access the master template: `cd my-master-detail-ng`.
3. Use the following command to set up your desired operating system: `tns run <android/ios>`.
4. Install the `@progress/jsdo-nativescript` and `@progress/jsdo-core` npm packages in your NativeScript project by running the following command: `npm install @progress/jsdo-nativescript @progress/jsdo-core`.
5. Make the following changes to the Cars app to use the JSDO and the Data Source modules:

   a. In `src/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";
     ```

   - Find 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}`);
        }).pipe(catchError(this.handleErrors));
}
```

- Replace the above code with the following:

```
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));
}
```

b. In `src/app/cars/car-list.component.html`, change `car.name` to `car.Name`:

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

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

Use JSDO to create mobile and web clients

As described in Overview of Progress Data Objects, Services, and Catalogs on page 9, Progress Data Objects support the development of mobile and web apps by providing access to OpenEdge 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 server. For a reference to the basic JavaScript classes and objects that support JSDO access, see the JSDO class and object reference and JSDO properties, methods, and events reference.

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:

- JSDO overview
- Enable cookies
- Create and manage access to a JSDO instance
- CRUD and Submit operations
- Access custom Invoke operations
- Manage JSDO login sessions
- JSDOSession Management Model

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 34
- How a JSDO maps to a Data Object resource on page 35
- How JSDO memory works on page 36
- Methods of the JSDO and JSRecord classes on page 39
- Asynchronous execution on page 43
- Properties of JSDO on page 49
- Requirements to use JSDO on page 51

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 topic. For more information, see the remaining topics of this JSDO overview on page 34.
- **`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 topic. Of the two supported options (`JSDOSession` and `Session`), this is the Progress-recommended class for managing JSDO login sessions. For more information, see Manage JSDO login sessions on page 75.
- **`progress.data.JSRecord` class** — References a record object in JSDO memory, as described in the progress.data.JSRecord class topic. For more information, see How a JSDO maps to a Data Object resource on page 35 and How JSDO memory works on page 36.
- **`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 topic. This is an alternative to the `progress.data.JSDOSession` class, which Progress recommends for managing JSDO login sessions. For more information, see Manage JSDO login sessions on page 75.

- **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 topic and the topics of this JSDO overview on page 34. For more information, see CRUD and Submit operations on page 54 and Access custom Invoke operations on page 69.
- **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) topic. For more information, see Table and field references on page 36 and Working records on page 37.

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 10, 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 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.

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). 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). 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

A table reference is a property on a JSDO instance that maps to a table in the resource being accessed after `fill( )` is called. For example, if you call `fill( )` on a `SportsService` with a `CustomerOrders` resource that has two tables, `ttCustomer` and `ttOrders`, the JSDO will have two table references named `ttCustomer` and `ttOrders`. The following example demonstrates how to populate and use table references:

```
// custOrders.ttCustomer and custOrders.ttOrders will be undefined before fill( ) is
called.
custOrders.fill( );
custOrders.ttCustomer.foreach((customer) => {
    if (customer.data.Name === "Jim Robert") {
        customer.assign({State: "Missouri"});
    }
});
custOrders.ttOrders.foreach((order) => {
    if (order.data.OrderNum === 50052) {
        order.assign({OrderStatus: "Shipped"});
    }
});
```

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).

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 37.

Working records

Note: This functionality is provided for users who are more experienced with ABL than JavaScript syntax. This functionality does no conform to object oriented or functional language conventions, and therefore should be avoided in favor of table and field references and standard JSDO functions.

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, allowing you to access its field values at any time during the same JSDO session.

For example in Table and field references on page 36, 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:

```
custOrders.ttCustomer.foreach((customer) => {
    if (customer.data.Name === "Jim Robert") {
        custOrders.ttCustomer.SalesRep = "Bob James";
    }
});
```

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 39). The documentation for each JSDO method (see JSDO properties, methods, and events reference) 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 5: 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
`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). 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 = custOrders.record.data.Name; // Single table only
var custName = custOrders.ttCustomer.Name; // One table of many
var custName = custOrders.ttCustomer.record.data.Name;
var custRecord = custOrders.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).

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 39.

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 execution on page 43. 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:

- **Execute standard Data Object operations and call 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.

- **Execute 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:
  - **Call 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).

  - **Call 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.

- **Update 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.
- **`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.

- **Move 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.
- **Sort 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.

- **Search 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.
- **Return 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.
  - **`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.
- **Manage 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 execution on page 43.

- **Create and access 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 and execute synchronously. For more information on user-defined properties, see Properties of JSDO on page 49.

Asynchronous execution

JSDO methods operate locally on the client and execute synchronously in JavaScript. Synchronous execution means that the client waits for the method to finish 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.

Some JSDO methods 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 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.

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 Named events on page 44.
- Whether your environment supports Promises. For more information, see Promises on page 46.

Compare 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 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 CRUD and Submit operations on page 54.

Asynchronous execution is also the default for custom invocation methods 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 Access custom Invoke operations on page 69.

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.

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.
  - `afterFill` (or its alias, `afterRead`) — Fires after the Data Object Read operation completes execution.
- For the standard `saveChanges( )` method:
  - `beforeSaveChanges` — Fires before the method executes any Data Object operations.
  - `beforeDelete` — Fires before any Data Object Delete operation executes and before any record delete initiates as part of a Submit operation.
  - `afterDelete` — Fires after each Data Object Delete operation completes execution and after any record delete completes as part of a Submit operation.
  - `beforeCreate` — Fires before any Data Object Create operation executes and before any record create initiates as part of a Submit operation.
  - `afterCreate` — Fires after each Data Object Create operation completes execution and after any record create completes as part of a Submit operation.
  - `beforeUpdate` — Fires before any Data Object Update operation executes and before any record update initiates as part of a Submit operation.
  - `afterUpdate` — Fires after each Data Object Update operation completes execution and after any record update completes as part of a Submit operation.
  - `afterSaveChanges` — Fires after all the Data Object operations executed by the method have completed (including a single Submit operation, if applicable).

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.
  - `afterInvoke` — Fires after the specified Data Object Invoke operation completes execution.

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 after all 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.

Promises

The JSDO allows most asynchronous methods to return results using ES6 Promises (if available) in addition to any results returned using named events (see Named events on page 44). This Promise support is available for the following standard JSDO methods:

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

Asynchronous JSDO methods return ES6 Promises. You can call methods on the returned Promise to register functions that will be called depending on whether the Promise resolves or rejects. Each entry for the JSDO methods in the JSDO properties, methods, and events reference show several examples of Promises.

The `fill()` and `saveChanges()`methods 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.

Each Promise will be resolved or rejected with an object that contains the outcome of the operation. The following functions return Promises:

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

The objects returned by Promises contain the following fields:

- `jsdo`
- `success`
- `request`

This allows you to share callbacks for both subscription to appropriate named events and registration by appropriate Promise methods. 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 in the JSDO properties, methods, and events reference.

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 a JSDO constructor. 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.

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. 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. 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. This parameter represents the record object in JSDO memory that is created, updated, or deleted.

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

The following code sample illustrates how to use event subscriptions:

```
const progress = require("@progress/jsdo-core").progress;

// This will be called after fill() is called
function onAfterFillCustomers (jsdo, success, request) {
  if (!success) {
    console.log("Error: status: " + request.xhr.status);
  }
  else {
    // Make some changes and save them to trigger the 'AfterSaveChanges' event
    jsdo.add({EmpNum: 1, LastName: "TEST" });
    jsdo.saveChanges(true);
  }
}

// This will be called after saveChanges() is called in onAfterFillCustomers
function onAfterSaveChanges (jsdo, success, request) {
  console.log("DEBUG: AfterSaveChanges: " + success + " errors: " + jsdo.getErrors());
}

// This assumes you have already set up a JSDOSession
let employee = new progress.data.JSDO({name: 'Employee'});
employee.subscribe('AfterFill', onAfterFillCustomers, this);
employee.subscribe('AfterSaveChanges', onAfterSaveChanges, this);
employee.fill();
```

Handle 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 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. 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, 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 as the corresponding output parameter, and has a JavaScript data type that maps to the ABL data type of the output parameter. 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 description.

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 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.

Error handling for Data Object operations

For any Data Object operation that completes with an error of any type, the `success` property of the returned request object (and the *success* parameter passed to any event or Promise callback) is set to `false`. 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. 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 by invoking the `getErrorString( )` method on the record object.

You can change this default behavior by setting the `autoApplyChanges` property to `false`. In this case, when the operation completes you can accept or reject the 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 available for return until the next call to either `fill( )` or `saveChanges( )`.

Properties of 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 36)
- 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 Error handling for Data Object operations on page 48)

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 Create and manage access to a JSDO instance on page 52.

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:

- setProperty( ) method
- setProperties( ) method
- getProperty( ) method
- getProperties( ) method

Requirements to use JSDO

The basic requirements for using 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 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 Additional development options on page 11.
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( )` , 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 Manage JSDO login sessions on page 75.

   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 Manage JSDO login sessions on page 75.

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 Create and manage access to a JSDO instance on page 52.

   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 CRUD and Submit operations on page 54 and Access custom Invoke operations on page 69.
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( )`. 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 Create and manage access to a JSDO instance on page 52.

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.

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

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.

Create and manage access to a JSDO instance

The following code fragment demonstrates how to create and use a JSDOSession and a JSDO:

```
let session, customer;
        // create a JSDOSession for a specified web application
        // that requires BASIC authentication
        try {
            progress.data.getSession({
```

```
            serviceURI: 'https://oemobiledemo.progress.com/OEMobileDemoServicesBasic/',

               catalogURI:
'https://oemobiledemo.progress.com/OEMobileDemoServicesBasic/static/CustomerService.json',

               authenticationModel: 'basic',
               username: 'basicuser',
               password: 'basicpassword'
           })
           .then((object) => {
               // The object that getSession() returns contains three fields:
               // object.JSDOSession, object.result, object.info
               session = object.jsdosession;
               customer = new progress.data.JSDO('Customer');
               return customer.fill();
           }, (object) => {
               console.log('getSession() failed');
               return false;
           })
           .then(function (object) {
               // The object that fill() returns contains three fields:
               // object.JSDO, object.success, object.info

               // Do other JSDO related tasks here because fill() succeeded
               console.log(object.success);
               console.log(object.jsdo.hasData());
               return session.invalidate();
           })
           .then(function(object) {
               console.log("Session is invalidated!");
               return true;
           });
       } catch (ex) {
           console.log("Exception: " + ex.message);
       }
```

This fragment relies on a public Progress web application with a `CustomerService` that has a JSDO Catalog.

The call to `getSession()` instantiates and authenticates a `JSDOSession`. After it is successful, a JSDO is created for the `Customer` resource. The `fill()` method is then called to read the `Customer` resource data. The second `then()` handles the result of `fill()`, and on success contains all of the `Customer` records. For security, it also invalidates the `JSDOSession` so that it cannot be reused. The final `then()` handles the results of `JSDOSession.invalidate()`. The `catch` block handles any exception thrown within the `try` block.

For more information on:

- These stand-alone functions — See getSession( ) stand-alone function.
- The `JSDOSession` methods that these functions invoke — See the progress.data.JSDOSession class description.
- Instantiating a JSDO instance and the features it provides — See the progress.data.JSDO class description.
- Implementing JSDO method calls to invoke operations on Data Object resources — See CRUD and Submit operations on page 54 and Access custom Invoke operations on page 69.
- Managing login sessions to support JSDO access to network resources — See Manage JSDO login sessions on page 75.

CRUD and Submit operations

After creating a JSDO for a `dsCustomer` resource, 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, only OpenEdge Data Object Services requires significant coding to implement server-side Data Objects. The information on JSDO in these examples refers to OpenEdge ABL used to implement a Data Object created for a DataSet resource named `dsCustomer`, which can support one or more temp-tables.

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 `sports2020` database:

  Table 6: 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 7: 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, this Business Entity must be revised as described for updating Business Entities for access by Telerik DataSources 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 descriptions of the add(), assign(), or remove() methods.
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.

CRUD examples

The following code example shows a number of client-side CRUD and submit operations that can be used with JSDO:

```
const progress = require("@progress/jsdo-core").progress;

const options = {
    catalogURI:
"https://oemobiledemo.progress.com/OEMobileDemoServices/static/CustomerService.json",
    serviceURI: "https://oemobiledemo.progress.com/OEMobileDemoServices/",
    authenticationModel: "anonymous"
};

let session,
    jsdo;

// A JSDOSession is how you connect to a service. JSDOs use the connections made by
// JSDOSessions to get data.
progress.data.getSession(options).then((object) => {
    // getSession() returns an object that has three properties: jsdosession, result,
and info
    console.log("The result of getSession() is: " + object.result);
    console.log("The JSDOSession is connected to the service at: " +
object.jsdosession.serviceURI);

    session = object.jsdosession;

    // A JSDO is a representation of a table. fill() reads records from the service
    jsdo = new progress.data.JSDO({name: "Customer"});
    return jsdo.fill();
})
.then((object) => {
    let rec;

    // fill() returns an object that has three properties: jsdo, success, and request
    console.log("The result of fill() is: " + object.success);
    console.log("The resulting HTTP status code returned is: " +
object.request.xhr.status);

   // After a fill(), you can read, modify, and delete records and then send the changes

    // back to the service via saveChanges().

   // You can iterate through the records via a foreach on the JSDO's temp-table, which

    // will return a record object with the appropriate fields in the data property
    jsdo.ttCustomer.foreach((customer) => {
        if (customer.data.SalesRep === "BBB") {
           console.log("The SalesRep BBB handles the customer: " + customer.data.Name);

            rec = customer;
        }
    });

    // You can update a record just by modifying any of its fields
    rec.data.Balance = rec.data.Balance + 1500;

    // You can also use assign() to overwrite a record
    jsdo.ttCustomer.foreach((customer) => {
        if (customer.data.Name === "All Sports") {
            // If you do not specify a value for a field, it will be overwritten to be
null
            customer.assign({
                Name: 'All Sports',
                SalesRep: 'R2D2',
                Balance: '1337',
                State: 'MN'
            });
        }
    });
```

```
    // add() creates a record
    jsdo.ttCustomer.add({
        Name: 'Robert Good',
        SalesRep: 'NS',
        Balance: '1337',
        State: 'MA',
    });

    // remove() deletes a record
    jsdo.ttCustomer.foreach((customer) => {
        if (customer.data.Name === "Bobby Bad") {
            customer.remove();
            console.log("This customer was deleted: " + customer.data.Name);
        }
    });

    // All of the changes so far are just local until you send them back to the server
    // via a call to saveChanges(). saveChanges() also updates the local records in the
 jsdo.
    return jsdo.saveChanges();
})
.then((object) => {
    // saveChanges() returns an object that has three properties: jsdo, success, and
request
    console.log("The result of saveChanges() is: " + object.success);

   // The local records are already updated but you can pass a filter to fill() to only
 get
    // a subset of the records
    return jsdo.fill('Balance = 1337');
})
.then((object) => {
    jsdo.ttCustomer.foreach((customer) => {
        console.log("We edited this customer: " + customer.data.Name);
    });

    // Always clean up your session afterwards!
    return session.invalidate();
})
.then((object) => {
   // invalidate() returns an object that has three properties: jsdo, result, and request

    console.log("The result of invalidate() is: " + object.result);
});
```

Event example

The following code example shows event handlers and demonstrates the proper sequence for firing them:

```
// This is a sample of how to use event handlers.
const progress = require("@progress/jsdo-core").progress;

const options = {
    catalogURI:
"https://oemobiledemo.progress.com/OEMobileDemoServices/static/CustomerService.json",
    serviceURI: "https://oemobiledemo.progress.com/OEMobileDemoServices/",
    authenticationModel: "anonymous"
};

let session,
    jsdo;

// Here is where we define all of our event handlers.
// Please note that the JSDO is passed to every event handler.
function beforeFill(jsdo) {
    console.log('In beforeFill()');
    console.log('The name of the table of the JSDO is: ' + jsdo.name);
}

function afterFill(jsdo) {
    console.log('In afterFill()');
}

function beforeUpdate(jsdo) {
    console.log('In beforeUpdate()');
}

function afterUpdate(jsdo) {
    console.log('In afterUpdate()');
}

function beforeSaveChanges(jsdo) {
    console.log('In beforeSaveChanges()');
}

function afterSaveChanges(jsdo) {
    console.log('In afterSaveChanges()');
}

progress.data.getSession(options).then((object) => {
    session = object.jsdosession;
    jsdo = new progress.data.JSDO({name: "Customer"});

    // Here is where we subscribe all of our event handlers
    jsdo.subscribe('beforeFill', beforeFill);
    jsdo.subscribe('afterFill', afterFill);

    jsdo.subscribe('beforeUpdate', beforeUpdate);
    jsdo.subscribe('afterUpdate', afterUpdate);

    jsdo.subscribe('beforeSaveChanges', beforeSaveChanges);
    jsdo.subscribe('afterSaveChanges', afterSaveChanges);

    console.log("Initial call to fill()");
    return jsdo.fill();

})
.then((object) => {
    let rec;

    console.log('In the success handler of fill()');

    console.log("Initial update of a record")
    jsdo.ttCustomer.foreach((customer) => {
        if (customer.data.Name === "All Sports") {
            customer.assign({
                Name: 'All Sports',
```

```
                SalesRep: 'R2D2',
                Balance: '1337',
                State: 'MN'
            });
        }
    });

    console.log("Initial call to saveChanges()");
    return jsdo.saveChanges();
})
.then((object) => {
    console.log('In the success handler of saveChanges()');

    // Always clean up your session afterwards!
    return session.invalidate();
});

// This is the output of this program when run:
// Initial call to fill()
// In beforeFill()
// The name of the table of the JSDO is: Customer
// In afterFill()
// In the success handler of fill()
// Initial update of a record
// Initial call to saveChanges()
// In beforeSaveChanges()
// In beforeUpdate()
// In afterUpdate()
// In afterSaveChanges()
// In the success handler of saveChanges()
```

Create operation

Call the `add()` method on a JSDO table reference to create a new record in JSDO memory. The fields of the new record contain the values specified and passed to the method. For any fields whose values you do not provide, 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 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.

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 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 a successful Create or Update operation makes changes to the record on the server that was sent from the client, 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.

Read operation

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 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. For more information, see the description of the fill( ) method.

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.

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 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.

Update operation

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.

Note: Do not 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. 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 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 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.

Delete operation

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 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.

Submit operation

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, Update operation example, and Delete operation example.
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.

Server-side implementation

The Server-side implementation calls an overload of the methods defined in the `OpenEdge.BusinessLogic.BusinessEntity` super class, which OpenEdge provides as an aid for implementing Data Object resources that support before-imaging. When invoking `CreateData()`, `UpdateData()`, or `DeleteData()`, a default implementation is provided to create, update, or delete a record in the server database from the single temp-table record in the ProDataSet input to the appropriate `dsCustomer()` method. The `ReadData()` method provides a default implementation for using the `filter` parameter to identify the data from the server database that fills the specified ProDataSet. The `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()`. Default implementations are location in the `OpenEdge.BusinessLogic.BusinessEntity` super class

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.
```

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 JSDO `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.

- 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.
```

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.
```

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.
```

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 `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.
```

Access 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.

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.

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.

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.

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 Promises on page 46). 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 `sports2020` database, such as `Customer` (see CRUD and Submit operations on page 54):

```
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.

Manage 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 `getSession( )` method on an instance of one of the following OpenEdge JavaScript classes:

- **progress.data.JSDOSession class** — (Recommended) Supports asynchronous access only to a web server using jQuery Promises that are returned by `JSDOSession` methods (such as `isAuthorized( )`) 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.

Using an instance of the progress.data.JSDOSession class, the `getSession( )` method takes any required user credentials as input and establishes a login session once authenticated on the specified web application. You can then create a JSDO based on the added catalog..

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.

Requirements for creating a JSDO login session

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

- Choose the web server authentication model on page 75
- Enable cookies on page 52
- Choose appropriate URIs for the app type and its deployment location on page 76
- Support page refresh for web apps on page 76

Choose 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 credentials
- **HTTP Basic authentication** — Credentials sent with every request, using Basic authentication
- **HTTP Form-based authentication** — Credentials sent using Form authentication; an authentication token is received on the initial request, which is sent back for each further request.

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 Use protected web resources on page 80.

Caution: Progress recommends that you always use TLS (HTTPS) when logging into a web application. If HTTP is used, credentials are sent as clear text. Using HTTPS is particularly important for Basic authentication because Base64 encoded credentials are sent on every request.

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 Use default web pages to support client app login on page 77.

Choose 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.

Support page refresh for web apps

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 `getSession( )` 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. You can use `getSession( )` as an alternative. It creates a new session if it is a brand new state or re-creates the old session on a page refresh.

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 and the getSession( ) stand-alone function. For more information on using the `isAuthorized( )` method, including example code, see the description of the isAuthorized( ) method.

Use 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 start up 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 start up. 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 Use protected web resources on page 80.

- **`/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 which the session must try to access. The client app typically provides a separate, unprotected login page for the user to enter their credentials, which are then used to authenticate against the web service.

  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.

Handle changes in session online status

The `progress.data.JSDOSession` instance provides 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 8: Events, methods, and properties for tracking app online status

Member	Brief description (See also the reference entry)
connected property	Returns a `boolean` that indicates the most recent online status of the current `JSDOSession` object when it last determined if the web application it manages was available.
offline event	Fires when the current `JSDOSession` 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.

Member	Brief description (See also the reference entry)
online event	Fires when the current JSDOSession objectdetects 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)	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.
pingInterval property	A number that specifies the duration, in milliseconds, between one automatic execution of the current JSDOSession object's ping( ) method and the next.
subscribe( ) method (JSDOSession class)	Subscribes a given event callback function to an event of the current JSDOSession object.
unsubscribe( ) method (JSDOSession class)	Unsubscribes a given event callback function from an event of the current JSDOSession object.
unsubscribeAll( ) method	Unsubscribes all event callback functions from a single named event of the current JSDO or JSDOSession object, or unsubscribes all event callback functions from all events of the current JSDO or JSDOSession 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 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.

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 getSession( ) method and successful execution of the invalidate( ) 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 a session has been successfully instantiated using getSession( ). 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 once a session has been successfully instantiated using getSession( ). 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 invalidate( ) method on its session instance.

Note that once a JSDO session has been successfully instantiated using getSession( ), the 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 getSession( ) on a session instance for the web application and not yet invoking invalidate( )). 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.

Use protected web resources

You can configure a client app so that it opens a protected web resource prior to logging in. 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 instantiating a session using getSession( ), 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 call getSession to initiate a session in the web application 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 `getSession( )` 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 `getSession( )` 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 `getSession( )` method to start a JSDO login session, configuring a client app to access a protected web page prior to invoking the `getSession( )` 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 `getSession( )` 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 `getSession( )` 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. Progress provides defaults so the `getSession( )` 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 `getSession( )` method, see Use default web pages to support client app login on page 77.

JSDOSession Management Model

Authentication has been assigned to a new class, `progress.data.AuthenticationProvider`. This is to reduce the load on the JSDOSession object. Now, the JSDOSession only handles server data, while the AuthenticationProvider class takes care of client authentication.

To make use of this separation of concerns, you must create a JSDOSession object using the standalone function, `getSession( )`, and not `login( )`. Upon execution, `getSession( )` performs the following actions:

- Creates an AuthenticationProvider object (of the `progress.data.AuthenticationProvider` class), which takes care of authenticating the user with the backend server
- Creates a JSDOSession object with the AuthenticationProvider object
- Executes the `addCatalog( )` method to load one or more Data Service Catalogs to the JSDOSession object

Once `getSession( )` is successfully executed, JSDO objects can be created.

```
getSession( ) – Sample Code
let session;
// Create a JSDOSession that is authenticated and has added a catalog via getSession()
progress.data.getSession({
    serviceURI: 'http://oemobiledemo.progress.com/OEMobileDemoServicesBasic/',
    catalogURI:
'http://oemobiledemo.progress.com/OEMobileDemoServicesBasic/static/CustomerService.json',

    authenticationModel: progress.data.Session.AUTH_TYPE_BASIC,
    username: user,
    password: pw
}).then((object) => {
    let jsdo;
    // We can verify that the JSDOSession is authorized by checking the
    // connected property
    console.log("JSDOSession status: " + object.jsdosession.connected);
    // Create a new JSDO now that we've got a session going
    jsdo =  new progress.data.JSDO("Customer");
    // Request data from the backend with a filter
    return jsdo.fill("CustNum < 11");
}).then((object) => {
    // We can verify that we succeeded because of the jsdo.success flag
    console.log("JSDO fill status: " + object.jsdo.success);
    // Print out the CustNum's and names of the object
    object.jsdo.ttCustomer.foreach((customer) => {
        console.log(customer.data.Custnum + ": " + customer.data.Name);
    });
});
```

To terminate a JSDOSession, use the `invalidate( )` method instead of the `logout( )` method. The `logout( )` method logs the user out of the session but retains some data in memory. The `invalidate( )` method not only terminates the user's login session but also correctly removes all its data from memory as well as prevents the JSDOSession from being reused.

```
invalidate( ) – Sample Code
// Since we're done, we can clean up the JSDOSession via invalidate()
    session.invalidate().then((object) => {
    // We can verify that the JSDOSession is no longer valid via the connected flag
    console.log("JSDOSession status: " + object.jsdosession.connected);
});
```