Progress®



The OpenEdge ABL Service Developer Guide

OpenEdge®

Copyright

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

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

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

February 2019

Last updated with new content: Release 12.0

1

Concepts

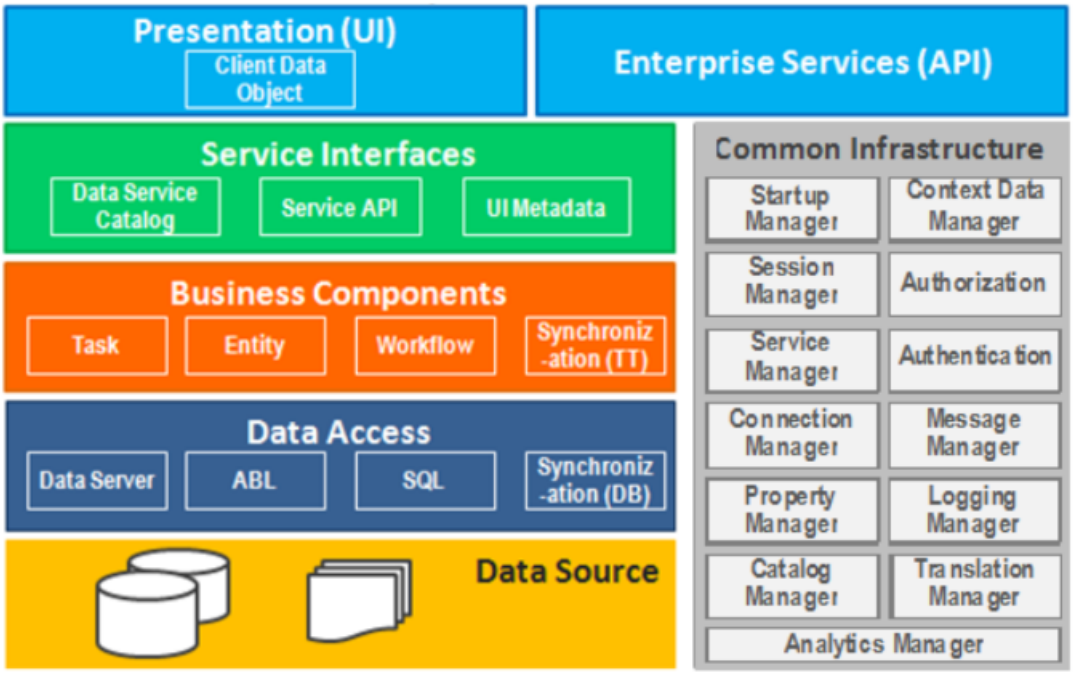
This section introduces the fundamental concepts of developing Progress OpenEdge ABL services. It describes the role of a Service Developer in the context of the OpenEdge Application Architecture. It includes a brief overview of application servers, web applications, and services, and relates these components to ABL applications deployed on Progress Application Server for OpenEdge (PAS for OpenEdge).

For details, see the following topics:

- Introduction
- Application servers, web applications, and services
- PAS for OpenEdge overview
- ABL application architecture
- Default PAS for OpenEdge instance (Windows only)

Introduction

OpenEdge Application Architecture (OEAA) is the recommended way to design a Progress OpenEdge application.



OEAA consists of the following layers:

- A **Data Access** layer that manages the retrieval of data from sources such as an OpenEdge database.
- A **Business Components** layer that contains the core business logic.
- A **Service Interfaces** layer that translates requests into a format that the Business Components layer can process.
- An **Enterprise Services** layer that exposes the business logic to applications and services that exist beyond the bounds of the OpenEdge application.
- A **Presentation** layer that contains the user interface.
- A **Common Infrastructure** layer containing framework code to support the application business logic.

Developing a Progress OpenEdge application based on OEAA typically involves many roles:

- Software Architects gather application requirements and design the architecture of the application. They identify use cases and determine the types of data, business components, and workflows that are needed for each use case.
- Business Logic Developers work on the Business Components and Data Access layers. They write business entities in their ABL code.
- Web UI Developers and Client Developers work on the Presentation layer. They develop web, Java, .NET, and other types of client applications.
- Service Developers define Service Interfaces and build the Enterprise Services layer. They develop APIs to expose functionality in the Business Components layer and develop mobile, REST, SOAP, and other types of services that are consumed by client applications.

About this guide

The primary aim of the Service Developer guide is to provide a content pathway for Service Developers. If you are a Service Developer, this guide will help you learn how to create ABL services, write PAS for OpenEdge event procedures, and implement service-level authorization.

To get the most from this guide, it is helpful to have a strong foundation in writing procedural and object-oriented ABL programs.

Note: Many of the examples assume that you are using Progress Developer Studio for OpenEdge, which is the recommended tool for developing OpenEdge ABL applications.

The role of a Service Developer

A Service Developer is responsible for ensuring that the core business logic, written in ABL, is accessible to external applications and users that require access to it.

As a Service Developer, you perform the following tasks:

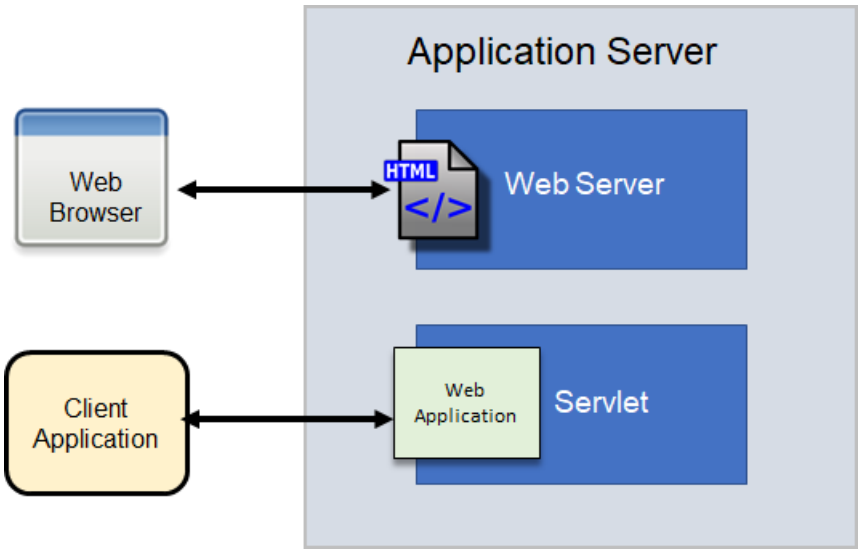
- Define service interfaces and develop ABL services for different types of client applications, including REST, SOAP, JavaScript Data Object (JSDO), and ABL clients.
- Write ABL code to manage client and user context. Context is any data that needs to be persisted over several interactions between a client application and the service. This includes things like user profile, role information, language preferences, etc.
- Implement application security to ensure that calls to the business logic layer are secure, and that only authenticated and authorized client applications and users are able to access it.
- Troubleshoot problems in services by viewing logs.

Application servers, web applications, and services

If you are new to PAS for OpenEdge, it may be useful to start by reviewing basic application server concepts. Although application servers fall under the purview of administrators and DevOps engineers, a general understanding of how they work is helpful as you develop services that are eventually deployed to an application server.

What is an application server?

An application server is a runtime framework for web applications. It contains a web server that serves static HTTP content, and a servlet container that hosts the web application. An application sever accepts client requests, passes the requests to the web application, and then sends responses back from the web application to the client application.



An application server also takes care of requirements such as load-balancing, clustering, network security, etc, thus enabling you—the developer—to focus on developing your web application.

Web applications

A web application is a set of services, each of which provides some type of functionality to a client application.

Services

The primary purpose of a service is to expose business logic to a client application. Service interfaces transform requests from a client application into a format that the business logic layer can process. For example, an OpenEdge REST service's interface may transform a JSON request message into an ABL object, a SOAP service may transform a SOAP request into ABL, etc. Each service, therefore, serves a specific type of client.

How application servers, web applications, and services work

Imagine a webpage that displays a list of customer orders to a user. When the user loads the webpage, the web client application code makes a request to the application server. The request URI may look something like this:

```
http://MyAppServerHost:8110/CustomerWebApp/OrderService/Orders/GetAllCustomerOrders
```

The URI includes the application server host and port (`MyAppServerHost:8110`), the web application name (`CustomerWebApp`) and the service name (`OrderService`).

The application server receives the request and uses the path parameters in the URI to determine the web application to pass the request to. The web application invokes the service and executes the operation (`GetAllCustomerOrders`) on the requested resource (`Orders`).

The result is customer order data that is returned by the application server to the client application. The task of displaying the data in a presentable way to the user is performed by the client application code.

A web application may contain multiple services (`OrderService`, `CustomerService`, `InventoryService`, etc) that are logically grouped based on the functionality they provide. For example, you may have a web application that contains a set of Customer services, another that contains a set of Employee services, and so on.

PAS for OpenEdge overview

PAS for OpenEdge is an application server for ABL web applications. It is based on the Apache Tomcat server and includes a web server as well as a servlet container for hosting ABL web applications.

PAS for OpenEdge is available as two separate products:

- The Progress Application Server for OpenEdge (Production)—configured as a secure web server for ABL application deployment
- The Progress Application Server for OpenEdge (Development)—configured as a web server for developing and testing ABL applications

---

Note: A Progress Developer Studio for OpenEdge installation includes the development version of PAS for OpenEdge to enable you to test and debug your services.

Key features of PAS for OpenEdge include:

- Ability to create instances
- HTTP/HTTPS for communicating with client applications
- Transports for different types of client applications
- Spring security

PAS for OpenEdge instances

A key feature of PAS for OpenEdge is that you can easily create and run more than one instance of the server, and configure the instances separately. Each instance shares the executables and libraries of a common Tomcat server, but each instance is a separate process, running in a separate JVM, with its own configuration (ports, security framework, Web applications, etc.).

Some advantages of instances are:

- Updates to the core Apache Tomcat server libraries and executables do not affect your web applications. You avoid the necessity of updating the applications and/or re-configuring them.
- You can establish different security policies for each of the instances.
- Instances provide you with quick way to create a test server for experimenting with new configurations and applications without the danger of permanently corrupting an existing server.

HTTP(S) for client communication

A PAS for OpenEdge instance uses HTTP and HTTPS to communicate with client applications. To make a request to a web application in PAS for OpenEdge, a client application must first establish an HTTP or HTTPS connection with the PAS for OpenEdge instance.

Note: We recommend using HTTPS because it is a more secure way to exchange data.

Transports

PAS for OpenEdge transports handle HTTP requests from client applications. Each ABL service is bound to a transport. When a client application makes a request, the request URI includes the transport name.

Supported transports and their assigned URLs are described in the following table:

Transport	Description	URL Path
APSV	Handles ABL clients, Java and .NET Open Clients requests. This transport also provides backwards compatibility for Classic AppServer clients over HTTP protocol.	`/apsv`
REST	Handles REST requests.	`/rest`
SOAP	Handles SOAP 1.1 requests.	`/soap`
WEB	Handles REST, classic WebSpeed, and HTTP requests, using a custom service interface called a WebHandler which is written in ABL.	`/web`

Spring security

PAS for OpenEdge uses the Spring Security framework to provide enterprise-level application security with regard to:

- Authentication — verification that username and password combinations are valid
- Authorization — enforcing access control (users are granted permissions according to their assigned roles).

Spring Security is typically configured by an administrator. After authenticating a client request, the Spring framework produces an ABL security token called the Client-Principal. The Client-Principal object encapsulates a single user identity for an OpenEdge database connection, an ABL session, or an entire application. As a Service Developer, you can use the Client-Principal object to assert the user's identity and implement authorization by using the user's role information contained in the Client-Principal object.

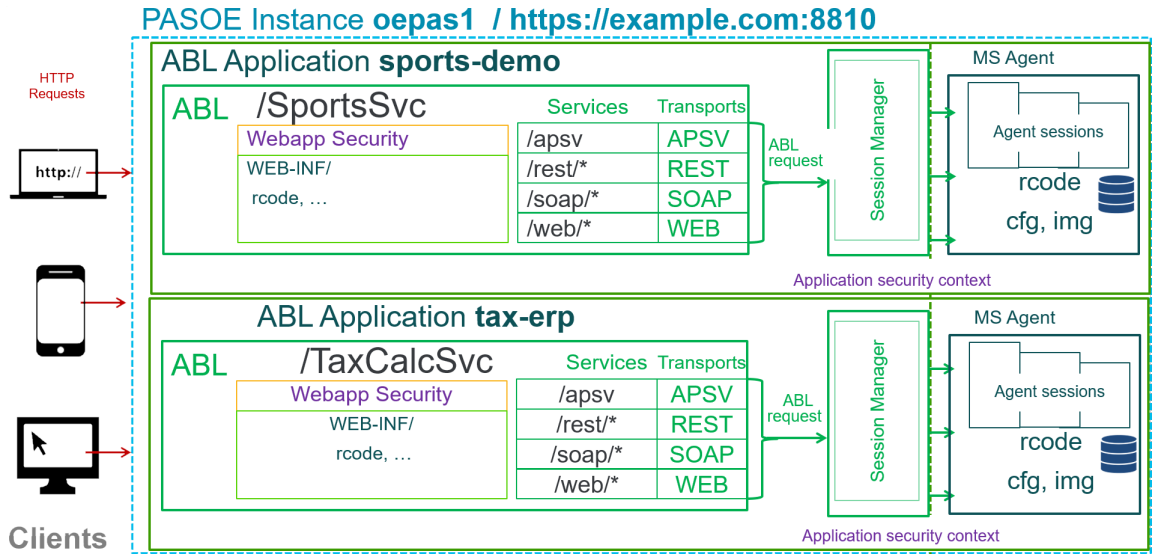
ABL application architecture

An ABL application is a container for ABL web applications. The hierarchy of components is as follows:

- A PAS for OpenEdge instance can contain multiple ABL applications.
- Each ABL application can contain multiple web applications.
- Each web application, in turn, can contain multiple ABL services.
- Each ABL service is bound to a PAS for OpenEdge transport and is mapped to ABL resources (ABL classes and procedures).

Note: As a Service Developer, you create ABL services in an OpenEdge project in Progress Developer Studio for OpenEdge. The project can be packaged into a web application archive (WAR) file and deployed to PAS for OpenEdge.

In addition to containing web applications, each ABL application has a **session manager** that launches one or more **multi-session agents** to process client requests. A multi-session agent is an operating system process that starts a session within which a client request is executed.



PROPATH

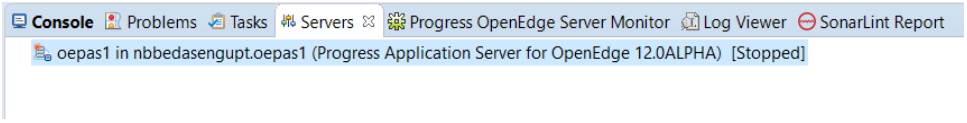
Each ABL application includes an environment variable called PROPATH, which is a list of directory paths containing compiled ABL code and other artifacts.

A PAS for OpenEdge multi-session agent starts an ABL Virtual Machine (AVM) either to process a request within a session or to run event procedures. When it does so, the AVM uses the PROPATH setting to find the code and artifacts that it needs to execute.

Default PAS for OpenEdge instance (Windows only)

A Progress Developer Studio for OpenEdge installation is shipped with a Progress Development Application Server for OpenEdge.

The Windows version of OpenEdge Developer Studio includes a PAS for OpenEdge instance called **oepas1** that is automatically created during installation. The oepas1 instance contains an ABL application with the same name (oepas1) that gets added to the **Servers** view in OpenEdge Developer Studio.



Your development environment may contain a PAS for OpenEdge instance that is set up remotely for you to use. If it does not, you can use the default oepas1 instance to test your services.

How to start and stop the default PAS for OpenEdge instance

Although the default oepas1 entry in the Servers view in OpenEdge Developer Studio is a pointer to the ABL application, you can use it to start and stop the host PAS for OpenEdge instance.

To start the instance, right-click the oepas1 entry in Servers view and select **Start**.

To stop the instance, right-click the oepas1 entry in Servers view and select **Stop**.

2

Develop services

This section describes an API-first approach to developing ABL services. It introduces the basic ideas behind API-first development and covers how to develop ABL services using each of PAS for OpenEdge's transports—WEB, REST, SOAP, and APSV.

Development choices

You can expose ABL business logic to client applications in the following ways:

- Create a REST service using one of the following methods.
  - Use the PAS for OpenEdge WEB transport (recommended). With the WEB transport, you define the service interface using a Web Handler[1] or a Data Object Handler[2].
  - Use the PAS for OpenEdge REST transport. With the REST transport, you define the service interface in a GUI editor where you map URIs to ABL classes and HTTP requests and responses to ABL method parameters.
  - Create a Data Object Service using either the WEB or REST transport. A Data Object Service provides access to a single resource and consists of both a server and a client data object. It is meant for a JavaScript Data Object (JSDO) client application
- Create a SOAP service. PAS for OpenEdge supports the deployment of a SOAP WSDL file. OpenEdge includes a tool called ProxyGenerator that enables you to generate a WSDL file from an ABL procedure that acts as a service interface.

[1] A Web Handler is an ABL class that enables you to handle GET, PUT, POST, and DELETE requests.
[2] A Data Object Handler is a type of Web Handler where, instead of writing ABL code, you define the service interface in a JSON mapping file.

- Use the PAS for OpenEdge APSV transport. In this case, client applications connect to the PAS for OpenEdge instance and remotely run ABL procedures on the server instance.

For details, see the following topics:

- APIs, services, and service interfaces
- Request URI path parameters
- API-first development
- ABL Web App project type
- Develop an ABL service using the WEB transport
- Develop an ABL service using the REST transport
- Develop an ABL service using the SOAP transport
- Develop a Progress Data Object Service

APIs, services, and service interfaces

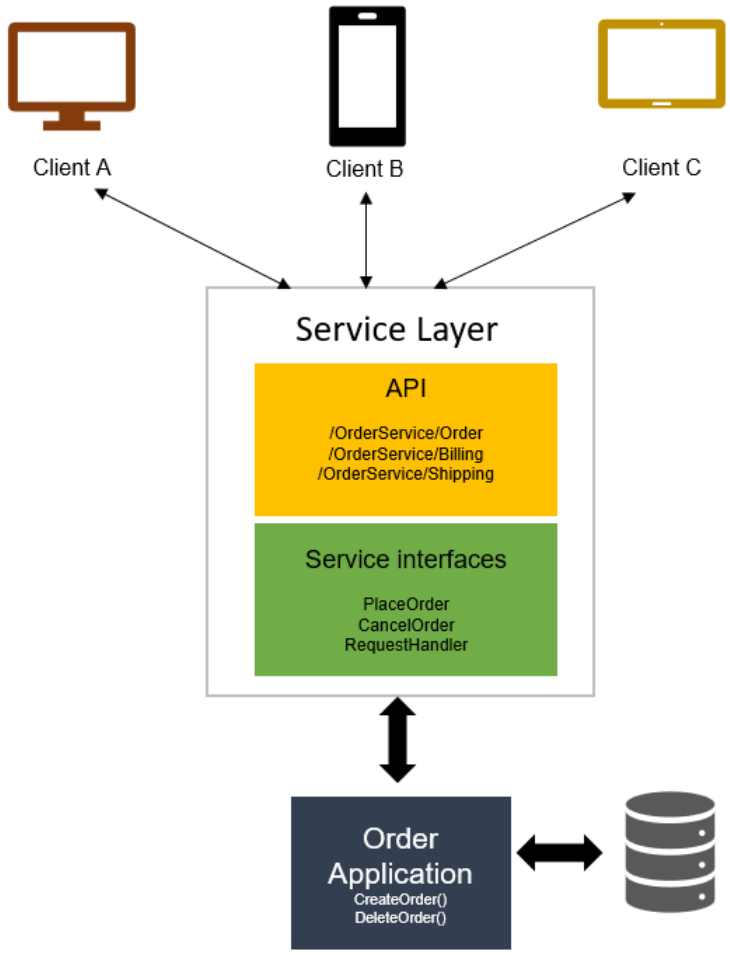
As you read through this section on developing ABL services, you'll find the terms *service*, *service interface*, and *API* used often. Let's put these terms in context.

You can think of a service as consisting of two layers:

- An API layer that exposes a set of URIs that client applications can call.
- A service interface layer, bound to the URIs, that handles client requests and transforms them (if required) so that the underlying business logic layer can process them. For example, a WebHandler is a service interface that transforms HTTP requests into ABL objects.

A service provides an API via a specific communication protocol. For example, a REST service may expose an API over HTTP.

Together, the API and the service interface provide a level of abstraction over the business logic layer, so that changes to the business logic layer do not require changes to the client application code.



Request URI path parameters

A request URI used by a client application looks something like this:

```
http://myPASServerHost:8110/OnlineStoreWebApp/web/OrderService/Order
```

It is important to know what parts of this URI you, as a service developer, control. The host and port (`myPASServerHost:8110`) and also usually the web application name (`OnlineStoreWebApp`) is determined by the administrator who deploys the web application. The remainder of the URI, which includes the transport (`web`) and the service endpoint (`/OrderService/Order`) is determined by how you design your API.



API-first development

In a modern, cloud-based environment, a web application is usually a means to an end; the goal is to expose the web application through APIs, so that it can integrate with client applications.

An API-first approach to developing services puts the concerns of the client application developer first. You begin by visualizing your application as a set of services that can be easily consumed by client applications. Complexities in your business logic layer are managed in the service interface and hidden from the client application. This leads to a better developer experience and enables you to add new functionality later on with minimal disruption to existing integrations.

An API-first approach to developing ABL services

API-first development typically consists of the following stages:

- API planning
- API design
- API development
- API testing

Plan your APIs

- Identify the types of client applications that require support.Do you have existing client applications that you need to integrate your ABL application with? What types of client applications will require support in the near future? Here are a few types of client applications:
  - REST clients
  - SOAP clients
  - JavaScript Data Object (JSDO) clients
  - ABL clients
  - Open clients (Java, .NET)
- Determine the types of ABL services you need to create. The types of client applications determine the services that you develop and the PAS for OpenEdge transports that you use.

Client type	Type of service that you need to create	PAS for OpenEdge transport
REST client	REST service	WEB or REST
SOAP client	SOAP service	SOAP
JSDO client	Progress Data Object service	WEB or REST
ABL, Java, .NET	-	APSV

To create a RESTful ABL service, you use either the WEB or the REST transport. The WEB transport is the recommended way to develop REST services. It supports all HTTP verbs and content types. Because its service interface—called a WebHandler—is an ABL class, it offers a lot of flexibility in how you handle requests and responses.

- Identify what data will be exchanged between your services and client applications. This will determine how you build the service interface layer. For example, when a user clicks a Place Order button in a client application's UI, you will want your Order API to pass the order data to a CreateOrder() method in an Order ABL class. There may be other utility classes for emailing the order, doing some XML transformation, etc, that you may not want to include in the Order API.
- Determine the data format. REST services and clients typically use JSON, SOAP services and clients use SOAP/XML, etc.

Design your APIs

Design the URIs that will be called by the client application. One way to do this is to map endpoints to operations and sample request and response data. Here is an example:

Operation	Endpoint	Request	Response
PlaceOrder	/OrderService/Order	HTTP POST `{ "orderid": "123abc", "order-date": "1/12/19", "order-line": [], "shipping": [], "billing": [] }`	Status: 200 OK
CancelOrder	/OrderService/Order	HTTP DELETE `{ "orderid": "123abc" }`	Status: 200 OK

Operation	Endpoint	Request	Response

GetOrder	/OrderService/Order?order-id={order-id}	HTTP GET	<pre>{
			"orderid":
			"123abc",
			
			"order-date":
			"1/12/19",
			
			"order-line":
			[],
			
			"shipping":
			[],
			"billing":
			[]
			}</pre>

Develop your APIs

You perform the following tasks to develop your APIs:

- Set up your OpenEdge project
- Create services that are bound to PAS for OpenEdge transports
- Develop service interfaces that are bound to URIs

The topics that follow cover these tasks.

Test your APIs

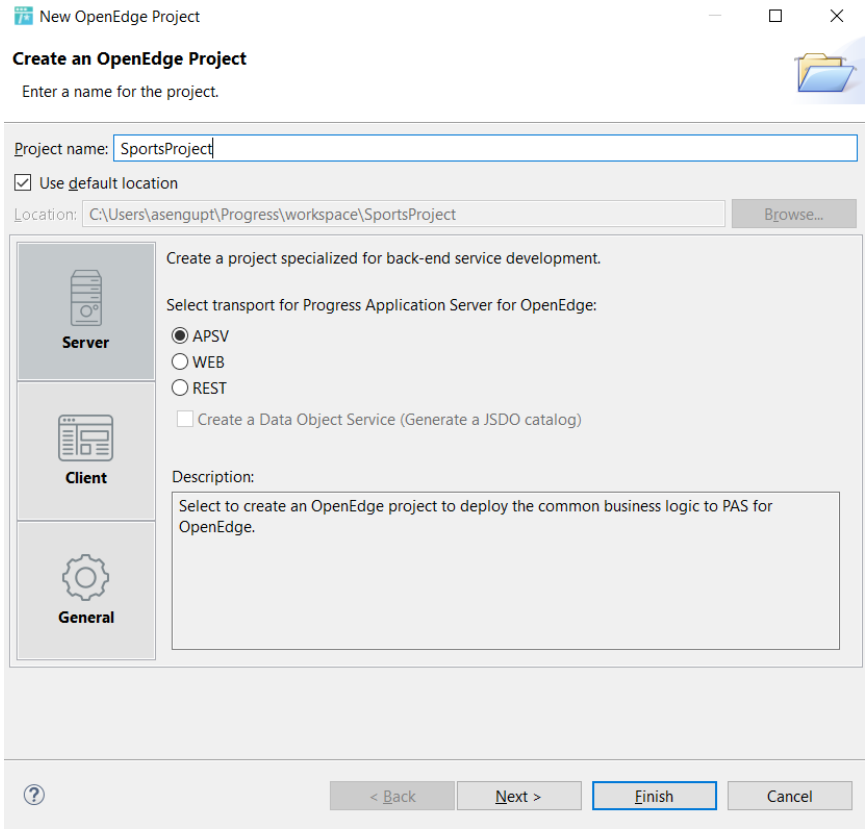
Before you can test your services, your OpenEdge project needs to be deployed to a PAS for OpenEdge instance as a web application archive (WAR).

Your environment may include a remote PAS for OpenEdge instance that is set up by an administrator. In this case, the administrator may extract ABL artifacts from your source control repository and deploy them to the instance. Alternatively, you can choose to publish the ABL services to a local development PAS for OpenEdge instance from within OpenEdge Developer Studio.

ABL Web App project type

You develop ABL services in an OpenEdge project in OpenEdge Developer Studio. We recommend using an ABL Web App project type; this is required if you are developing a REST or WEB transport-based service and you can use it for the APSV transport too.

If you are creating a new OpenEdge project, choosing WEB or REST as the transport during the project creation process adds the required ABL Web App project facets.

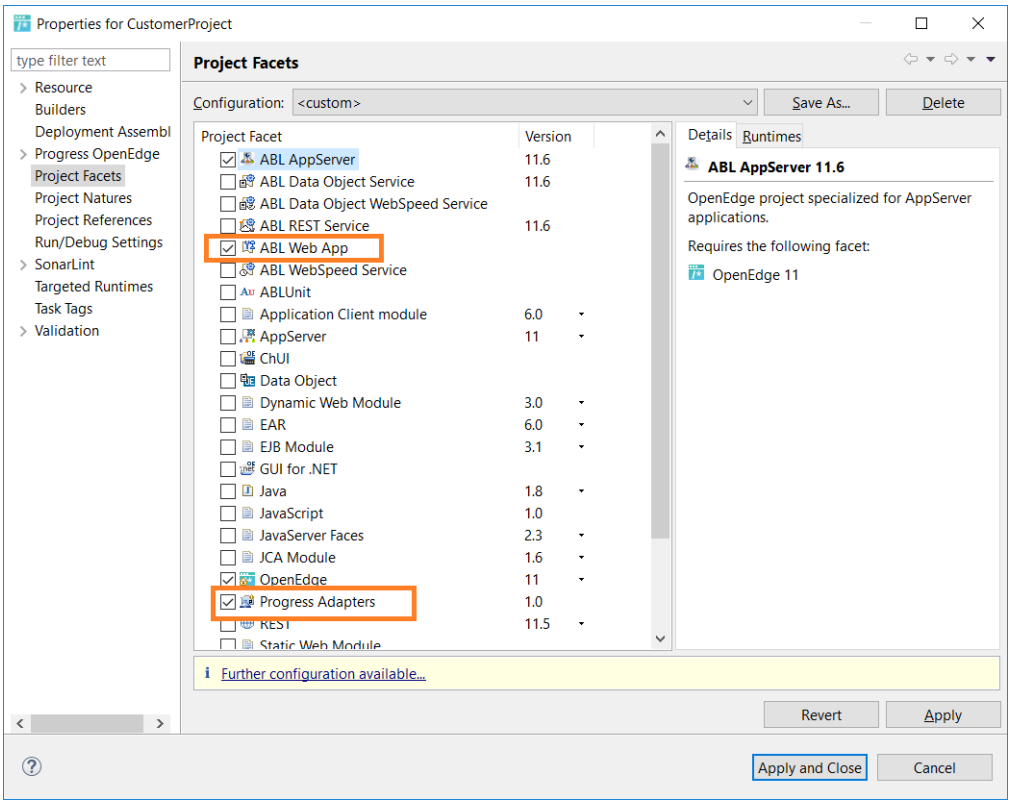


You can also add these facets to an existing project manually.

Convert an existing project into an ABL web app project

If you have an existing project that you want to convert into an ABL web app project, perform the following steps:

1. Right-click the project in the **Project Explorer** view in OpenEdge Developer Studio, and select **Properties**.
2. In the **Properties** dialog box, select **Project Facets** in the left pane.
3. In the **Project Facets** screen, select **Progress Adapters** and **ABL Web App**.



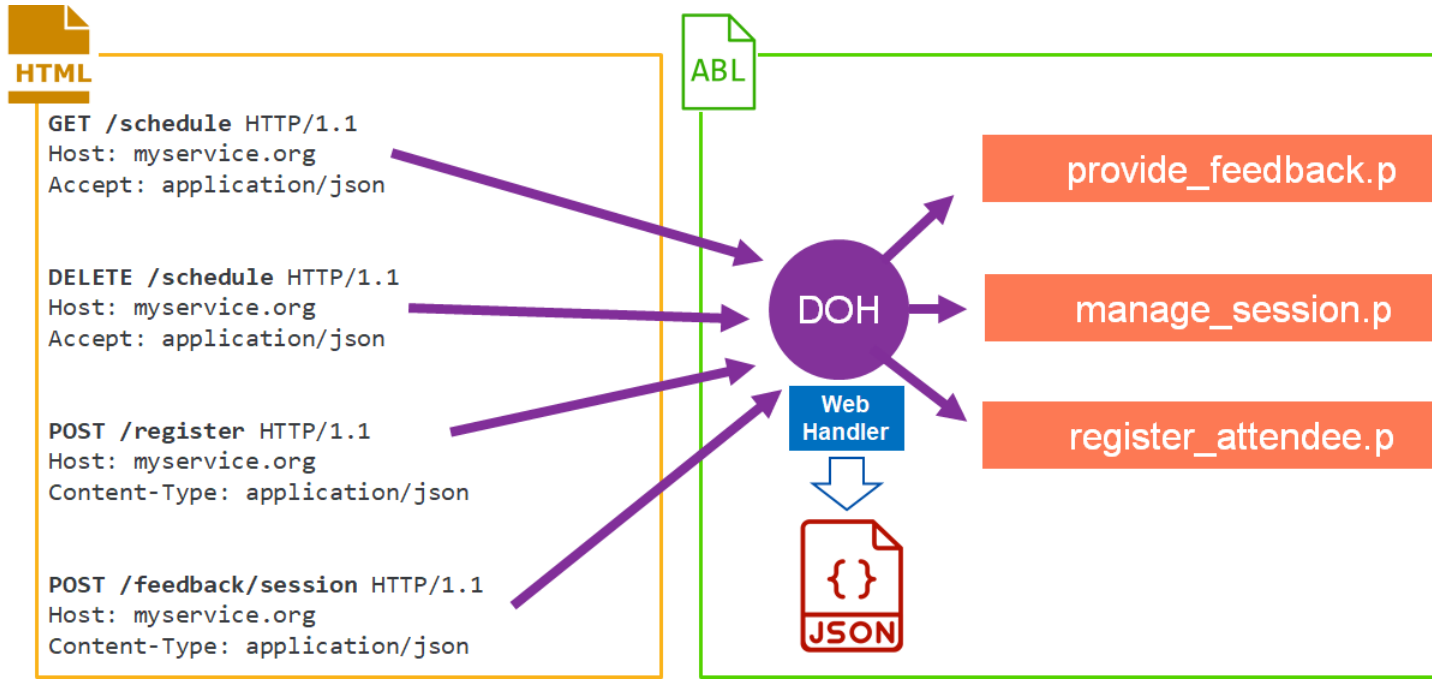
4. Click **Apply and Close**.

Develop an ABL service using the WEB transport

The WEB transport supports all HTTP methods and content types and is the recommended way to create REST ABL services.

With the WEB transport, you can choose from two types of service interfaces when creating a service:

- Web Handler
- Data Object Handler



Both service interfaces enable you to:

- Transform HTTP requests into ABL objects
- Pass ABL objects to business logic code
- Transform ABL objects into HTTP responses

Should I use a Web Handler or a Data Object Handler?

A Web Handler is an ABL class that inherits the `OpenEdge.Web.WebHandler` class. You develop the Web Handler class and write ABL code to transform the HTTP request, pass it to business logic code, etc.

A Data Object Handler is a built-in Web Handler. To use a Data Object Handler, you create a JSON mapping file where you map URIs to business logic classes, message elements to method parameters, etc.

The Web Handler gives you the flexibility to customize requests and responses before they are passed to the business logic layer or client application. However, If you don't require customizations at the service level, the Data Object Handler is easier to configure and use.

Use a Web Handler

A Web Handler is a custom ABL class that inherits the `OpenEdge.Web.WebHandler` class and contains methods that enable you to handle REST requests.

You use a Web Handler in a WEB transport service that you create. When you create the service, you define a list of URIs that get associated with the Web Handler. At runtime, when a client application makes a request to a URI, the request is passed to the associated Web Handler and processed by a method that corresponds to the HTTP verb used in the request (GET, PUT, POST, or DELETE).

Overview of steps to use a Web Handler

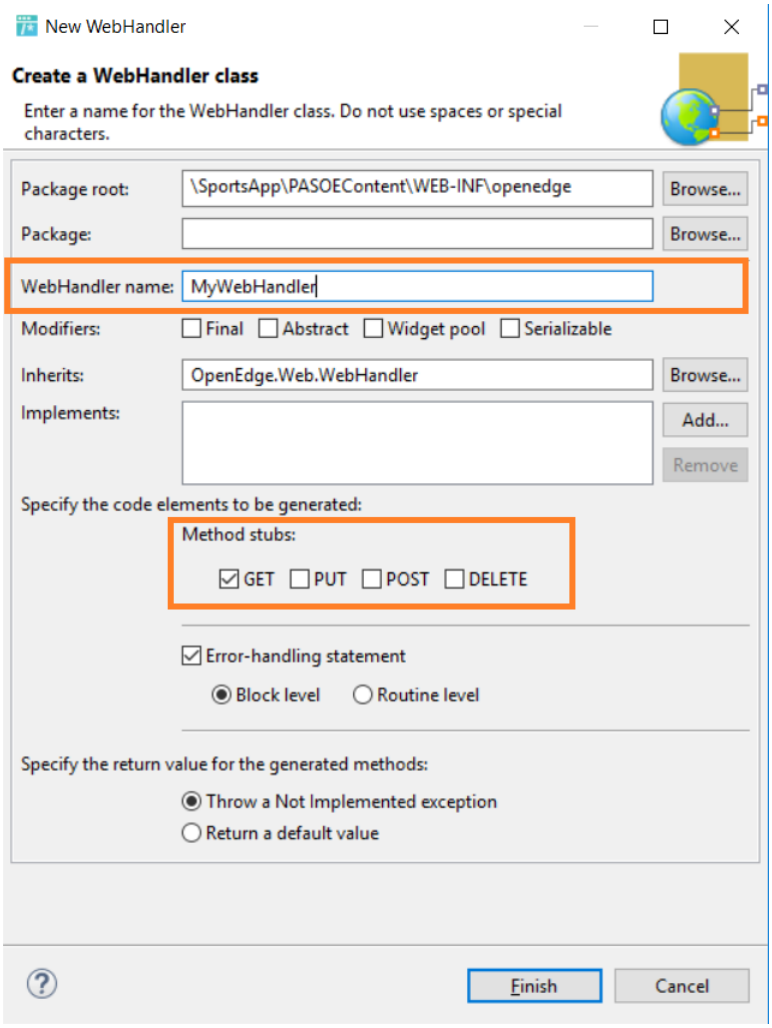
You perform the following steps to use a Web Handler:

1. Create a WebHandler class.
2. Implement methods to handle requests.
3. Create a WEB transport service.

Create a WebHandler class

Perform the following steps to create a WebHandler class file:

1. In OpenEdge Developer Studio, right-click the root project folder and select **New > WebHandler**.
2. In the **New WebHandler** dialog box, enter a name for the WebHandler class, select the method stubs that you want to create, and click **Finish**.



Note:

- Method stubs in the **New WebHandler** wizard correspond to HTTP REST methods. You *must* select the GET method.
- The **Package root** field defines the location for the WebHandler class. While the WebHandler can be stored in any location accessible to the ABL application's PROPATH, we recommend retaining the default location, which is `/PASOEContent/WEB-INF/openedge` in the root project directory.

Implement methods to handle requests

To implement methods in the Web Handler, open the WebHandler class file (the default location is `/PASOEContent/WEB-INF/openedge`).

What method stubs are generated

Method stubs in the WebHandler class are created with the prefix **Handle** (`HandleGet()`, `HandlePut()`, etc). In addition to method stubs of your choice, stubs are created for two methods—`HandleNotAllowed()` and `HandleNotImplemented()` to enable you to handle unsupported requests.

How to pass requests to business logic

Each method stub takes one input parameter—an `OpenEdge.Web.IWebRequest` object with the default name `poRequest`. This input parameter represents the client application's request message and is automatically passed to the method at runtime, based on the request's HTTP method.

You can perform various types of operations on this request object within a method. For example, you can extract path parameter values from the object, pass the values to an ABL class that contains business logic, etc.

```
METHOD OVERRIDE PROTECTED INTEGER HandleGet( INPUT poRequest AS OpenEdge.Web.IWebRequest
 ):
   ...
   ...
   DEFINE VARIABLE CustomerName AS CHARACTER NO-UNDO.
   DEFINE VARIABLE Customer AS Customer.
   Customer = NEW Customer().
   CustomerName = Customer:getCustomerName(poRequest:GetPathParameter("custid")).


END METHOD.
```

How to return a response

To return a response, you perform two steps:

- Construct an `OpenEdge.Web.WebResponse` object using the `IHttpResponse` interface to build a response message with headers, status code, body, etc.

```
METHOD OVERRIDE PROTECTED INTEGER HandleGet( INPUT poRequest AS
OpenEdge.Web.IWebRequest ):

  ...
  ...
  DEFINE VARIABLE oResponse AS OpenEdge.Net.HTTP.IHttpResponse NO-UNDO.
  DEFINE VARIABLE oWriter   AS OpenEdge.Web.WebResponseWriter  NO-UNDO.
  DEFINE VARIABLE oBody     AS OpenEdge.Core.String            NO-UNDO.
  ...

  CustomerName = Customer:getCustomerName(poRequest:GetPathParameter("custid")).
  ASSIGN
     oResponse            = NEW OpenEdge.Web.WebResponse()
     oResponse:StatusCode = INTEGER(StatusCodeEnum:OK)
     .
  ASSIGN
      oBody = NEW OpenEdge.Core.String(CustomerName).
  ASSIGN
      oResponse:Entity        = oBody
      oResponse:ContentType   = 'application/json':u
      oResponse:ContentLength = oBody:Size
      .


END METHOD.
```

- Pass the `WebResponse` object to a `WebResponseWriter()` and call `Open()` and `Close()` on it.

```
ASSIGN
        oWriter = NEW WebResponseWriter(oResponse).
        oWriter:Open().
        oWriter:Close().
```

Examples

These examples show how to use a Web Handler to address common use cases.

How to get a path parameter value

A path parameter is a segment in the request URI. A path parameter can be a variable. In this example, the number **2** at the end of the URI is a customer ID.

```
http://MyAppServerHost:8810/SportsApp/web/CustomerService/Customer/2
```

To get a path parameter value, pass the parameter name to the `GetPathParameter()` method.

```
CustomerName = Customer:getCustomerName(poRequest:GetPathParameter("custid")).
```

For this to work, the path parameter name (`custid`) must be specified in the list of URIs handled the WEB transport service (see the section *Create a WEB transport service*).

How to get a query parameter value

A query parameter is a query specified in the request URI. Here is an example of a customer ID passed as a query parameter:

```
http://MyAppServerHost:8810/SportsApp/web/CustomerService/Customer?custid=2
```

To get a query parameter value, use `GetContextValue("QUERY_STRING")`. This returns a string containing the query. If there are multiple queries, joined by the '&' symbol (for example `custid=2&orderid=2`) you get all the queries in a single string. You can extract values from the query parameters as shown in this example:

```
DEFINE VARIABLE cQryString AS CHARACTER NO-UNDO.
cQryString = STRING(poRequest:GetContextValue("QUERY_STRING")).

DEFINE VARIABLE cPair AS CHARACTER NO-UNDO.
DEFINE VARIABLE ix AS INTEGER   NO-UNDO.
DEFINE VARIABLE cKey AS CHARACTER NO-UNDO.
DEFINE VARIABLE cValue AS CHARACTER NO-UNDO.
REPEAT ix = 1 TO NUM-ENTRIES(cQryString,'&'):
   cPair = ENTRY(ix,cQryString,'&').
   cKey = ENTRY(1,cPair,'=').
   cValue = ENTRY(2,cPair,'=').
   oWriter:Write("query string pair: " + cKey + " / " + cValue + "<br
      />").
END.
```

How to get headers

To get the value of a header by the header name:

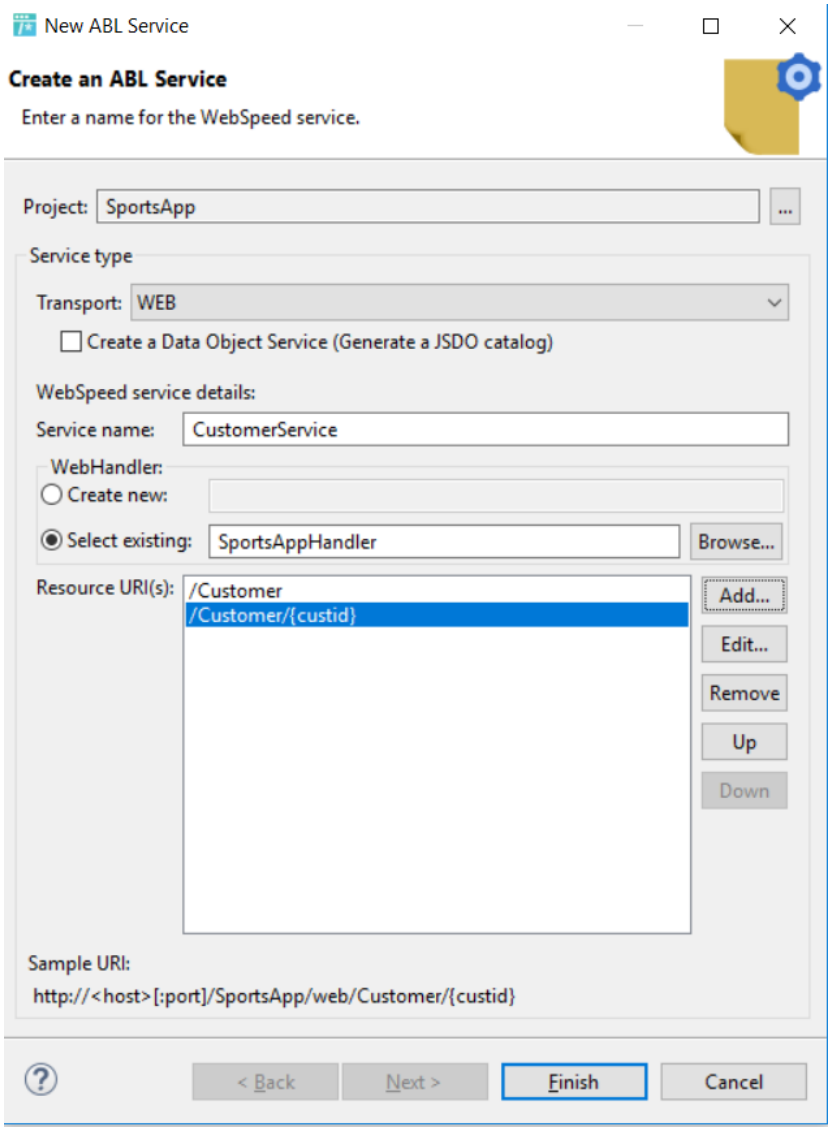
```
poRequest:GetHeader("User-Agent"))
```

Create a WEB transport service

A WEB transport service is an ABL service that uses the WEB transport and a Web Handler to process requests. It contains a list of URIs that are handled by the Web Handler. When a client application makes a request to a URI in the service, the Web Handler that is associated with the service handles the request.

Perform the following steps to create a WEB transport service:

1. Right-click the project in OpenEdge Developer Studio and select **New > ABL Service**.
2. In the **New ABL Service** wizard:
   - Choose WEB as the transport.
   - Enter a name for the service.
   - Choose **Select existing** and specify the WebHandler class that you want to associate with the service. The **Browse** button enables you to search for existing WebHandlers, including those that you created.
   - Specify URIs that will be handled by the service by clicking the **Add** button in the **Resource URIs** section. Specify variable path parameters by enclosing them in curly braces (for example `{custid}`).



Use a Data Object Handler

The Progress OpenEdge Data Object Handler (DOH) is a built-in Web Handler that enables you to design your own ABL service interface and expose ABL data objects as RESTful resources in a configurable way. It is available via the **WEB** transport in Progress Application Server for OpenEdge (PAS for OpenEdge).

If you have developed Progress ABL Data Object Services, you have already used the DOH under the hood.

How the DOH works

The DOH class implements the `Progress.Web.IWebHandler` interface, and is a more dynamic and business-ready form of the `OpenEdge.Web.WebHandler` class. With the DOH, you create an ABL service, set it to use the `DataObjectHandler` class, and then map custom service endpoints with ABL classes and procedures in a JSON mapping file.

At runtime, when a service endpoint is called, the DOH transforms the HTTP request into an ABL object and invokes the associated ABL class or procedure. It also transforms the output from the ABL class or procedure into an HTTP response.

Should I use the DOH or a mapped REST service?

Although you can also define a custom service interface using a mapped REST service (that uses the PAS for OpenEdge **REST** transport), a DOH-based ABL service offers the following advantages over the mapped REST service:

- It supports more content types.
- It is easier to debug, since the DOH is an ABL class. You can also readily inspect and modify its JSON mapping file.
- It is more customizable; you can extend its functionality by accessing its class events. To learn more about these class events, see the white paper on using the Data Object Handler.
- It uses the WEB transport, which is the recommended way to expose your ABL applications as RESTful services. This is because the WEB transport uses the ABL `WebHandler` class and thus provides better insight and control over your ABL service.

Overview of steps to use the DOH

1. **Create an ABL service** — You begin by creating an ABL service that uses the DOH class.
2. **Create a mapping file** — You then create a mapping file with the same name as the ABL service.
3. **Define custom service endpoints** — You start describing the service interface in the mapping file by defining your custom endpoints.
4. **Associate HTTP verbs with operation handlers** — For each service endpoint, you associate an HTTP verb (GET, PUT, POST, etc) with one of the following operation handlers:
   - `void`, that returns only a status code
   - `file`, that returns the contents of a file
   - `entity`, that invokes an ABL class or procedure

Create an ABL service

The first step is to create an ABL service that uses the DOH. Perform this task as follows:

1. Right-click the root project name in the **Project Explorer** view in Progress Developer Studio and select **New > ABL Service**.
2. Select **WEB** in the **Transport** dropdown.
3. Enter a name for the service, (such as HelloService).
4. In the **Web Handler** section, choose **Select existing** and then browse and select **OpenEdge.Web.DataObject.DataObjectHandler**.

5. Add a single resource URI that matches the service name (/HelloService).
6. Click **Finish**.

Create a mapping file

After creating a DOH-based ABL service, you describe its service interface in JSON in a mapping file with the extension `.map`. The DOH uses this mapping file to execute the service's operations. If a client application attempts to call an endpoint or service that is not described in the mapping file, the DOH returns a `404` (Not Found) error.

Create a new text file in your ABL web app project's `/PASOEContent/WEB-INF/openedge` folder with the same name as the ABL service and with the extension `.map` (for example, **HelloService.map**). The matching filename instructs the DOH to use the mapping file for the service's operations. Placing it in the "**openedge**" folder makes it discoverable in the application PROPATH.

At minimum, you need to define a single service within a `services` object. Like this:

```
{    "services":
      {
        "HelloService": {
        "version": "1.0.0"
      }
    }
 }
```

The service name must match the ABL service and resource URI name that you set while creating the ABL service. The "**version**" property defines the version of the service and should follow the Semantic Versioning pattern (for example,. "**1.0.0**"). The JSON snippet, as shown in this example, provides just enough to make the service discoverable but does not do much else. Without this minimum configuration, the server would only return an HTTP-404 response for the service.

Define custom service endpoints

In the JSON mapping file, define custom service endpoints under an `operations` object. For example:

```
{
 "services": {
  "HelloService": {
   "version": "1.0.0",
   "operations": {
    "/Greeting": {},
    "/Message": {},
    "/": {}
   }
  }
 }
}
```

This registers the endpoints with the DOH class and makes them accessible as REST resources. Client applications access these resources using the full endpoint URI. For example:

- `http://localhost/CustomerApp/web/HelloService/Greeting`
- `http://localhost/CustomerApp/web/HelloService/Message`

Associate HTTP verbs with operation handlers

For each custom service endpoint, you associate HTTP verbs with operation handlers. You can set up operation handlers for all HTTP verbs that are supported by the WEB transport. These include:

- GET
- PUT
- POST
- DELETE
- HEAD
- OPTIONS
- TRACE
- PATCH

Specify each verb that you want to use as a JSON object under the service endpoint in the mapping file. For example:

```
{
 "services": {
  "HelloService": {
   "version": "1.0.0",
   "operations": {
    "/Greeting": {
     "GET": {},
     "PUT": {},
     "POST": {}
    },
    "/Message": {
     "GET": {},
     "PUT": {}
    },
    "/": {
     "GET": {}
    }
   }
  }
 }
}
```

Content types

Within each verb, define the MIME content type of the data that is returned by the endpoint. For example:

```
"/Greeting": {
 "GET": {
  "contentType": "application/json"
 }
}
```

All MIME types (`text/`, `image/`, `audio/`, `video/`, `application/`, `multipart/`) are supported.

Status codes

In addition to the content type, define the status code that is returned by the endpoint. For example:

```
"/Greeting": {
 "GET": {
  "contentType": "application/json",
  "statusCode": 200
 }
}
```

Operation handlers

Next, map each HTTP verb with one operation handler. The DOH class provides the following operation handlers:

- `void`, that returns only a status code
- `file`, that returns the contents of a file
- `entity`, that invokes an ABL class or procedure

To map an HTTP verb with an operation handler, nest the operation handler under the verb as shown in this example:

```
"/Greeting": {
 "GET": {
  "contentType": "application/json",
  "entity": {}
 },
 "PUT": {
  "contentType": "application/json",
  "entity": {}
 }
}
"/Message": {
 "GET": {
  "contentType": "application/json",
  "file": "filepath"
 }
}
"/": {
 "GET": {
  "contentType": "application/json",
  "void": null
 }
}
```

The `file` and `void` operation handlers are simple key-value pairs whereas the entity handler is a JSON object. This is because the entity handler requires further information (such as the ABL class or procedure name, the method that will be invoked, etc).

How to use operation handlers

Here is a detailed description of how you can use each of the DOH's operation handlers.

`void`

The `void` operation handler returns only the status code that is defined for the HTTP verb. This is useful in certain situations. For example, when you want to reserve an endpoint for future enhancements (via a `501` - Not Implemented status code) or when you want to use it to confirm that the server is alive (`204` - No Content).

Use the `void` handler as shown in this example:

```
"/CustomizedGreeting": {
 "GET": {
  "contentType": "application/json",
  "statusCode": 501,
  "void": null
 }
}
```

`file`
The `file` operation handler returns the contents of a file, as long as the contents of the file conform to a supported MIME type.

Note: For the `file` operation handler to work, the file must be placed within the web app directory, but not in the `WEB-INF` folder for security reasons. The `static` folder is the recommended location for it.

Since the web app may be deployed to different PAS for OpenEdge instances, as the ABL code gets moved from one environment to another (for example, from development to testing or from testing to production), the best practice is to reference the PAS for OpenEdge and web app directories using environment variables:

Directory	Environment variable
PAS for OpenEdge instance root	`${env.CATALINA_BASE}`
Web app directory (located in `webapps` in the PAS for OpenEdge instance root directory)	`${web.webapp}`

For example:

```
"/Message": {
 "GET": {
  "contentType": "application/json",
  "statusCode": 200,
  "file": "${env.CATALINA_BASE}/webapps/${web.webapp}/static/message.json"
 }
}
```

`entity`
You use the `entity` operation handler to map an HTTP verb with an ABL procedure or an ABL class method. At runtime, when the verb is used in an HTTP request to the service endpoint, the associated ABL procedure or class method gets executed.

Note: Although this operation handler is called `entity`, its use is not limited to ABL Business Entities. You can use it to reference any ABL class or procedure.

Here is a simple example of the entity handler:

```
"/Greeting": {
 "GET": {
  "entity": {
   "name": "Customer.Greeting",
   "function": "getPersonalizedGreeting",
   "arg": [
    {
     "ablName": "Cust-Name",
     "ablType": "CHARACTER",
     "ioMode": "INPUT",
     "msgElem": {
      "type": "query",
      "name": "CustomerName"
     }
    },
    {
     "ablName": "GreetingMessage",
     "ablType": "CHARACTER",
     "ioMode": "OUTPUT",
     "msgElem": {
      "type": "field",
      "name": "Message"
     }
    }
   ]
  }
 }
}
```

The `name` property identifies the ABL class (`Customer.Greeting`) that is to be referenced; the `function` property denotes the method (`getPersonalizedGreeting`) in the class that needs to be invoked. The `ioMode` property identifies the parameter type (`INPUT`, `OUTPUT`, etc).

The `arg` property (short for arguments) is an array of JSON objects. Each object maps to an input or output parameter that is required by the `getPersonalizedGreeting` method. The first three properties in each `arg` object (`ablName`, `ablType`, and `ioMode`) identify the ABL parameter and correspond to how the parameters are defined in the ABL method:

```
METHOD getPersonalizedGreeting(INPUT Cust-Name AS CHARACTER, OUTPUT GreetingMessage AS
 CHARACTER):
...
END.
```

The last property in each `arg` object--`msgElem`--defines how the parameter maps to a message element in the HTTP request or response. Parameters of the type INPUT retrieve their values from message elements in HTTP requests. Parameters of the type OUTPUT pass on their values (returned by the ABL method or procedure) to message elements in HTTP responses. The `msgElem` object therefore consists of two properties:

- The `type` of message element in an HTTP request or response. Examples include `query`, `header`, etc.
- The `name` of the message element in an HTTP request or response. Examples include `customer-id` (for a `query`), From, Host (for a `header`), etc.

At runtime, the DOH uses these property mappings to convert HTTP requests and responses into ABL objects. In some cases, such as for parameters that are of the type INPUT-OUTPUT, you may need to map HTTP requests as well as HTTP responses to the same parameter. In this case, you can use `msgElem` as an array as shown here:

```
"arg": [
 {
  "ablName": "CustomizedMessage",
  "ablType": "CHARACTER",
  "ioMode": "INPUT-OUTPUT",
  "msgElem": [
   {
    "type": "query",
    "name": "customer-name"
   },
   {
    "type": "field",
    "name": "message"
   }
  ]
 }
]
```

These examples illustrate a couple of different ways in which you can use the entity operation handler. The complete list of entity handler properties and the types of values that you can set for each property is described in the *Entity Handler properties* section.

Entity handler properties

You use the `entity` operation handler as a JSON object with the following properties:

- `type` — use this property to identify the type of ABL program. Possible values include `class` and `procedure`. If no value is set, the DOH uses `class` as the default value. If you are referencing a procedure, you must set the value to `procedure`.
- `name` — the name of the ABL class or persistent procedure, as is set in your application's PROPATH. If the ABL class is contained in a package, specify the full package name (for example, `Business.Customer`). If the ABL procedure file is in a folder in the ABL web app project, specify the relative path (for example, `Business/Customer`).
- `function` — the name of the ABL class method, procedure, or user-defined function that needs to be invoked.
- `arg` — a JSON array of objects to map the method or procedure's input and output parameters to message elements in the HTTP request or response. Each object in the array represents the mapping of a single parameter to HTTP message elements. Each parameter object contains the following properties:
  - `ablName` — the name of the ABL input or output parameter. If the parameter type is RETURN, as set in the `ioMode` property (see below), skip this property.

- `ablType` — the datatype of the ABL parameter. Possible values include `CHARACTER`, `LONGCHAR`, `INTEGER`, `INT64`, `DECIMAL`, `LOGICAL`, `ROWID`, `RECID`, `DATE`, `DATETIME`, `DATETIME-TZ`, `RAW`, `MEMPTR`, `DATASET`[3], `TABLE`[4], `BLOB`, `CLOB`, `EXTENT`[5], and `CLASS`[6].
- `ioMode` — the parameter type. Possible values include `INPUT`, `OUTPUT`, `INPUT-OUTPUT`, and `RETURN`.
- `msgElem` — a JSON object or array of objects that maps the ABL parameter to a message element in the HTTP request or response. Each `msgElem` object contains two properties--`type` and `name`. The value of the `name` property depends on the value of the `type` property. Specify values for `type` and `name` as described in this table.

Type	Name	When to use	Use for both requests and responses?
`none`	Leave the `name` property blank or use `null`.	• When the ABL parameter is an input type and there is no corresponding element in the HTTP request to map it to. Specifying `none` prompts the DOH to send an empty parameter to the ABL class method or ABL procedure. • When the ABL parameter is of an output type and you do not want to produce an HTTP message element from it.	Requests and Responses
`path`	Specify the name of the path parameter as defined in the WEB transport properties	When the ABL parameter is an input type and its value needs to be extracted from a parameter in the requested URL path. For example, assume that a client application makes a GET request to `http://localhost:8810/myapp/web/CustomerOrderService/customers/123` to retrieve customers records by customer ID (`123`). To set this up, set `path` as the msgElem `type` and `customerID` as the msgElem `name`. Edit the PAS for OpenEdge instance's `openedge.properties` file and add a handler for the path parameter in the web application's WEB transport properties section (that has a title similar to `oepas1.myapp.WEB`). Here is an example of the handler property to add: `handler2= OpenEdge.Web.DataObject.DataObjectHandler:/customers/{customerID}`	Requests only
`query`	Specify the name of the query parameter	When the ABL parameter is an input type and its value needs to be extracted from a query in the requested URL path. For example, assume that a client application makes a GET request to `http://localhost:8810/myapp/web/`	Requests only

[3] If you specify a `DATASET` as the `ablType`, you also need to define the dataset schema in the mapping file. To learn more, see the example dataset schema that follows the `msgElem` properties table below.

[4] Use this to specify temp-tables. You also need to define the temp-table schema in the mapping file. To learn more, see the example dataset schema that follows the `msgElem` properties table below.

[5] If the parameter is an ABL array, use the syntax `<Datatype> EXTENT` (for example, `CHARACTER EXTENT`).

[6] If the parameter is an ABL object, use the syntax `CLASS <full.type.name>` (for example, `CLASS Business.Customer`)

Type	Name	When to use	Use for both requests and responses?
		`CustomerOrderService?id=123` to retrieve customers records by customer ID (`123`). To set this up, set `query` as the msgElem `type` and `id` as the msgElem `name`.	
`queryMap`	Leave the `name` property blank or use `null`	When the ABL parameter is an input type and the requested URL path contains a query of the type `OpenEdge.Core.Collections.IStringStringMap`. This is mostly useful for debugging.	Requests only
`header`	The name of the HTTP header. For complex headers with parameters, use the syntax `<headername>; <parameter>`	When the ABL parameter is an input, output, or input-output type, and its value needs to be either retrieved from an HTTP request header (for input operations) or passed on to an HTTP response header (for output operations).	Requests and responses
`headerSet`	Leave the `name` property blank or use `null`	When the ABL parameter is an input, output, or input-output type and its value is an array of HTTP header objects. This is mostly useful for debugging.	Requests and responses
`field`	The name of a root-level key in a JSON or multipart message	When the ABL parameter is an input, output, or input-output type, and the value needs to be retrieved from or passed on to a JSON or multi-part message in an HTTP request or response. You can only retrieve or pass the value of a key that is at the topmost level in the message hierarchy. For example, in the following JSON message, you can only retrieve the value of `message`, `status`, `date`, or `customer`. If you need to access a 'child' element (such as `text`, `target`, `id`, or `name` in this example), you must do so from within your ABL code. `{ "message": { "text": "Order number 213 was delivered." "target": "239888" }, "status": "sent", "date": "10-APRIL-2018", "customer": { "id": "1282", "name": "John Jones" } }`	Requests and responses
`cookie`	Specify the name of the browser cookie	When the ABL parameter is an input, output, or input-output type, and its value needs to be retrieved from or passed on to a single-value browser cookie.	Requests and responses

Type	Name	When to use	Use for both requests and responses?
`statusCode`	Leave the `name` property blank or use `null`	When the ABL parameter is an output type, and the its value needs to be passed to the client application as an HTTP status code. Note that this overrides the `statusCode` property set for the HTTP verb.	Responses only
`body`	Leave the `name` property blank or use `null`	When the ABL parameter is an input, output, or input-output type, and you need to either retrieve its value from the message body of the HTTP request or pass on the value from the ABL parameter to the message body of the HTTP response. While retrieving the value for an INPUT parameter, you can set a flag to strip the envelope from the HTTP request message. Similarly, in the case of an OUTPUT parameter, you can optionally set a flag to add the envelope to the parameter output before forming the HTTP response. To set these flags, add a an `options` property under the HTTP verb in the mapping file as shown in this example: `"/getPersonalizedMessage": { "GET": { "contentType": "application/json", "options": { "writeBI": true, "requestEnvelope" :true "responseEnvelope" :true "errorEnvelope" :true },`	Requests and responses
`request`	Leave the `name` property blank or use `null`	When the ABL parameter is an input type, and you want to populate its value with the complete HTTP request message, including the headers, body, etc.	Requests only
`httpMethod`	Specify the name of the HTTP verb	When you want to perform some type of debugging by passing the HTTP method name (GET, PUT, POST, etc) to an ABL method via an INPUT parameter.	Requests only
`constant`	Specify the value that you want to pass on to the ABL parameter	When you want to populate the value of an input parameter with the value of the `name` property. This is mostly useful for testing.	Requests only

Example of a dataset schema in the mapping file

Place the dataset or temp-table schema under a `schemas` object inside your *`ServiceName`* (`HelloService` in the following example), at the bottom of the JSON mapping file. Here is an example of a schema for a dataset that contains a temp-table:

```
"services": {
 "HelloService": {
  ....
  ....
  "schemas": {
      "dsCustomer" : {
      "ttCustomer": {
       "trackChanges": true,
       "indexes" : {
                  "Comments": {
                      "primary": false,
                      "unique": false,
                      "word": false,
                      "fields": ["Comments"]
                  },
                  "CountryPost": {
                      "primary": false,
                      "unique": false,
                      "word": false,
                      "fields": ["Country", "PostalCode"]
                  },
                  "CustNum": {
                      "primary": true,
                      "unique": true,
                      "word": false,
                      "fields": ["CustNum"]
                  },
                  "Name": {
                      "primary": false,
                      "unique": false,
                      "word": false,
                      "fields": ["Name"]
                  },
                  "SalesRep": {
                      "primary": false,
                      "unique": false,
                      "word": false,
                      "fields": ["SalesRep"]
                  }
       },
       "fields": [
                    {"name": "CustNum", "ablType": "INTEGER  ", "extent": null},
                    {"name": "Country", "ablType": "CHARACTER", "extent": null},
                    {"name": "Name", "ablType": "CHARACTER", "extent": null},
                    {"name": "Address", "ablType": "CHARACTER", "extent": null},
                    {"name": "Address2", "ablType": "CHARACTER", "extent": null},
                    {"name": "City", "ablType": "CHARACTER", "extent": null},
                    {"name": "State", "ablType": "CHARACTER", "extent": null},
                    {"name": "PostalCode", "ablType": "CHARACTER", "extent": null},
                    {"name": "Contact", "ablType": "CHARACTER", "extent": null},
                    {"name": "Phone", "ablType": "CHARACTER", "extent": null},
                    {"name": "SalesRep", "ablType": "CHARACTER", "extent": null},
                    {"name": "CreditLimit", "ablType": "DECIMAL  ", "extent": null},
                    {"name": "Balance", "ablType": "DECIMAL  ", "extent": null},
                    {"name": "Terms", "ablType": "CHARACTER", "extent": null},
                    {"name": "Discount", "ablType": "INTEGER  ", "extent": null},
                    {"name": "Comments", "ablType": "CHARACTER", "extent": null},
                    {"name": "Fax", "ablType": "CHARACTER", "extent": null},
                    {"name": "EmailAddress", "ablType": "CHARACTER", "extent": null}
       ]
      }
     }
```

```
      }
   }
}
```

Develop an ABL service using the REST transport

Using Progress Developer Studio for OpenEdge (Developer Studio), you can create a REST Web application from your Progress OpenEdge ABL application. This section describes how to create a REST Web application from an existing Progress OpenEdge ABL application.

Preparing for creating a REST Web application

Before you create a REST Web application from your existing ABL application, you have to:

- Ensure that your ABL application is state-free.
- Gather information about the ABL resources that you want to expose as REST resources.

Note: A REST Web application can only use services from one OpenEdge project. You must determine the project that you want to use for creating a REST Web application.

State-free application

A REST Web application is state-free. To create a REST Web application from your existing ABL application, you must ensure that:

- For procedure files that contain internal procedures, the top-level procedure does not define parameters and state is not maintained between calls to internal procedures and functions.
- Each class is designed to run as a singleton.

Planning

You should create a checklist of the ABL resources in your application that you want to expose as REST resources. From those ABL resources, make a note of:

- The input and output parameters of ABL operations
- REST service URI naming conventions
- The HTTP verb that you will associate with each operation

Making a note of these will be helpful when you create your REST Web application in Developer Studio.

Guidelines for REST URI naming conventions

When you determine naming conventions for REST service URIs, follow these guidelines:

- URIs are case-sensitive.
- The name must start with “/”. For example, `/Customers/{custId}`.

- In a REST resource URI, do not use Java-reserved words such as int, char, or Boolean as the path or query parameter name.
- The resource name in the URI must represent the resource being operated on in the application. For example, suppose you add a resource for the Read method in Customer.cls. Since it is a customer record that is being operated on, the service relative URI should be /Customers (not /readcustomer or /read1).

Creating an ABL REST Web application

The following are the high-level tasks that you must perform for creating a REST Web application from an existing ABL application:

1. Annotate ABL resources for creating a REST service.
2. Configure a REST service by mapping REST operations to ABL operations.
3. Publish the REST service to the default Pacific Application Server for OpenEdge (PAS for OpenEdge) instance (oepas1) in your development environment.
4. Test the REST service using a REST client.

Annotating ABL resources for creating a REST service

The first step is to annotate the ABL resources in the ABL application using the **Define Service Interface** wizard.

Do the following in the project where you want to create the REST service:

1. Launch the Define Service Interface wizard for an ABL resource (class or procedure file).
2. Select the ABL operations you want to annotate to expose them as REST operations.
3. Review, and optionally edit, the generated annotations.

If you have a large application, Developer Studio enables you to annotate multiple files within your workspace at once to save time. Remember that, you should only annotate the same type of files (class files, procedure files, external procedure files) together. This ensures that the annotations include the correct execution mode (singleton, single-run, external) for each file type.

Annotate ABL resources using the Define Service Interface wizard

Follow these steps to annotate ABL resources using the Define Service Interface wizard:

1. Based on the ABL resources that you want to include in your REST service, right-click an ABL class or procedure file in the Project Explorer view.
2. Select Progress OpenEdge> Define Service Interface. The Define Service Interface wizard appears.
3. Select REST from the Definition mode drop-down list.
4. In the Project Explorer view, the ABL class or procedure file that you selected is marked in the list of procedures and classes in the workspace. The corresponding operations (functions, procedures, and methods) are listed on the right. Select the operations that you want to annotate and then click Next.The Edit Annotation page opens.
5. Review the REST annotations, and then click Finish. Notice that in the OpenEdge editor the annotations have been added to the ABL class or procedure file.

REST annotations

A REST annotation makes an ABL class or procedure file available as a REST resource.

When you use the Define Service Interface wizard to create a REST service, Developer Studio generates two types of REST annotations:

- REST main annotation
- REST detail annotation

Annotations always begin with the @ sign, so that the compiler treats them as comments. Annotations do not affect how your code runs.

REST main annotation

A REST main annotation appears at the top of a class or procedure file, preceding any code in the file.

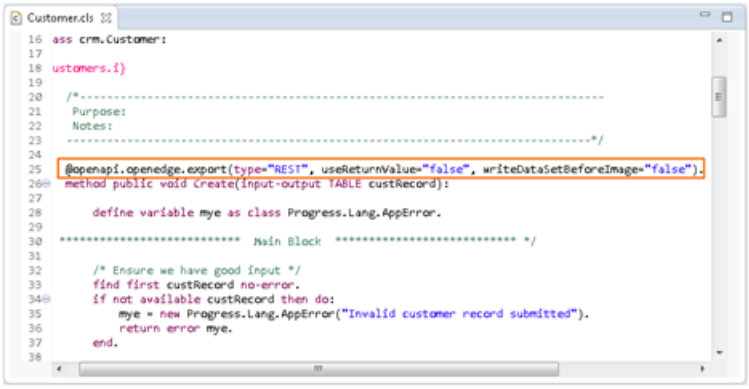
A REST main annotation contains the following attributes:

- File
- Type
- executionMode
- The value of executionMode can be:

executionMode value	Description
external	Default value for an external procedure (a file that contains no internal procedures or functions).
single-run	Default value for a procedure that contains internal procedures and/or user-defined functions. This value enables you to call internal procedures and user-defined functions in a procedure file.
singleton	Default value for a class that contains methods. This value enables you to call methods in a class file or internal procedures in a procedure file.

REST detail annotation

REST detail annotation



 © 2015 Progress Software Corporation. All rights reserved.

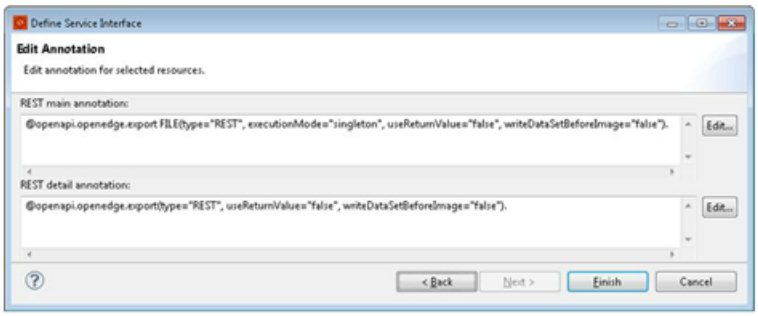
A REST detail annotation is added before each internal procedure or function in a procedure file or before each method in a class file.

A REST detail annotation contains some of the same attributes as a main annotation but, in addition, it provides details about how an internal procedure, function or method operates.

Reviewing and editing an annotation

Reviewing and editing an annotation

- The Edit Annotation page enables you to:
  - Review annotations.
  - Edit annotations to change the execution mode.



 © 2015 Progress Software Corporation. All rights reserved.

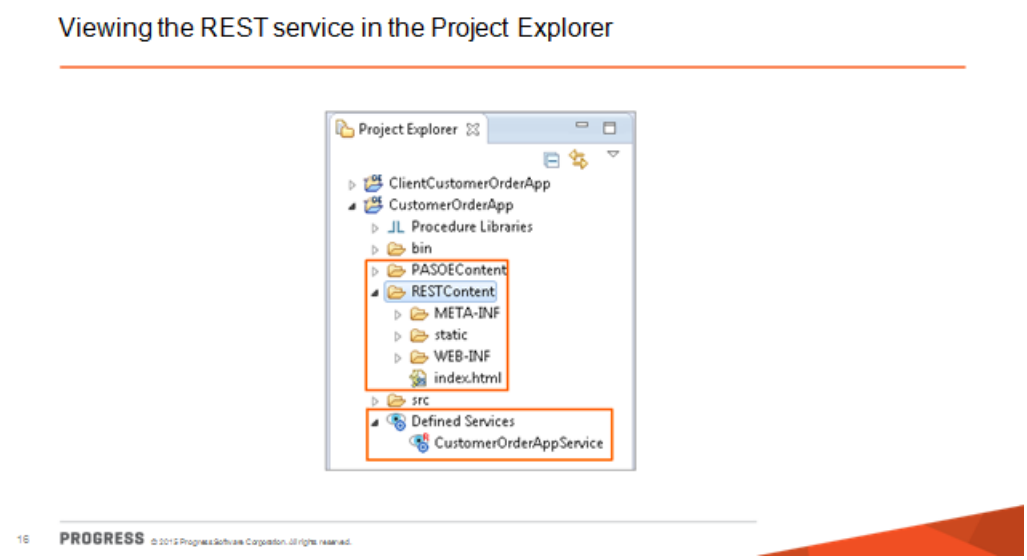
When you create a REST service using the Define Service Interface wizard, the Edit Annotation page enables you to review the annotations that are added to the selected ABL resources.

Depending on your requirements, you may want to edit the annotations to change the execution mode. For example, for a procedure with internal procedures you can set the execution mode to single-run or singleton depending on how you want to set the user context on the application server.

To edit an annotation, click the Edit button next to the annotation.

Note: You should only edit an annotation in the Edit Annotation page. Never edit the source file.

Viewing the REST service in the Project Explorer



After you annotate ABL resources using the Define Service Interface wizard, you can see that:

- The ABL project is converted to a REST project. In addition, the RESTContent and PASOEContent folders are added to the project.
- A Defined Services node with a REST service is created. The REST service is by default named “<your project name> Service”.

Note: By default, the Define Service Interface wizard saves a backup of the selected ABL resources without annotations in a default location. You can modify the location if you want.

Mapping REST operations to ABL operations

After creating a REST service using the Define Service Interface wizard, you must map REST operations to ABL operations.

To map REST operations to ABL operations:

1. Add REST resources to the defined REST service.
2. For each resource, map REST (HTTP) verbs to operations (methods, procedures, and functions) of the annotated ABL class (.cls) and/or procedure (.p) file.
3. For each REST verb, map the HTTP Request and Response parameters to the Interface parameters.

Adding REST resources to the defined REST service

Follow these steps to add REST resources to the defined REST service:

1. In the Project Explorer view, expand Define Services and then double-click a service from Defined Services to open it in the REST Resource URI Editor.
2. On the Resources toolbar, click the Add Resource icon. The New REST Resource dialog box opens.
3. In the Resource URI field, enter a URI for the resource and then click OK. This creates the REST resource and displays it in the Resources list.
4. Repeat steps 2 and 3 for all the REST resources that you want to add.

Associating REST verbs with ABL operations

All the REST resources that you add to the defined service appear under Resources. For each REST resource, you map REST verbs to ABL operations, but you can only associate each REST verb in a REST resource with one ABL operation.

Follow these steps to associate a REST verb with an ABL operation:

1. In the REST Resource URI Editor, select a REST resource from the Resources list.
2. In the Verb Association section, click the Select operation icon beside the verb of a REST resource. The Associate Operation With Verb dialog box appears with a list of ABL class or procedure files.
3. From the Resources list, select an ABL resource (procedure or class file). The ABL routine displays a list of operations in the selected ABL resource.
4. Select the ABL operation that you want to associate with the selected REST verb and then click OK.
5. The selected operation appears beside the associated verb in the Verb Association section.
6. Repeat steps 2-4 for all the operations that you want to map with the verbs for the selected resource.Repeat steps 1-6 for each resource in the Resource list.

Mapping parameters

The last step in mapping REST operations to ABL operations is mapping the parameters. A REST resource verb has two distinct parameter mappings:

- Request mapping
- Response mapping

For each REST verb, you must map sections of the REST request with ABL input parameters and sections of the REST response with ABL output parameters.

In your REST service, you can map interface parameters to any of the sections present in an HTTP request or response such as:

- Header
- Body
- Cookie
- Query string parameters
- Path parameter
- Form parameter

To create a mapping rule, click on a section under Request or Response and drag a line to the required Interface parameter (ABL input or output parameter).

Block line

Mapping request parameters

In the REST Expose Editor, in the Input tab, you map input parameters from the HTTP request to the ABL operation.

When you map a request parameter, ensure that you map a section of the HTTP request that contains input in the format expected by the procedure, method, or function. For example, if the input parameter of a procedure requires an integer from the path parameter of your URI, then the path parameter must be an integer.

Remember that you will not receive any alerts while mapping. However, at runtime if you pass the wrong parameter type, you will receive an error.

Mapping response parameters

In the REST Expose Editor, in the Output tab, you map output parameters from an ABL operation to an HTTP response.

An HTTP response consists of:

- A response status line that contains response code
- A response header that contain headers and cookies
- The response body

You can map the output of your ABL procedure, method, or function to any section of the HTTP response.

Develop an ABL service using the SOAP transport

The first step in creating a Web service is considering the requirements of the application and its intended clients. Knowing these requirements can decide certain details of a Web service design and implementation. These requirements include the session model, the SOAP message format, and how you group operations into Open Client objects.

Security considerations also have a major impact on how you expose your application. For more information on security options for managing WSA configuration and administration and on managing access control for Soap-based Web services, see *OpenEdge Application Server: Administration*. For more information on how OpenEdge supports security for OpenEdge Web services, see *OpenEdge Getting Started: Core Business Services - Security and Auditing.*

Web service objects and the WSDL file

During Web service deployment, the WSA generates a WSDL file that defines one or more *Web service objects* to describe the Web service. These Web service objects conform to the Open Client object model. There must be a single AppObject and, if appropriate, there might also be SubAppObjects and ProcObjects. The Web service objects include all operations—procedures, user-defined functions, and built-in operations for object management—that define the Web service interface. The WSDL file might also define an object identifier (*object ID*) for each Open Client object. The WSDL defines this object ID for most Open Client object types, depending on the session model.

For more information on the Open Client object model and its architecture, see *OpenEdge Development: Open Client Introduction and Programming*. For more information on how the Open Client object model supports OpenEdge SOAP Web services, see Building Clients for OpenEdge SOAP Web services.

Object representation

WSDL does not have an inherent concept of objects for encapsulating operations. However, OpenEdge Web services follow common usage in representing each Web service object within a unique namespace of the WSDL file. Typically, interface generators code each such object defined in the WSDL file as a corresponding object in the client interface. However, the interface generator on a particular client platform ultimately determines how each WSDL object is represented in the client development language.

Object binding

In addition to defining the Web service interface, the WSDL file specifies how this interface is bound to the application server executing the operations. For OpenEdge Web services, this binding information specifies how to generate and handle SOAP messages exchanged over HTTP/S between the client and the WSA instance managing the Web service. This includes information to identify the Web service and its WSA instance, as well as to format and send the SOAP messages. The Web service deployer specifies this binding and implementation information (*deployment information*) during Web service deployment.

Deployment information

This deployment information consists of the data stored in WSDL elements as follows:

- **Web service namespace** — A namespace used to uniquely identify the Web service to the WSA instance where the Web service is deployed. In the WSDL, this value is specified by the `targetNamespace` attribute of the `<definitions>` element. It is typically specified as a Universal Resource Identifier (URI), usually in the form of a Universal Resource Locator (URL) or Universal Resource Name (URN). The value can be specified in ProxyGen, during Web service definition, or in OpenEdge Management or OpenEdge Explorer, during Web service deployment.
- **WSA URL** — The URL for the WSA instance where the Web service is deployed, and that forms the root for all URLs that access the Web services it manages. In the WSDL, this is specified by the `location` attribute of the `<soap:address>` element within the `<services>` element.
- **SOAP action** — A string (if specified) that the client application must place in the SOAPAction HTTP header when it invokes operations on the Web service. In the WSDL, this value is specified using the `soapAction` attribute of the `<soap:operation>` element, and the value is included in the SOAPAction header of all HTTP messages that transport SOAP messages for the Web service.

  The SOAPAction HTTP header is normally required for all HTTP messages that carry SOAP messages and is used by intervening security servers (such as firewalls) to determine if each HTTP message is allowed to pass through to its destination. The default is a blank string, but can be any string specified at deployment that is required by the intervening security servers on the network.
- **SOAP format** — Two string values that specify the format of SOAP messages exchanged between the client and the Web service. In the WSDL, these are specified by the `style` attribute in the `<soap:binding>` and `<soap:operation>` elements, and by the `use` attribute in the `<soap:header>`, `<soap:body>`, and `<soap:fault>` elements. While these values are specified in many places within the WSDL, the same style and use values are used everywhere in the file, ensuring that all SOAP messages generated for the Web service use the same SOAP format. For more information on SOAP formats, see SOAP message formats.

For more information on specifying this deployment information in a WSDL, see *OpenEdge Development: Open Client Introduction and Programming* for the optional WSDL typically used for development testing, and see the Web service deployment chapters in *OpenEdge Application Server: Administration* for the WSDL deployed to a WSA instance. For more information on how the WSDL file specifies this information, see WSDL element overview for OpenEdge Web services.

Defining and deploying a Web service definition

After you have built and compiled the ABL r-code for the AppServer application, you use ProxyGen to assemble a definition of your Web service. That definition enables you to define WSA instances to host your Web service and to generate a WSDL file that end users can use to build clients for your Web service.

Creating Web service definitions follows the Open Client model. For instructions on creating Web service definitions, see the chapter on generating proxies and Web service definitions in *OpenEdge Development: Open Client Introduction and Programming*. For more information on working with ProxyGen, see the online help.

Once you have a definition for your OpenEdge Web service, you need to deploy it to a Web Service Adapter (WSA) instance for testing. Building Clients for OpenEdge SOAP Web services discusses how to create clients to test your Web service.

Configuring a Web Service Adapter instance

The WSA is a Java servlet running on a Web server or stand-alone Java Servlet Engine (JSE). Situated between the client and AppServer, this Java servlet understands how to communicate at run time with:

- A Web service client using SOAP messages
- An AppServer using the ABL interface to the application service

The WSA generates the WSDL file needed to design a client for your Web service. The WSA maintains the status of all OpenEdge Web services that have been deployed to its Web application context. It also manages the exchange of Web service SOAP requests and generates SOAP responses between Web service clients and AppServers at run time.

Each WSA instance can support multiple Web services. But, you can define multiple WSA server instances (Java servlets), each with different properties. For example, you might create separate WSA instances to deploy the same Web service using different SOAP message formats. For more information on WSA configuration and administration and on WSA deployment and Web service run-time support, see *OpenEdge Application Server: Administration*.

At any time in the cycle of Web service design and development, you can create and configure a WSA instance on which to deploy your Web service. When you do, think carefully about security requirements for the WSA instance and the Web services you plan to deploy to it.

Creating and configuring a WSA instance is a two-step process. You must configure the WSA servlet and the JSE that executes the WSA.You first use either OpenEdge Management and OpenEdge Explorer or the `mergeprop` utility to define and configure the WSA instance. While OpenEdge Management and OpenEdge Explorer supports most WSA configuration options, you must use the `mergeprop` utility to edit some WSA configuration properties in the local `ubroker.properties` file, particularly those that define WSA administration roles and their associated access permissions.

You can use OpenEdge Management and OpenEdge Explorer or the `WSAMAN` utility to temporarily change a selected set of WSA instance properties at run time (WSA Run-time Properties). This can be helpful for debugging or testing certain settings before making them permanent in your WSA configuration.

After configuring the WSA instance, you use the appropriate tools to define the WSA instance as a Java servlet to your JSE. After completing the creation and configuration of a WSA instance, you might have to restart the JSE to start up the WSA instance before deploying Web services to it.

For specific information on creating and configuring WSA instances, see *OpenEdge Application Server: Administration*. For information on using the `mergeprop` utility, see *OpenEdge Getting Started: Installation and Configuration*.

Installing WSA on a DMZ server

A *DMZ server* is a server that might not allow you to open untrusted network ports. Since the AdminServer uses ports which might be interpreted as untrusted, the standard WSA configuration might run on such a server. The WSA supports a "remote" configuration to work around this issue. The remote configuration installs only enough of OpenEdge on the remote server to run the WSA. This installation includes the WSA Java Servlet and the files used to support its local configuration.

The remote WSA uses a `ubroker.properties` file with a limited set of static startup parameters. You must manually edit this file to change the WSA instance's static configuration properties.

You can continue to administer the remote WSA's dynamic run-time properties and deploy and manage SOAP services by mapping it to an AdminServer on the Intranet behind the DMZ server. The Intranet AdminServer uses HTTP/S to pass WSA administration operations as SOAP messages to the WSA's Administration service.

To run a remote WSA configuration:

1. Install the remote WSA on the DMZ server. Manually edit the static startup parameters in its `ubroker.properties` file and install the WSA as a Web application in the local Web server's JSE.
2. Start the WSA the Web server's JSE and verify that a browser can access its URL from the Internet.
3. Connect to an AdminServer on the Intranet and configure a "remote" WSA instance using OpenEdge Management and OpenEdge Explorer or the `WSAMAN` utility by specifying the remote WSA's URL.

   The Intranet AdminServer creates a WSA mapping entry in its local `ubroker.properties` file with the URL of the remote WSA's administration SOAP service.

4. Use OpenEdge Management and OpenEdge Explorer or the `WSAMAN` utility to manage the "remote" WSA's dynamic run-time properties and deploy and manage the WSA's SOAP services.

   OpenEdge Management and OpenEdge Explorer connects to the Intranet AdminServer where you mapped the WSA and then connects to the remote WSA's SOAP administration service. When OpenEdge Management and OpenEdge Explorer sends an operation to the Intranet AdminServer, the AdminServer turns it into a SOAP message and forwards it to the WSA's SOAP administration service which executes the operation.

Sample Code with SOAP Messages for OpenEdge Web Services

The following samples provide individual examples of how SOAP messages are sent between the client and an OpenEdge SOAP Web service, and code that produces some of these messages. The code samples are written in VB.NET or C#.NET.

Note: You can find some of the sample applications in *OpenEdge_install_dir*`/src/samples/webservices`.

Sample Web service specifications

These examples rely on sample Web service specifications shown in the following table.

Sample Web service specifications

Property or component	Configuration, value, or name
Session model	Managed **Note:** Some information and examples are also provided to show session-free interactions.
SOAP format	`Doc/Lit`
TargetNamespace	`"urn:OrderSvc"`
AppObject	`OrderInfo`
ProcObject	`CustomerOrder`

Some of the information presented in these examples provides a review of concepts introduced and described in previous chapters of this manual. Note that some SOAP messages and code samples are edited for brevity.

Client references to the sample VB.NET interface include the following method declarations for these interface objects:

- Interface AppObject, `OrderInfo`:

VB.NET method declarations on AppObject, OrderInfo

```
Public Sub Connect_OrderInfo(
       ByVal userId As String,
       ByVal password As String,
       ByVal appServerInfo As String)
Public Function FindCustomerByNum(
       ByVal CustomerNumber As Integer,
       ByRef CustomerName As String) As String
Public Function CreatePO_CustomerOrder(
       ByVal custNum As Integer) As String
Public Sub Release_OrderInfo( )
```

The `FindCustomerByNum` and `CreatePO_CustomerOrder` methods are represented as functions because methods in Doc/Lit WSDLs are always defined to return a value, regardless of whether the procedure defined in ProxyGen is specified to return the ABL `RETURN-VALUE`. If **Return ABL RETURN-VALUE** is not specified, the return value for the function will be a null String. For more information, see *OpenEdge Development: Open Client Introduction and Programming.*

VB.NET method declarations on ProcObject, CustomerOrder

```
Public Function GetOrderDetails(
       ByRef OrderDetails( ) As OrderDetailsRow) As String
Public Function GetTotalOrdersByNumber(
       ByVal Threshold As Decimal) As Integer
Public Sub Release_CustomerOrder( )
```

Note: ProxyGen normally configures a `Connect_AppObject( )` method with three parameters (`string userid`, `string password`, and `string appServerInfo`). Enabling the **Return ABL RETURN-VALUE on Connect** option adds the output parameter `string result`, which contains the return value of the Connection procedure.

Consuming a session-managed Web service

To access a session-managed Web service, you maintain connection context by passing object ID's associated with each request.

To begin using a session-managed Web service:

1. Instantiate (create) the AppObject as appropriate for the client platform.
2. Connect to an AppServer by calling the connect method on the AppObject before calling any other Web Service (AppObject) method.
3. Obtain the AppObject ID value from the SOAP response header for the connect method and use it for all subsequent calls to methods on the session-managed AppObject.
4. Invoke any available methods on the AppObject, as required.
5. Ensure that the last method you invoke on the AppObject is the object's release method. For more information, see Releasing an object on page 57.

As with other Open Clients, there is no ABL involved in implementing the connect method on an AppObject. For Web services, however, the operation is an object method that is required by the WSA.

This is a VB.NET declaration for an AppObject connect method, `Connect_OrderInfo( )`:

VB.NET prototype for an AppObject Connect_Object method

```
Public Sub Connect_OrderInfo(
              ByVal userId As String,
              ByVal password As String,
              ByVal appServerInfo As String)
```

Note: For more information on the parameters to this method, see the sections on connecting to an AppServer in *OpenEdge Development: Open Client Introduction and Programming*. Note that there is no AppServer URL parameter. For a Web service (unlike other Open Client applications), the deployer manages the AppServer connection information for each Web service through the WSA instance where the Web service is deployed. Each deployed Web service has writable properties to specify this information. For more information, see the sections on Web service deployment in *OpenEdge Application Server: Administration*.

When the client executes the connect method, the SOAP response message contains a SOAP header with the AppObject ID. You must send this AppObject ID in the SOAP header of the request message for every subsequent method call on the AppObject. How you handle the SOAP header and AppObject ID depends on your client type. For some clients, such as .NET, this process is automated. For other clients, such as ABL and Java, you need to create the code to handle this yourself. For more information on handling SOAP headers and object IDs, see Handling SOAP Message Headers in ABL and Developing a Java Client to Consume OpenEdge SOAP Web Services.

This is a sample instantiation and invocation of the connect method on the `OrderInfo` AppObject:

VB.NET client code for the connect method on AppObject, OrderInfo

```
webService = New OrderSvc.OrderInfoObj( )
webService.Connect_OrderInfo( "", "", "" )
```

This is a sample Doc/Lit SOAP request message that might be generated from invoking the `Connect_OrderInfo( )` method, as in the example:

SOAP connect request for AppObject, OrderInfo

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope namespaces defined here...>
  <soap:Body>
    <Connect_OrderInfo xmlns="urn:OrderSvc:OrderInfo">
      <userId />
         <password />
         <appServerInfo />
         </Connect_OrderInfo>
  </soap:Body>
</soap:Envelope>
```

Note the data for the request highlighted in the example for parameters passed as empty strings.

This is a sample Doc/Lit SOAP response that might be generated by the WSA as a response to the `Connect_OrderInfo( )` method:

SOAP response to OrderInfo AppObject connect request

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope namespaces defined here...>
<soap:Header>
  <OrderInfoID xmlns="urn:OrderSvc:OrderInfo">
    <UUID>2e62cab6b81150d5:167f64e:f295e997b0:-8000;
          <OrderInfo|PX-000001|AO>;M/IryPm3piDcF/W5DsH4GA==</UUID>
  </OrderInfoID>
</soap:Header>
<soap:Body>
  <Connect_OrderInfoResponse xmlns="urn:OrderSvc:OrderInfo" />
</soap:Body>
</soap:Envelope>
```

Note the value for the returned AppObject ID, `OrderInfo`, highlighted in the example. Thus, the SOAP response header returns the following AppObject ID contained in the `OrderInfoID` element:

```
2e62cab6b81150d5:167f64e:f295e997b0:-8000;<OrderInfo|PX-000001|AO>;M/IryPm3p
iDcF/W5DsH4GA==
```

Consuming a session-free Web service

Because the AppObject for a session-free Web service maintains no connection to the Web service, you can call the methods of a session-free AppObject with no preparation. When the WSA instance receives such a method request, it simply locates an available AppServer connection in the connection pool for that Web service and invokes the corresponding procedure or function on the available AppServer.

Note: All AppServers that participate in the connection pool for a session-free Web service are assumed to share the same version and capabilities and have access to the same set of databases and other shareable resources on a network. For more information, see *OpenEdge Application Server: Developing AppServer Applications.*

To use a session-free Web service:

Instantiate (create) the AppObject as appropriate for the client platform. For example:

VB.NET client code to instantiate session-free AppObject, OrderInfo

```
webService = New OrderSvc.OrderInfoObj( )
```

Invoke any available methods on the AppObject, as required.

Running an ABL non-persistent procedure

Methods that run a non-persistent procedure can:

- Appear on any AppObject
- Require an object ID in the SOAP request header for each method invocation unless the object is a session-free AppObject

This is the ABL prototype for the sample `FindCustomerByNum( )` method:

ABL prototype for a non-persistent external procedure

```
/* FindCustomerByNum.p */

DEFINE INPUT PARAMETER CustomerNumber AS INTEGER.
DEFINE OUTPUT PARAMETER CustomerName AS CHARACTER.
```

This is a sample VB.NET declaration for the ABL non-persistent procedure method, `FindCustomerByNum( )`:

VB.NET prototype for a method that runs an ABL non-persistent procedure

```
Public Function FindCustomerByNum(
       ByVal CustomerNumber As Integer,
       ByRef CustomerName As String) As String
```

This is a sample call to this method:

VB.NET client code calling an external procedure method

```
Dim CustomerName As String
Dim retVal As String
retVal=webService.FindCustomerByNum(3, CustomerName)
```

This is a sample Doc/Lit SOAP request that might be generated from this `FindCustomerByNum( )` method invocation:

SOAP request from calling the FindCustomerByNum procedure method

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope namespaces defined here...>
  <soap:Header>
    <OrderInfoID xmlns="urn:OrderSvc:OrderInfo">
      <UUID>2e62cab6b81150d5:167f64e:f295e997b0:-8000;
            <OrderInfo|PX-000001|AO>;M/IryPm3piDcF/W5DsH4GA==</UUID>
    </OrderInfoID>
  <soap:Header>
  <soap:Body>
    <FindCustomerByNum xmlns="urn:OrderSvc:OrderInfo">
      <CustomerNumber>3</CustomerNumber>
    </FindCustomerByNum>
  </soap:Body>
</soap:Envelope>
```

Note the following elements in the preceding SOAP message:

- AppObject ID (`OrderInfoID`) sent using the `<UUID>` element of the SOAP header
- Data for the request highlighted in the example, including the request for a `CustomerNumber` value of 3

This is a sample Doc/Lit SOAP response that is generated by the WSA from this invocation of the `FindCustomerByNum( )` method:

SOAP response from calling the FindCustomerByNum method

```
<?xml version="1.0" encoding="UTF-8" ?>
<soap:Envelope namespaces defined here...>
  <soap:Body>
    <FindCustomerByNumResponse xmlns="urn:OrderSvc:OrderInfo">
      <result xsi:nil="true" />
      <CustomerName>Hoops</CustomerName>
    </FindCustomerByNumResponse>
  </soap:Body>
</soap:Envelope>
```

Note: The CustomerName value, `"Hoops"`, returned and highlighted in the example.

Creating and using a ProcObject

Each AppObject that defines ProcObjects also defines a built-in class factory method specifically for each ProcObject. This method runs a persistent procedure on the AppServer that thereby establishes a persistent AppServer session context that is encapsulated by the ProcObject for access by the client.

To create and use a ProcObject:

1. Instantiate (create) the ProcObject as appropriate for the client platform.
2. Call the class factory method for this ProcObject on the parent AppObject or SubAppObject to run the persistent procedure on the AppServer.
3. Obtain the ProcObject ID value from the response header of the class factory method. Use this ProcObject ID for all subsequent calls to methods on the ProcObject.
4. Invoke any available methods on the ProcObject.
5. Ensure that the last method you invoke on the ProcObject is the object's release method. For more information, see Releasing an object on page 57.

Note: You can instantiate a ProcObject either before or after calling the class factory method and getting the ProcObject ID, but you must complete all of these steps before you invoke methods on the ProcObject.

ProcObject session context

For a session-managed ProcObject, the method establishes the persistent session context using the same AppServer connection as the parent AppObject. For a session-free ProcObject, the method establishes the persistent session context on any AppServer that the WSA makes available through the Web service connection pool, and the client connection to that context persists until the ProcObject is released. Note that for multiple ProcObjects defined for the same session-free Web service, no two objects ever share the same session context. While for a session-managed Web service, all defined ProcObjects always share the same session context.

ProcObject IDs

You use the same type of class factory method to create ProcObjects for both session-managed and session-free Web Services. For a ProcObject whose defining parent is a SubAppObject or a session-managed AppObject, you must send the object ID of the parent in the SOAP request message header. However, for a ProcObject whose defining parent is a session-free AppObject, you do not have an AppObject ID to send in the SOAP request header when you invoke the class factory method to create the ProcObject.

For every ProcObject, regardless of its parent, the SOAP response header for the class factory method returns a ProcObject ID that associates the ProcObject session context with the client. You must send this ProcObject ID in the SOAP request header for all other method calls on the ProcObject.

ProcObject class factory methods

This is the ABL prototype for the persistent procedure that runs for the sample ProcObject class factory method, `CreatePO_CustomerOrder( )`:

ABL prototype for a persistent procedure to implement a ProcObject

```
/* CustomerOrder.p */

DEFINE INPUT PARAMETER custNum AS INTEGER.
```

Note: The parameter list for the persistent procedure that runs for the ProcObject class factory method is AppServer application dependent, and is the basis for creating the parameter list of the ProcObject class factory method. A persistent procedure can also be specified in ProxyGen to return a string value using the ABL `RETURN` statement.

This is a VB.NET declaration for the ProcObject class factory method, `CreatePO_CustomerOrder( )`:

VB.NET prototype for the ProcObject CreatePO_CustomerOrder method

```
Public Function CreatePO_CustomerOrder
       (ByVal custNum As Integer) As String
```

Note: This method maps to a persistent procedure that has been specified to return the string from the ABL `RETURN-VALUE` function.

The following code snippet:

1. Instantiates the ProcObject on the client, enabling access to its methods.
2. Calls the CreatePO_CustomerOrder( ) method on the AppObject, `webService`, to create the ProcObject, CustomerOrder, on the WSA and run `CustomerOrder.p` persistently.
3. Copies the ProcObject ID to the ProcObject from the AppObject (`webService`) that creates the ProcObject.

VB.NET client code to create the ProcObject, CustomerOrder

```
custOrder = New OrderSvc.CustomerOrderObj( )
custName = webService.CreatePO_CustomerOrder(3)
custOrder.CustomerOrderIDValue = webService.CustomerOrderIDValue
```

This is a Doc/Lit SOAP request generated by invoking the `CreatePO_CustomerOrder( )` method to create the ProcObject, passing in a `custNum` value of 3:

SOAP request to create ProcObject, CustomerOrder

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope namespaces defined here...>
  <soap:Header>
    <OrderInfoID xmlns="urn:OrderSvc:OrderInfo">
      <UUID>2e62cab6b81150d5:167f64e:f295e997b0:-8000;
            <OrderInfo|PX-000001|AO>;M/IryPm3piDcF/W5DsH4GA==</UUID>
    </OrderInfoID>
  </soap:Header>
  <soap:Body>
    <CreatePO_CustomerOrder xmlns="urn:OrderSvc:OrderInfo">
      <custNum>3</custNum>
    </CreatePO_CustomerOrder>
  </soap:Body>
</soap:Envelope>
```

Note the value for the request highlighted in the example, especially the AppObject ID sent for the AppObject, `OrderInfo`, which is the parent of the ProcObject being created for customer number 3.

This is a sample Doc/Lit SOAP response that is generated by the WSA from this invocation of the `CreatePO_CustomerOrder( )` method:

SOAP response from creating ProcObject, CustomerOrder

```
<?xml version="1.0" encoding="UTF-8" ?>
<soap:Envelope namespaces defined here...>
  <soap:Header>
    <CustomerOrderID xmlns="urn:OrderSvc:CustomerOrder">
      <UUID>2e62cab6b81150d5:167f64e:f295e997b0:-8000;
            <OrderInfo|PX-000002|PO>;G1Vc/vmohvLnwxQQXwA6Cg==</UUID>
    </CustomerOrderID>  </soap:Header>
  <soap:Body>
    <CreatePO_CustomerOrderResponse xmlns="urn:OrderSvc:OrderInfo">
      <result>Hoops</result>
    </CreatePO_CustomerOrderResponse>
  </soap:Body>
</soap:Envelope>
```

Note the value returned for the `CustomerOrder` ProcObject ID highlighted in the example. The Web service returns this ProcObject ID even if it is session free, to allow the ProcObject to access its own AppServer session context.

Thus, the SOAP response header returns the following ProcObject ID contained in the `CustomerOrderID` element:

```
2e62cab6b81150d5:167f64e:f295e997b0:-8000;<OrderInfo|PX-000002|PO>;G1Vc/vmoh
vLnwxQQXwA6Cg==
```

Finally, note the ABL `RETURN-VALUE` result, returned from running the persistent procedure, which returns the customer name, `"Hoops"`.

Running an internal procedure or user-defined function

Methods to run an ABL internal procedure and user-defined function of a persistent procedure (ProcObject) are indistinguishable in client code. These methods:

- Run only in the context of a defining ProcObject
- Require a ProcObject ID in the SOAP request header, which is the object ID of the defining ProcObject

This is the ABL prototype for the sample user-defined function, `GetTotalOrdersByNumber( )`:

ABL prototype for a user-defined function

```
/* CustomerOrder.p */

FUNCTION GetTotalOrdersByNumber RETURNS INTEGER
   (Threshold AS DECIMAL):
```

This is a VB.NET declaration for the ABL user-defined function method, `GetTotalOrdersByNumber( )`:

VB.NET prototype for an ABL user-defined function method

```
Public Function GetTotalOrdersByNumber
                   (ByVal Threshold As Decimal) As Integer
```

The following is a sample method call for the user-defined function method, `GetTotalOrdersByNumber`, which is an interface method on the sample ProcObject, `CustomerOrder`:

VB.NET client code of method to run an ABL user-defined function

```
totNumber = custOrder.GetTotalOrdersByNumber(2150.99)
```

Note that user-defined function methods return a value whose data type maps to the ABL data type of the user-defined function's return value.

This is a sample Doc/Lit SOAP request that might be generated from invoking the `GetTotalOrdersByNumber( )` method to execute the ABL user-defined function, passing in a `Threshold` order value of 2150.99:

SOAP request for user-defined function method, GetTotalOrdersByNumber

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope namespaces defined here...>
  <soap:Header>
    <CustomerOrderID xmlns="urn:OrderSvc:CustomerOrder">
         <UUID>2e62cab6b81150d5:167f64e:f295e997b0:-8000;
               <OrderInfo|PX-000002|PO>;G1Vc/vmohvLnwxQQXwA6Cg==</UUID>
    </CustomerOrderID>
  </soap:Header>
  <soap:Body>
    <GetTotalOrdersByNumber xmlns="OrderSvc:CustomerOrder">
      <Threshold>2150.99</Threshold>
    </GetTotalOrdersByNumber>
  </soap:Body>
</soap:Envelope>
```

Note the object ID for the ProcObject, `CustomerOrder`, sent to make the request on a method of the ProcObject.

This is the SOAP response returning a function value of `5`, which is the total number of orders that satisfy the specified order `Threshold` value:

SOAP response from calling the GetTotalOrdersByNumber method

```
<?xml version="1.0" encoding="UTF-8" ?>
<soap:Envelope namespaces defined here...>
  <soap:Body>
    <GetTotalOrdersByNumberResponse xmlns="urn:OrderSvc:CustomerOrder">
      <result>5</result>
    </GetTotalOrdersByNumberResponse>
  </soap:Body>
</soap:Envelope>
```

Releasing an object

When you are finished using any Web service object (except a session-free AppObject), you must invoke the built-in release method to release that object and return its resources for use by other clients.

As with other Open Clients, there is no ABL involved in implementing the release method on an Open Client object. However, for Web services, the operation is an object method is required by the WSA.

This is a VB.NET declaration for the AppObject release methods:

VB.NET prototype for a Release_Object method

```
Public Sub Release_OrderInfo( )
Public Sub Release_CustomerOrder( )
```

Although these methods take no parameters, you must remember to send the object ID for the object you are releasing in the SOAP header of the release request message. How you handle the SOAP header and object ID depends on your client type. For some clients, such as .NET, this process is automated. For other clients, such as ABL and Java, you need to create the code to handle this yourself. For more information on handling SOAP headers and object IDs, see Handling SOAP Message Headers in ABL and Developing a Java Client to Consume OpenEdge SOAP Web Services.

These are sample calls to the release methods declared for the sample Web service:

VB.NET client code calling methods to release objects

```
custOrder.Release_CustomerOrder( )
webService.Release_OrderInfo( )
```

Note: AppServer connections used by session-free methods are usually returned to the Web service connection pool after the SOAP response message for the release method is returned.

Passing static and dynamic temp-table parameters

The examples in the following sections show how parameters are passed for static and dynamic temp-tables.

Invoking a method with a TABLE parameter

To invoke a method that passes a `TABLE` parameter, the client application typically must:

- Create rows using the interface object that represents the `TABLE` row
- Insert the rows in an array that represents the `TABLE` parameter

This is the ABL prototype for a sample method, `staticTT( )`, that passes a `TABLE` parameter:

ABL prototype that passes a TABLE parameter

```
/* staticTT.p */

DEFINE INPUT-OUTPUT PARAMETER TABLE FOR ttEmp.
```

This is the declaration for a VB.NET client interface method, `staticTT( )`, which has a `TABLE` parameter, `ttEmp`, passed as the class, `staticTT_ttEmpRow`:

VB.NET declaration of interface method passing static TEMP-TABLE, ttEmp

```
Public Class staticTT_ttEmpRow    Public Name As String
    Public Number As Integer
End Class

Public Sub staticTT(ByRef ttEmp( ) As staticTT_ttEmpRow)
```

The following client code defines a two row array (`myTempTable`) according to a defined schema (`staticTT_ttEmpRow`), then defines and creates two rows assigned with values. It then initializes the array (as `TABLE`, `ttEmp`) with the two rows and passes it to the `staticTT( )` interface method, which passes the `TABLE` to the Web service:

VB.NET client code passing TABLE, ttEmp, to an interface method

```
Dim myTempTable(1) as webService.staticTT_ttEmpRow
Dim myRow1 as webService.staticTT_ttEmpRow =
                            New webService.staticTT_ttEmpRow( )
Dim myRow2 as webService.staticTT_ttEmpRow =
                            New webService.staticTT_ttEmpRow( )

myRow1.Name = "Fred"
myRow1.Number = 1
myRow2.Name = "Barney"
myRow2.Number = 2

myTempTable(0) = myRow1
myTempTable(1) = myRow2

webService.staticTT(myTempTable)
```

The following is a Doc/Lit SOAP message that this call to `staticTT( )` might send:

Doc/Lit SOAP request to pass a TABLE parameter

```
<?xml version="1.0" encoding="utf-8" ?>
  <soap:Envelope namespaces defined here...>
    <soap:Header>
      <EmployeeID xmlns="urn:EmployeeSrvc:Employee">
        <UUID>2e62cab6b81150d5:-1380e8e:f27fb934f4:
        -8000;<Employee|PX-000004|AO>;5REiR9inxXQi4s6ghRWkfg==</UUID>
      </EmployeeID>
    </soap:Header>
    <soap:Body>
      <staticTT xmlns="urn:EmployeeSrvc:Employee">
        <ttEmp>
          <ttEmpRow>
            <Name>Fred</Name>
            <Number>1</Number>
          </ttEmpRow>
          <ttEmpRow>
            <Name>Barney</Name>
            <Number>2</Number>
          </ttEmpRow>
        </ttEmp>
      </staticTT>
    </soap:Body>
  </soap:Envelope>
```

Note: The `staticTT( )` method must send the object ID for the `Employee` AppObject in the SOAP request header because `staticTT( )` is a method on the session-managed AppObject.

As a point of comparison, the following is a RPC/Encoded SOAP message that this call to `staticTT( )` might send:

RPC/Encoded SOAP request to pass a TABLE parameter

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope namespaces defined here...>
  <soap:Header>
    <q1:EmployeeID id="h_id1" xmlns:q1="urn:EmployeeSrvc:Employee">
      <UUID xsi:type="xsd:string">2e62cab6b81150d5:-1380e8e:f27fb934f4:
      -8000;<Employee|PX-000004|AO>;5REiR9inxXQi4s6ghRWkfg==</UUID>
    </q1:EmployeeID>
  </soap:Header>
  <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <q1:staticTT xmlns:q1="urn:EmployeeSrvc:Employee">
      <ttEmp href="#id1" />
    </q1:staticTT>
    <soapenc:Array id="id1" xmlns:q2="urn:EmployeeSrvc:Employee"
          soapenc:arrayType="q2:staticTT_ttEmpRow[2]">
      <Item href="#id2" />
      <Item href="#id3" />
    </soapenc:Array>
    <q3:staticTT_ttEmpRow id="id2" xsi:type="q3:staticTT_ttEmpRow"
          xmlns:q3="urn:EmployeeSrvc:Employee">
      <Name xsi:type="xsd:string">Fred</Name>
      <Number xsi:type="xsd:int">1</Number>
    </q3:staticTT_ttEmpRow>
     <q4:staticTT_ttEmpRow id="id3" xsi:type="q4:staticTT_ttEmpRow"
          xmlns:q4="urn:EmployeeSrvc:Employee">
      <Name xsi:type="xsd:string">Barney</Name>
      <Number xsi:type="xsd:int">2</Number>
    </q4:staticTT_ttEmpRow>
  </soap:Body>
</soap:Envelope>
```

Note: The `staticTT( )` method must send the object ID for the `Employee` AppObject in the SOAP request header because `staticTT( )` is a method on the session-managed AppObject.

Invoking a method with a TABLE-HANDLE parameter

To invoke a method that passes a `TABLE-HANDLE` parameter:

- The client application must create and send an XML Schema along with the data to fully describe the dynamic temp-table in the SOAP request message. The `TABLE-HANDLE` parameter in a SOAP request or response consists of an XML `<DataSet>` element containing two child elements:
  - An XML Schema element, `<schema>`, representing the schema for the `TABLE-HANDLE`
  - An XML representation of data using an element, `<Data>`, with each row represented as a sub-element, `<Item>`
- A non-ABL client application must parse the XML Schema and data from the SOAP response message to make the `TABLE-HANDLE` accessible as native data within the application.

This is the ABL prototype for a sample method, `dynttIO( )`, that passes a `TABLE-HANDLE` parameter:

ABL prototype that passes a TABLE-HANDLE parameter

```
/* dynttIO.p */

DEFINE INPUT-OUTPUT PARAMETER TABLE-HANDLE ttHandle.
```

This is the declaration for a VB.NET client interface method (`dynTTIO( )`) which has a `TABLE-HANDLE` parameter, `ttHandle`, as a VB.NET object:

VB.NET declaration of interface method passing TABLE-HANDLE, ttHandle

```
Public Sub dynttIO(ByRef ttHandle As Object)
```

The following VB.NET client code passes the dyntt object representing a `TABLE-HANDLE` to the `dynttIO( )` method:

VB.NET client code passing TABLE-HANDLE, ttHandle, to interface method

```
Dim dyntt as Object

'... Code to build up the dyntt Object (XML Schema and data)

webService.dynttIO(dyntt)
```

For more information on how you might manage `TABLE-HANDLE` parameters in VB.NET, see the subsections on handling `TABLE-HANDLE` parameters in Developing a .NET Client to Consume OpenEdge SOAP Web Services and in Java see the subsections on handling `TABLE-HANDLE` parameters in Developing a Java Client to Consume OpenEdge SOAP Web Services.

This is the structure of the Doc/Lit SOAP request message that the sample `dynttIO` method sends to pass a dynamic temp-table that you create for the `ttHandle TABLE-HANDLE` parameter:

Doc/Lit SOAP request for TABLE-HANDLE, ttHandle

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope namespaces defined here...>
  <soap:Header .../>
  <soap:Body>
    <dynttIO xmlns="urn:DynTTSrvc:DynTT">
      <ttHandle>
        <DataSet xmlns="">
          <schema>
            <!-- Schema definition goes here -->
          </schema>
          <Data>
            <!-- Data goes here -->
          </Data>
        </DataSet>
      </ttHandle>
    </dynttIO>
  </soap:Body>
</soap:Envelope>
```

The ttHandle TABLE-HANDLE parameter becomes an XML <ttHandle> element containing the <DataSet> element that contains the schema and the data.

Note: Not shown is any required object ID that must be sent in the SOAP header for the object on which dynttIO( ) is invoked.

This is a sample XML Schema created by VB.NET for the TABLE-HANDLE contained by the sample <DataSet> element in the sample Doc/Lit SOAP request:

TABLE-HANDLE <schema> element for an Doc/Lit SOAP request

```
<xs:schema id="Data" xmlns=""
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="Data" msdata:IsDataSet="true">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="Item">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Name" type="xs:string" minOccurs="0"/>
              <xs:element name="Number" type="xs:int" minOccurs="0"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Note the definition of the <Data> element containing the data for the table, and how the column type information is specified within the <Item> element.

This is a sample <Data> element you would create to accompany the specified schema in the sample Doc/Lit SOAP request, including the column values for the two rows initialized in the sample VB.NET code:

TABLE-HANDLE <Data> element for an Doc/Lit SOAP request

```
<Data>
  <Item>
    <Name>Fred</Name>
    <Number>1</Number>
  </Item>
  <Item>
    <Name>Barney</Name>
    <Number>2</Number>
  </Item>
</Data>
```

Receiving a SOAP fault message

The following sample SOAP fault shows the response to a request sent for an object (OrderInfo) of a disabled Web service:

Sample SOAP fault for disabled Web service

```
<?xml version="1.0" encoding="UTF-8" ?>
<soap:Envelope namespaces defined here…>
  <soap:Body>
    <soap:Fault>
      <faultcode>SOAP-ENV:Server</faultcode>
      <faultstring>An error was detected ... request. (10893) </faultstring>
      <detail>
        <FaultDetail ... >
          <errorMessage>The urn:OrderSvc:OrderInfo service is unavailable
                        (10921)</errorMessage>
          <requestID> 2e62cab6b81150d5:-17c6a3c:f1dffbd11b:-8000#1f
          </requestID>
        </FaultDetail>
      </detail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Caution: When exiting the program in response to a SOAP fault, you must release all Web service objects you created in the program.

Note: SOAP fault messages conform to SOAP 1.1.

The contents of the highlighted SOAP elements include some of the key components of the fault message, containing such error information as a general error code (`<faultcode>`) and message (`<faultstring>`). These elements are followed by more detailed information, including the message indicating a more precise cause for the error (`<errorMessage>`), and a unique identifier for the request that returned the error (`<requestID>`). The contents of the `<detail>` element is defined in the WSDL `Types` section. For more information on handling errors, including SOAP fault messages generated for WSA-managed Web services, see Testing and Debugging OpenEdge SOAP Web Services on page 68.

Passing static and dynamic ProDataSet parameters

The examples in the following sections show how parameters are passed for static and dynamic ProDataSets.

These examples rely on the sample Web service specifications shown in the following table.

Property or component	Value or name	Object type
Web service	`CustOrders`	–
URL	`http://servicehost:80/wsa/wsa1`	–
Session model	Session-Free	–
SOAP format	`Doc/Lit`	–
TargetNamespace	`urn:CustOrders`	–

Property or component	Value or name	Object type
WSDL objects	CustOrdersObj	AppObject
	dsCustOrd	ProDataSet dsCustOrd

Note: This section does not continue the previous examples. It uses C#.NET, rather than VB.NET like most of this chapter.

Invoking a method with a DATASET parameter

To invoke a method that passes a DATASET parameter, the client application typically must:

- Create rows using the interface object that represents the TABLE row
- Insert the rows in arrays that represent the constituent temp-tables in the ProDataSet

For example, the following snippet of code passes a ProDataSet parameter represents and operation in the Web service:

getCustOrders.p

```
/* getCustOrders.p */
DEFINE TEMP-TABLE ttCust NO-UNDO
    FIELD CustNum AS INTEGER
    FIELD Name AS CHARACTER
    INDEX CustNumIdx IS UNIQUE PRIMARY CustNum.

DEFINE TEMP-TABLE ttOrder NO-UNDO
    FIELD OrderNum AS INTEGER
    FIELD CustNum AS INTEGER
    INDEX OrderNumIdx IS UNIQUE PRIMARY OrderNum
    INDEX CustOrdIdx IS UNIQUE CustNum OrderNum.
```

```
DEFINE TEMP-TABLE ttOrderLine NO-UNDO
    FIELD OrderNum AS INTEGER
    FIELD LineNum AS INTEGER
    INDEX OrderLineIdx IS UNIQUE PRIMARY OrderNum LineNum.

DEFINE DATASET dsCustOrd FOR ttCust, ttOrder, ttOrderLine
    DATA-RELATION CustOrdRel FOR ttCust, ttOrder
        RELATION-FIELDS (CustNum, CustNum)
    DATA-RELATION OrdLinesRel FOR ttOrder, ttOrderLine
        RELATION-FIELDS (OrderNum, OrderNum) NESTED.

DEFINE INPUT PARAMETER iCustNum AS INTEGER.
DEFINE OUTPUT PARAMETER DATASET FOR dsCustOrd.
/* fill dataset and return to caller */
...
```

When you add a Web Reference for the CustOrders Web service to Microsoft® Visual Studio, it creates the following proxies in a References file for the getCustOrders operation:

Sample C#.NET proxy code for getCustOrders

```
public string getCustOrders(
       System.Nullable<int> iCustNum, out dsCustOrd dsCustOrd)
  {
    object[] results = this.Invoke("getCustOrders", new object[] {
      iCustNum});
    dsCustOrd = ((dsCustOrd)(results[1]));
    return ((string)(results[0]));
  }

...

public partial class dsCustOrd {
    private dsCustOrdTtCust[] ttCustField;
    private dsCustOrdTtOrder[] ttOrderField;
    ...
}

...

public partial class dsCustOrdTtCust {
    private System.Nullable<int> custNumField;
    private string nameField;
    ...
}

...

public partial class dsCustOrdTtOrder {
    private System.Nullable<int> orderNumField;
    private System.Nullable<int> custNumField;
   <!-- nested data relation with ttOrderLine -->
   private dsCustOrdTtOrderTtOrderLine[] ttOrderLineField;
    ...
}

...
```

```
public partial class dsCustOrdTtOrderTtOrderLine {
    private System.Nullable<int> orderNumField;
    private System.Nullable<int> lineNumField;
    ...
}
```

When you reference the Web service in your code, Microsoft Visual Studio offers these proxy objects as appropriate. You can then create code to access the ProDataSet parameter like the following:

Sample C#.NET application for getCustOrders operation

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;

namespace getCustOrders
{  class Program
  {
    static void Main(string[] args)
    {
      int iCustNum;
      wsCustOrders.CustOrdersService ws = new
                                          wsCustOrders.CustOrdersService();
      wsCustOrders.dsCustOrd dsCustOrd;
      iCustNum = 1;
      ws.getCustOrders(iCustNum, out dsCustOrd);

      FileStream fs = new FileStream("getCustOrder.out", FileMode.Create);
      StreamWriter w = new StreamWriter(fs);

      //temp-table ttCust
      for (int i = 0; i < dsCustOrd.ttCust.Length; i++)
      {
         w.Write(dsCustOrd.ttCust[i].CustNum);
         w.Write(dsCustOrd.ttCust[i].Name);
         w.WriteLine();
      }
      //temp-table ttOrder
      for (int i = 0; i < dsCustOrd.ttOrder.Length; i++)
      {
        w.Write(dsCustOrd.ttOrder[i].CustNum);
        w.Write(dsCustOrd.ttOrder[i].OrderNum);
        w.WriteLine();
        //nested temp-table ttOrderLine
        for (int j = 0; j < dsCustOrd.ttOrder[i].ttOrderLine.Length; j++)
          {
             w.Write(dsCustOrd.ttOrder[i].ttOrderLine[j].LineNum);
             w.Write(dsCustOrd.ttOrder[i].ttOrderLine[j].OrderNum);
          }
        }
        w.Close();
      }
  }
}
```

Note: Because .NET has a proprietary method of recognizing and exposing ADO .NET DataSets in WSDL documents, its toolkit cannot translate the WSDL definition of a ProDataSet directly into an ADO .NET DataSet. For an example of creating an ADO .NET DataSet from a ProDataSet parameter, see Creating .NET DataSets from ProDataSet parameters.

Invoking a method with a DATASET-HANDLE parameter

To invoke a method that passes a `DATASET-HANDLE` parameter:

- The client application must create and send an XML Schema along with the data to fully describe the dynamic temp-table in the SOAP request message. The `DATASET-HANDLE` parameter in a SOAP request or response consists of an XML element containing two child elements:
  - An XML schema element representing the schema for the `DATASET-HANDLE`

  - An XML representation of data, using a data element with child elements representing individual rows for each constituent temp-table

- The client application must parse the XML Schema and data from the SOAP response message to make the `DATASET-HANDLE` accessible as native data within the application.

For example, the following snippet of code is from a procedure that passes a dynamic ProDataSet parameter:

```
/* getDynDs.p */
DEFINE INPUT PARAMETER criteria as CHARACTER.
DEFINE OUTPUT PARAMETER DATASET-HANDLE hDset.

/* Create dataset based on criteria
** fill dataset and return to caller */
. . .
```

When you add a Web Reference for the Web service to Microsoft® Visual Studio, it creates the following proxies in a `References` file for the `getDynDS` operation:

Sample C#.NET proxy code for getDynDs

```
public string getDynDs(
   string criteria,
   out DataSetHandleParam hDset) {
      object[] results = this.Invoke("getDynDs", new object[] {
         criteria});
      hDset = ((DataSetHandleParam)(results[1]));
      return ((string)(results[0]));
}
. . .

public partial class DataSetHandleParam {
   private System.Xml.XmlElement[] anyField;
. . .
}
```

When you reference the Web service in your code, Microsoft Visual Studio offers these proxy objects as appropriate. You can then create code to access the ProDataSet parameter like the following:

Sample C#.NET code for invoking getDynDs

```
sampleDynDS.sampleDynDSService mySvc = new sampleDynDS.sampleDynDSService();
sampleDynDS.DataSetHandleParam dsh;

mySvc.getDynDs(textBox1.Text, out dsh);
```

Testing and Debugging OpenEdge SOAP Web Services

This chapter explains how to test and debug OpenEdge Web services. It assumes you are familiar with Web services and the Web Services tools in OpenEdge, as described in the preceding chapters of this manual. Information in this chapter is useful for both Web service and client developers to troubleshoot OpenEdge Web services and client applications that access them.

Testing the AppServer application as a Web service

You can generally use two different approaches for testing an AppServer application written to support an OpenEdge Web service:

- **Unit testing from ABL** — From ABL, you can access AppServer functionality directly or as an OpenEdge Web service.
- **Testing from a Web service client** — From a Web service client using any platform, including ABL, you can thoroughly test the Web service as it might behave in the environment in which you will deploy it.

The approach that you take depends on your preferences, needs, and experience.

Unit testing from ABL

Accessing the AppServer directly is useful for doing unit testing without having to bother with the overhead and inconvenience of managing the Web service deployment environment. This form of testing also eliminates some of the ABL logic required to access a Web service. However, it can provide a good measure of confidence in the basic integrity of the application service itself.

Testing from a Web services client

This method allows you to create a client application environment and functionality exactly as you expect real Web service clients to be created, and you can test the application end-to-end using a complete Web service testing infrastructure.

The rapid prototyping capability of ABL can also make it a convenient platform to test the application service as a Web service, depending on the requirements of the Web service. Some client platforms might handle certain types of Web service interaction more easily than others. However, especially where ABL most easily handles a particular Web service application feature, modifying an existing ABL unit test bed to access the AppServer as a Web service is a natural next step in testing Web service functionality.

Otherwise, especially if you expect your primary client base to use another platform, you might want to test on that platform to fully understand what users of that platform need to do in order to use the Web service that you plan to deploy.

To test from a Web services client:

1. Configure the AppServer to support the required application session model:
   - For information on configuring the session-managed operating modes, see *OpenEdge Application Server: Developing AppServer Applications* and the *OpenEdge Management and OpenEdge Explorer* online help.
   - For information on configuring the state-free operating mode (session-free model) see the *OpenEdge Management or OpenEdge Explorer* online help.
2. Define the Web service and generate the Web service mapping (WSM) file using ProxyGen. For more information, see *OpenEdge Development: Open Client Introduction and Programming.*

3. Generate a test WSDL file from ProxyGen when you generate the WSM file, or deploy the WSM file to a WSA instance to obtain a deployed WSDL file. For more information on deploying a Web service, see the sections on deploying and managing Web services in *OpenEdge Application Server: Administration.*
4. Write the test client application, using the WSDL file to help generate client interfaces or to manually write the code to invoke Web service requests. For more information, see Creating OpenEdge SOAP Web services. For information on building an ABL client to access your Web service, see Creating ABL clients to consume SOAP Web services.
5. Once you have an executable test client, ensure that the AppServer and WSA are running, and the WSA instance where you deployed the Web service is enabled to allow client access to the Web service. For more information on running and enabling the WSA for Web service client access, see the sections on working with the WSA and deploying Web services in *OpenEdge Application Server: Administration.*
6. Begin testing the client.

Setting error information levels

To make it easier to diagnose errors, you can control the amount of information provided for the following sources of Web service error information:

- WSA log file
- Web service SOAP fault response messages
- AdminService log file
- AppServer log file
- NameServer log file

For information on managing logging for the AdminService, NameServer, and in general for all OpenEdge servers managed by the Unified Explorer Framework, see *OpenEdge Getting Started: Installation and Configuration*. For more information on managing logging for an AppServer, see *OpenEdge Application Server: Administration*. The following sections describe how to control logging and SOAP fault information output for the WSA and Web services.

WSA and Web service logging level properties

The following table shows the property files where these settings occur and the default installation settings for basic properties that affect output for each of these information sources.

Table 1: Setting error levels for WSA and Web services

Affected error output	Property files	Default property setting
WSA log file	`ubroker.properties` (WSA properties)	`loggingLevel=2`
	`*.props` (Web service properties)	`serviceLoggingLevel=2`
SOAP fault response messages	`*.props` (Web service properties)	`serviceFaultLevel=2`

Note: For complete information on the log file settings for the WSA log file, including more advanced logging properties, see the OpenEdge Management or OpenEdge Explorer online help and the `unbroker.properties.README` file in the `properties` directory under the OpenEdge installation directory.

Logging information

The properties in Table 1: Setting error levels for WSA and Web services on page 69 have the following functions and value ranges (the greater the value, the more information is provided):

- **`loggingLevel`** — Affects information logged for a single WSA instance. Set this property to change the logging detail for WSA administration, WSDL document retrieval, user authorization, and HTTP request handling. Valid values range between 1 and 4.
- **`serviceLoggingLevel`** — Affects information logged for a single deployed Web service. Set this property to change the logging detail for the processing of a Web service client request. Valid values range between 1 and 4.
- **`serviceFaultLevel`** — Affects the information returned in all SOAP faults from a single deployed Web service. Set this property to change the level of detail returned for client exceptions. Valid values are 2 or 3.

Hints for setting the WSA loggingLevel property

Set the `loggingLevel` property for the WSA instance to the following value:

- **3** — To get information about user authorization errors or to log the start of a new client request for any Web service deployed to this WSA instance.
- **4** — To get detailed information about WSA startup parameters.

Hints for setting the Web service serviceLoggingLevel property

Set the `serviceLoggingLevel` property for the Web service to the following value:

- **4** — To record Web service request parameter values.
- **3** — To track Web service request execution. Use this value during Web service development to provide more information for each Web service request error.
- **2** — To reduce logging overhead when the Web service is running in production. This value only allows severe errors to be recorded that affect the Web service as a whole, and no errors to be recorded for single Web service requests.

During production run time, you might temporarily set the value to 3 or 4 when working with a specific client to resolve an error, then return the logging level to 2 for general client access when the problem has been resolved.

Setting the logging level properties

Settings to the `serviceLoggingLevel` and `serviceFaultLevel` take effect immediately at run time. You can also set an immediate run-time value for the WSA `loggingLevel` property, as well as a persistent value that takes affect the next time the WSA is started.

To change these settings, use OpenEdge Management or OpenEdge Explorer or the `wsaman` utility. If you are working with a WSA installation on a system that does not have an AdminServer installed, you must edit the `ubroker.properties` file to change the persistent value of `loggingLevel` property.

For more information on setting these logging level properties and their appropriate values using OpenEdge Management and OpenEdge Explorer, see the online help. You can also set the `loggingLevel` property If you are using the `wsaman` utility, see the sections on the `wsaman` utility in *OpenEdge Application Server: Administration*. For more information on the `serviceLoggingLevel` and `serviceFaultLevel` properties, see the Web service properties reference sections of *OpenEdge Application Server: Administration.*

For complete information on OpenEdge general logging capabilities, see *OpenEdge Development: Debugging and Troubleshooting.*

For more information on using log files, see Identifying relevant log files on page 71. For more information on handling SOAP faults, see Working with SOAP faults on page 71.

Identifying relevant log files

A common way to detect software errors of all kinds, not just Web service errors, is to examine log files for error messages. The following table lists OpenEdge log files providing Web-service-related messages. These files all reside in your OpenEdge work directory.

Table 2: OpenEdge log files providing Web service-related messages

This component . . .	Logs messages here . . .
WSA instance	*instance-name*`.wsa.log`
Web service	*instance-name*`.wsa.log`
AdminServer	`adminserv.log`
AppServer	*instance-name*`.broker.log`*instance-name*`.server.log`
NameServer	*instance-name*`.NS.log`

For information on managing logging for the AdminService, NameServer, and in general for all OpenEdge servers managed by the Unified Explorer Framework, see *OpenEdge Getting Started: Installation and Configuration*. For more information on managing logging for an AppServer, see *OpenEdge Application Server: Developing AppServer Applications.*

Working with SOAP faults

When an error occurs while processing a Web service client request, the WSA generates a SOAP fault message and returns it to the client application. The client application should catch all SOAP faults and handle them programmatically, perhaps by displaying an error message to the user or by logging the fault information for review off line. If the client application does not catch the SOAP fault, the behavior of the client application depends on the default error handling of the client software development kit.

Setting the serviceFaultLevel property

You can control the amount of information provided with SOAP fault messages by how you set the `serviceFaultLevel` property for a Web service. Any change in value takes effect immediately at run time.

Set the `serviceFaultLevel` property to the following value:

- **2** — To log general error information in a production environment
- **3** — To log error information in a development environment or in a production environment where more detailed information is required

Note that all Web service logging and fault levels are controlled individually for each Web service to further reduce processing overhead. For more information on setting this property, see Setting error information levels on page 69.

Typical SOAP fault response message

The following is the content of a typical SOAP fault response that appears within the body of a WSA-generated SOAP fault message:

```
<soap:Fault>
  <faultcode>SOAP-ENV:Server</faultcode>
  <faultstring>An error was detected … request. (10893) </faultstring>
  <detail>
    <FaultDetail xmlns="http://servicehost:8080/wsa/wsa1">
      <errorMessage>The urn:OrderSvc:OrderInfo service is unavailable (10921)
      </errorMessage>
      <requestID>2e62cab6b81150d5:-17c6a3c:f1dffbd11b:-8000#1f
      </requestID>
    </FaultDetail>
  </detail>
</soap:Fault>
```

Note that SOAP faults can be returned by the client's own SOAP libraries and might have a different XML definition for the contents encapsulated by the `<detail>` element, depending on the Web service client toolkit that is used. However, a SOAP fault generated by the OpenEdge WSA always includes an `<error Message>` and `<requestID>` element within the `<detail>` element section.

WSA's <errorMessage> element

The SOAP `<faultstring>` element provides only a very general description of the error. For the most effective diagnostic detail, always look at the information provided by the `<errorMessage>` element.

WSA's <requestID> element

Use the information provided by the `<requestID>` element when you contact the Web service deployer for assistance with an error. This information tells the Web service deployer whether more information can be obtained from the WSA log file. For more information on how the WSA uses the information in the `<requestID>` element to log information, see How the WSA logs Web service information on page 74.

Handling SOAP faults programmatically

The client application should handle all SOAP faults. Many SOAP client toolkits convert SOAP faults to client exceptions that can be handled in a `try...catch` block. The following code is a VB.NET fragment from the sample applications that are available on the Documentation and Samples (`doc_samples`) directory of the OpenEdge product DVD:

```
Try
    ' Code to access the Web service
    ...
    Catch soapEx As System.Web.Services.Protocols.SoapException
       Dim detail As String, reqId As String
          detail = parseSoapException(soapEx.Detail, reqId)
          MsgBox(detail, MsgBoxStyle.Critical, soapEx.Message)
End Try
```

In this example, `parseSoapException( )` is a client helper function to parse the SOAP fault detail provided in .NET sample code. You can find more examples like this in both the .NET and Java sample client applications. For more information on these sample applications, see Sample SOAP Web service applications.

Using a SOAP message viewer

To view SOAP messages as they are sent and received, use a SOAP message viewer. The following table lists some examples.

This SOAP message viewer . . .	Is . . .
WSAViewer	Provided with the OpenEdge installation
ProSOAPView	Provided with the OpenEdge installation
Microsoft SOAP Toolkit	Available for free download from Microsoft's Web site

To use a SOAP message viewer:

1. Change the Web service client to send requests to the port of the SOAP message viewer by changing the SOAP end point's URL host and port fields.
2. Configure the SOAP message viewer to forward requests to the host and port of the WSA, where the Web service is deployed, and run the viewer.

Configuring and running WSAViewer

To configure and run WSAViewer:

1. Enter the `wsaviewer` command from a Proenv shell using the following syntax:

```
wsaviewer listen-portWSA-hostWSA-port
```

2. Change the Web service URL in the Web service client to access the WSAViewer. Assume that the original Web service URL in the client is as follows:

```
http://bedrockquarry.com:8080/wsa/wsa1
```

   In the client, change this URL to use an agreed-upon viewer listening port:

```
http://bedrockquarry.com:8081/wsa/wsa1
```

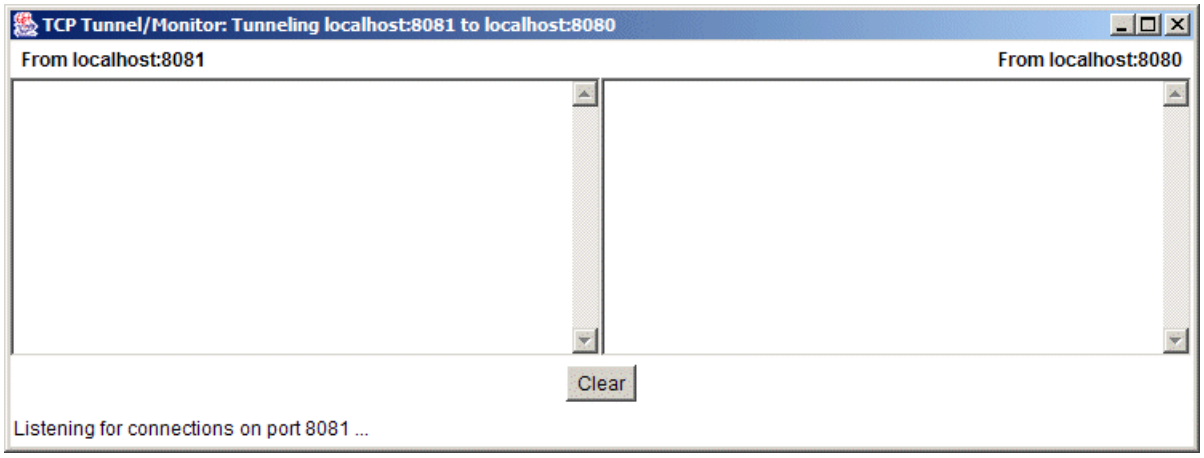
3. To run the WSAViewer, enter this command in the Proenv shell on the WSA machine:

```
wsaviewer 8081localhost8080
```

The WSAViewer runs configured to listen for requests from the reconfigured client interface on port 8081 and forward those requests to port 8080. SOAP responses from the WSA, then, move in the reverse direction.

The following figure shows the main window of the WSAViewer when it first opens.

Figure 1: WSAViewer main window



SOAP requests from the client appear in the left-hand viewer and SOAP responses from the WSA forwarded back to the client appear in the right-hand viewer. A continuous stream of requests and responses is recorded that you can review by using the vertical scroll bars.

To configure and run ProSOAPView, see Using ProSOAPView.

ProSOAPView is more general purpose than WSAViewer, and it allows you to trace not only SOAP messages, but also HTTP messages and other document contents exchanged between a client and server. ProSOAPView works as both an HTTP proxy server and an HTTP client that can connect to your choice of another proxy server or the final destination.

Identifying errors

This section describes how to identify errors and describes some of the more common errors that occur in WSA administration requests and in Web service application requests.

How the WSA logs Web service information

Each time the WSA starts up, each WSA instance receives a unique, alphanumeric identifier (*WSA ID*), which the WSA always places in a log file entry that indicates the WSA is starting up. This WSA ID appears in the log file similar to this example, always appearing after the `"ID"`:

```
ID 2bbafbc35852308b:7a8a02:f18e8ee918:-8000
```

When a WSA instance first receives a request for one of its deployed Web services, the WSA instance assigns the request an alphanumeric identifier (*request ID*) that is unique for the current session of the WSA instance. When the WSA logs information about a Web service request, whether the information concerns processing steps or errors, the WSA always includes its unique request ID in the log entry text using the following format, where *`reqid-value`* is the request ID for the Web service request to which the log entry applies:

```
(reqid:reqid-value)
```

When the WSA returns a SOAP fault to a client, it includes the request ID of the Web service request that triggered the SOAP fault, as well as the WSA ID of the WSA instance where the Web service is deployed. The WSA places these two IDs within the `<requestID>` element of the SOAP fault message according to the following format, where *`wsaid-value`* is the WSA ID and *`reqid-value`* is the request ID:

```
wsaid-value#reqid-value
```

The client can then obtain these two IDs from the `<requestID>` element of the SOAP fault and provide it to the WSA host operator to look up more detailed information in the corresponding entries of the WSA log file. The WSA operator then uses the WSA ID to locate the WSA log file for the corresponding WSA instance. They can now open the log file and search for log file entries that contain the "`(reqid:`*`reqid-value`*`)`" string to find all information related to the specific request and WSA instance.

Errors occurring in WSA administration requests

An error might occur in a WSA administration request. Such errors occur when you use OpenEdge Management or OpenEdge Explorer or the `wsaman` utility. Both these tools are front ends to the AdminServer, which creates SOAP requests as a Web service client, then sends them to the WSA. When an error in a WSA administration request occurs, it is logged. Where it is logged depends on when it occurred, as shown in the following table.

If the error occurred . . .	The error is logged . . .
While the request is being prepared in the AdminServer	In the AdminServer log
While the request is being processed in the WSA	In the WSA instance log

Errors occurring in Web service application requests

A Web service application request is initiated by a client, which creates a SOAP message and sends it to the WSA. The WSA processes the SOAP message, creates a response, and sends the response back to the client. An error can occur on the client side or the server side.

Client-side errors

A client-side error occurs before the request is sent or while it is being processed by the WSA.

An example of an error occurring before the request is sent is a mismatch in interface parameters. For these types of errors, the type of client response depends on the client platform.

Examples of errors occurring while the request is being sent (which involve a session-managed AppServer application) include:

- Forgetting to call the connect method first
- Not sending the correct Object ID in the SOAP header
- Web server not running
- Web service URL incorrectly specified (host or port)
- Network down
- Using HTTP instead of HTTPS when required

These types of errors result in a SOAP fault response and log file entries.

Server-side errors

A server-side error occurs while the request is being processed by the WSA and the error is not related to the contents of the SOAP request. Examples of server-side errors include:

- Forgetting to enable a specific deployed Web service
- Forgetting to enable client access to Web services for the WSA instance
- Configuring or starting an AppServer incorrectly—this might involve one or more of the following issues:
  - Invalid Propath on the AppServer
  - Invalid AppService (Application Service name)
  - Incompatible AppServer operating mode for the Web service session model
  - Missing ABL, or an incorrect version of ABL
- Configuring the Java Servlet Engine (JSE) incorrectly so it does not recognize the WSA

These types of errors result in a SOAP fault response and log file entries.

Sample error scenarios

This section discusses sample error scenarios. These scenarios involve a VB.NET client accessing a Web service whose specification is listed in the following table.

Web service specification	Value
Name	`OrderInfo`
TargetNamespace	`urn:www-progress-com:samples:Orderinfo`
AppObject	`OrderInfo`
ProcObject	`CustomerOrder`
Session Model	`Managed`

Note: These sample scenarios are not necessarily based on the samples available on the Documentation and Samples (`doc_samples`) directory of the OpenEdge product DVD or the Progress Documentation Web site.

Scenario 1: Web service deployed but not enabled

In this scenario, a Web service is deployed but not enabled. At this point:

1. The client sends the following connection request:

```
Connect_OrderInfo("","","")
```

2. The WSA gets the request and determines the Web service is disabled. At this point:

a. The WSA sends the client a SOAP response containing a SOAP fault.

b. The log files do not show any error.

The following is the SOAP response containing the SOAP fault:

```
<?xml version="1.0" encoding="UTF-8" ?>
<soap:Envelope  namespaces defined here…>
  <soap:Body>
    <soap:Fault>
      <faultcode>SOAP-ENV:Server</faultcode>
      <faultstring>An error was detected … request. (10893)
      </faultstring>
      <detail>
        <FaultDetail xmlns="http://servicehost:8080/wsa/wsa1">
          <errorMessage>The urn:www-progress-com:samples:OrderInfo
                        :OrderInfo service is unavailable (10921)
          </errorMessage>
          <requestID>2e62cab6b81150d5:-17c6a3c:f1dffbd11b:-8000#1f
          </requestID>
        </FaultDetail>
      </detail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

3. The interface translates the SOAP fault into a SOAP exception.
4. The client code catches the SOAP exception.
5. The client displays the following error message (assuming the client application has a user interface):



Note: All client applications should always catch SOAP faults and handle them. For client applications without a user interface, all error information must be written to an application log file to ensure it is not lost. If a .NET client does not do this, the .NET Framework's default exception handler displays the SOAP fault general error information. This might well hide and cause the loss of the more useful error detail information in the SOAP fault.

Scenario 2: ABL updated, but Web service not redeployed

In this scenario, ABL is updated with a different procedure prototype (signature), but the Web service is not updated accordingly. At this point:

1. The client sends the following method request:

```
FindCustomerByNum(3, custNameVar)
```

2. The WSA gets the request and asks the AppServer to run the procedure `FindCustomerByNum.p`.
3. The AppServer:
   a. Tries to run `FindCustomerByNum.p`
   b. Detects a parameter-type mismatch and records the following error message in its log, `*.server.log`:

```
[02/12/11@16:07:01.427-0500] P-000371 T-000370 0 AS --
  Mismatched parameter types passed to procedure   OrderInfo/FindCustomerByNum.p.
 (3230)
```

   c. Returns the error to the WSA
4. The WSA:
   a. Records the following message in its log:

```
[02/12/11@16:07:01.802-0500] P-000120 T-Thread-11 2 OrderInfo ABL-Provider
(reqid:6) Error in SOAP request execution: ABL
ERROR condition: Mismatched parameter types passed to
procedure OrderInfo/FindCustomerByNum.p. (3230) (7211) (10926)
```

   b. Sends the following SOAP containing a SOAP fault back to the client:

```
<?xml version="1.0" encoding="UTF-8" ?>
<soap:Envelope  namespaces defined here…>
  <soap:Body>
    <soap:Fault>
      <faultcode>SOAP-ENV:Server</faultcode>
      <faultstring>An error was detected … request. (10893)
      </faultstring>
      <detail>
        <FaultDetail xmlns="http://servicehost:8080/wsa/wsa1">
          <errorMessage>Error in SOAP request execution: ABL ERROR
            condition: Mismatched parameter types passed to
            procedure OrderInfo/FindCustomerByNum.p.  … (10926)
          </errorMessage>
          <requestID>2e62cab6b81150d5:-87f76e:f20f57227d:-8000#6
          </requestID>
        </FaultDetail>
      </detail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Note: This example uses a value of 3 for the `serviceFaultLevel` property.

Notice the correspondence between the "(reqid:6)" string in the log file entry and the "<requestID>...2e62...#6" string in the SOAP fault message, indicating information for the same request.

5. The interface translates the SOAP fault into a SOAP exception.
6. The client catches the SOAP exception.
7. The client displays the following error message (assuming the client application has a user interface):



Scenario 3: Web service deployed but AppServer not running

In this scenario, a Web service is deployed, but its AppServer is not running. At this point:

1. The client sends the following connection request:

```
Connect_Orderinfo("", "", ")
```

2. The WSA:
   a. Receives the request
   b. Determines that the AppServer is not available

c. Records the following in its log:

```
[02/12/11@16:17:27.349-0500] P-000120 T-Thread-11 2 OrderInfo ABL-Provider (reqid:8)
 Error in SOAP request execution: Connect Failure: Application Service asbroker1
 Unknown at NameServer
at Host servicehost Port 5162. (8234) (10926)
```

d. Sends a SOAP message with a SOAP fault back to the client. The message is as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<soap:Envelope  namespaces defined here…>
  <soap:Body>
    <soap:Fault>
      <faultcode>SOAP-ENV:Server</faultcode>
      <faultstring>An error was detected … request. (10893)
      </faultstring>
      <detail>
        <FaultDetail xmlns="http://servicehost:8080/wsa/wsa1">
          <errorMessage>Error in SOAP request execution: Connect
            Failure: Application Service asbroker1 Unknown at
            NameServer at Host servicehost Port 5162. (8234) (10926)
          </errorMessage>
          <requestID>2e62cab6b81150d5:-87f76e:f20f57227d:-8000#8
          </requestID>
        </FaultDetail>
      </detail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

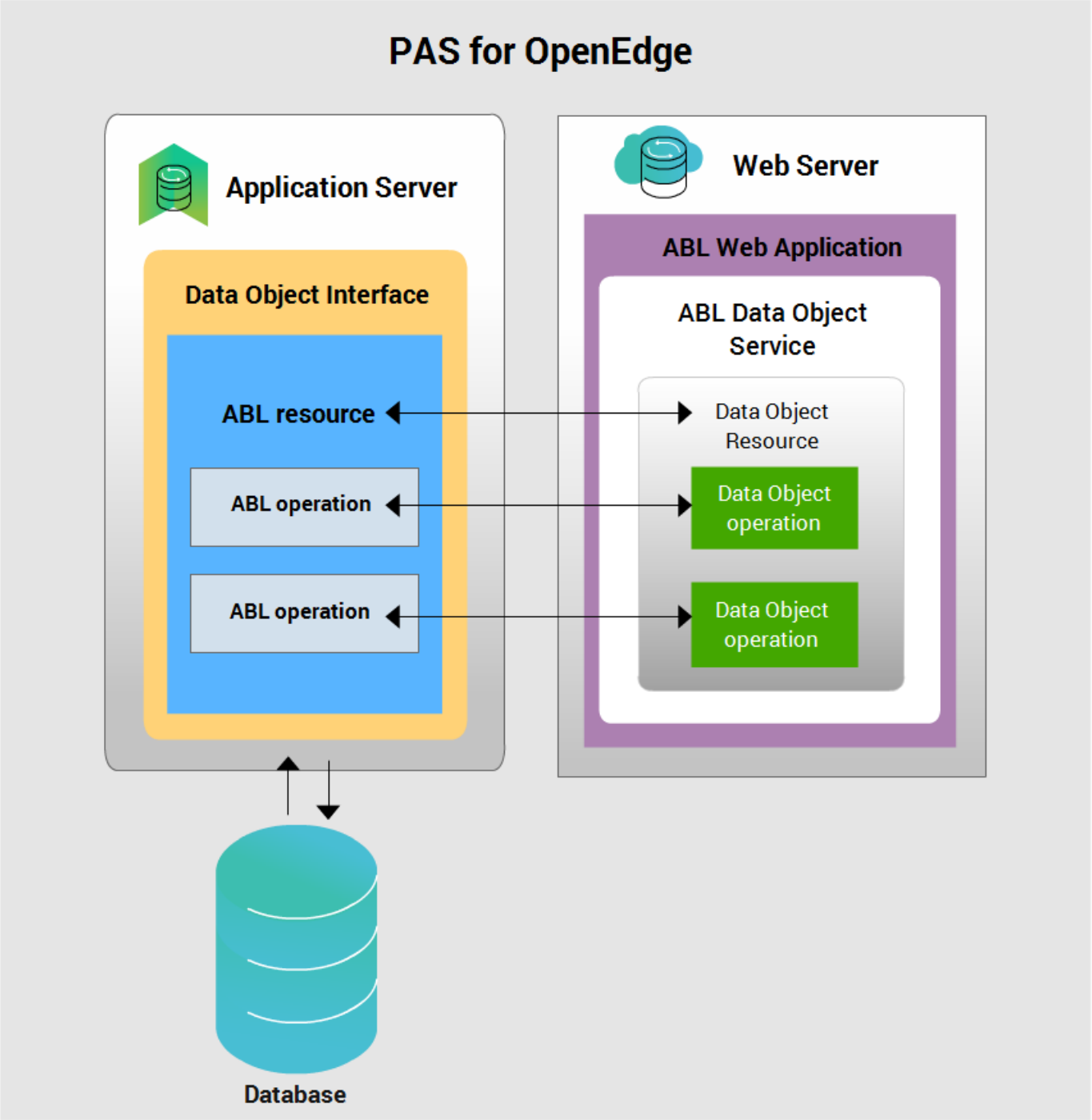
3. The interface translates the SOAP fault into a SOAP exception.
4. The client catches the SOAP exception.
5. The client displays this error message (assuming the client application has a user interface):



Develop a Progress Data Object Service

This section describes how to create a Progress Data Object Service.

Data Object services overview



A Data Object Service consists of Data Object resources that correspond to business entities (ABL user-defined classes or persistent procedures). Each Data Object resource consists of one or more Data Object operations that correspond to ABL operations (methods or internal procedures).

Data Object services use Progress Application Server (PAS) for OpenEdge and Progress OpenEdge databases. PAS for OpenEdge is both an application server and a Web Server. The application server contains the Data Object interfaces to ABL resources (procedures, classes, or, database tables). The Web Server contains one or more ABL Web Applications. Each ABL Web Application consists of one or more Data Object services.

You can create a new ABL Web App project or convert an existing OpenEdge project into an ABL Web App project. Before you create an ABL Web App project, you must set up the development environment.

You can choose to create a WebHandler-based or a REST-based Data Object service. A WebHandler-based Data Object service uses a WebSpeed transport for client connections, while a REST-based service uses a REST transport.

Data Object service operations

Data Object services support the following operations:

- Read—A standard HTTP operation that returns records to the client. It also supports server-side preprocessing to filter data, sort data, or, page through data.

Note: Server-side filtering, sorting, and paging are very useful when dealing with a large number of records.

- Create, Update, Delete—Standard HTTP operations that create, update, or delete a record as per a client request.
- Submit—A standard HTTP operation that can create, update, and delete multiple records in a single request.
- Invoke—A custom operation that enables you to go beyond the limitations of standard HTTP operations and execute any other method in the Business Entity other than the standard operations.

Note: The Create, Read, Update, and Delete operations use the POST, GET, PUT, and DELETE HTTP methods respectively.

Creating Data Object services

To create Data Object services, you must:

1. Set up the development environment.
2. Create Data Object interfaces.
3. Create Data Object services.

After you create your Data Object services, you test them using a PAS for OpenEdge instance and package them as one or more ABL Web Applications.

Setting up the development environment

You need an ABL Web App project to start creating Data Object services. You can create an ABL Web App project from scratch or you can convert an existing OpenEdge project into an ABL Web App project.

Creating an ABL Web App project from scratch

When you create an ABL Web App project, a Data Object service is auto-generated. The default service, which appears under Defined Services in the project folder in the Project Explorer, uses the project name. The service is either WebHandler-based or REST-based depending upon what you choose while creating the project.

If you select WebHandler as the service provider when creating an ABL Web App project, the project will contain the following facets:

- ABL AppServer
- ABL Data Object WebSpeed Service
- ABL Web App
- OpenEdge
- Progress Adapters

If you select REST as the service provider, the ABL Web App project will contain the following facets:

- ABL AppServer
- ABL Data Object Service
- ABL REST Service
- ABL Web App
- OpenEdge
- Progress Adapters

Regardless of the service provider you choose at the time of project creation, you can configure both WebHandler-based and REST-based Data Object services in the same project. As mentioned earlier, the service provider determines which transport (WebSpeed or REST) is used by clients accessing a Data Object service.

If you want to create an ABL Web App project from scratch, you must use the New OpenEdge Project wizard.

Follow these steps to create a new ABL Web App project from scratch:

1. Open Progress Developer Studio for OpenEdge.
2. Click File > New > OpenEdge Project. The New OpenEdge Project wizard appears.
3. On the Create an OpenEdge Project page:
4. Specify a name in the Project name field. Optionally, specify a location.
5. Click the Server tab on the left.
6. Retain PAS for OpenEdge in the Select a server section.
7. Select REST from the Select a transport drop-down.
8. Check the Create a Data Object Service (Generate a JSDO catalog) checkbox. Click Next.
9. On the Provide ABL Web App deploy details page:
10. Select Deploy as WebApp, if it is not already selected.
11. Retain the default module name and ABL source folder name.
12. Select the PAS for OpenEdge instance to which you want to deploy the ABL web application. Click Next.
13. On the Create an ABL Service page, retain the default service name and service URI, and click Next.
14. On the Select AVM and layout options page, retain the default options and click Next.
15. On the Define PROPATH page, retain the default options and click Next.
16. Select the database connections on the Select database connections page and then click Finish.

Creating Data Object interfaces

A Data Object interface is an annotated ABL resource (a class file or a procedure file). The ABL resource must be a single class file or an external procedure that can be implemented as a singleton[7] object.

If you have existing ABL resources, you annotate them using the Define Service Interface wizard.

If you want to create a Data Object interface from scratch, you must use the New Business Entity wizard and then add code to the class file that is generated. You can also use the New Business Entity wizard to quickly create a Data Object interface from a database table. OpenEdge automatically generates fully functional operations for the selected database table.

Note: You can only create class files using the New Business Entity wizard. If you want to use a procedure, you must create a procedure and annotate it using the Define Service Interface wizard.

Using the Define Service Interface wizard

Before you annotate an existing ABL resource, you must ensure that the ABL resource and its operations meet the following requirements:

- The class or procedure can be executed as a singleton object.
- ABL operations that implement CRUD and Submit operations must use the same data model.
- The return type for class methods must be VOID. Any return values from internal procedures and user-defined functions are ignored.
- The fields of the temp-tables in the data model, and the fields for the parameter lists and return types of ABL operations that implement invoke operations can use all ABL data types except RAW and RECID.
- If you want to use the Submit operation, the temp-table (or the temp-tables of the ProDataSet) must also define before-tables to support before-imaging.
- If you want the hierarchy of the parent and child temp-tables of a ProDataSet to be retained, you must define the data relation between them with the NESTED option.

Each ABL operation that implements a standard operation must have a prescribed parameter list based on the data model that the Data Object resource supports.

Standard operations and prescribed parameters

Each standard operation must have a fixed parameter set. The table shows the required parameters for each ABL operation that corresponds to a standard operation.

Built-in operation	Prescribed parameter list
Create	INPUT-OUTPUT { TABLE table-name | DATASET dataset-name | TABLE-HANDLE table-hdl | DATASET-HANDLE dataset-hdl}

[7] A singleton resource is a class or external procedure that once instantiated is shared by all consumers of the class or procedure. ABL classes are inherently coded to be instantiated as singletons. However, an external procedure must not contain DEFINE PARAMETER statements in the main block to be instantiated as a singleton. An external procedure can contain internal procedures and user-defined functions with their own parameters, each of which can implement a Data Object operation.

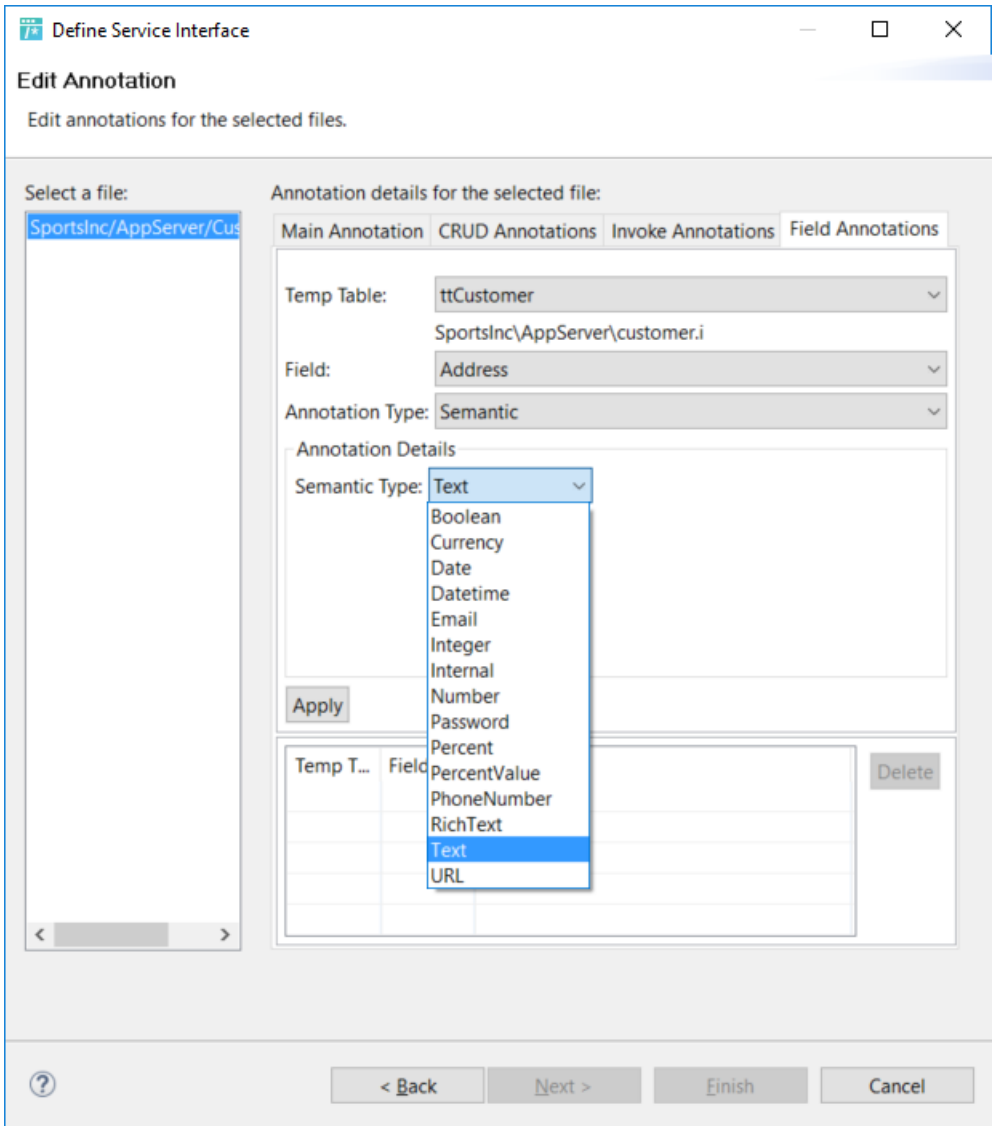
Built-in operation	Prescribed parameter list
Read	INPUT filter AS CHARACTER, OUTPUT { TABLE table-name | DATASET dataset-name | TABLE-HANDLE table-hdl | DATASET-HANDLE dataset-hdl}
Update	INPUT-OUTPUT { TABLE table-name | DATASET dataset-name | TABLE-HANDLE table-hdl | DATASET-HANDLE dataset-hdl}
Delete	INPUT-OUTPUT { TABLE table-name | DATASET dataset-name | TABLE-HANDLE table-hdl | DATASET-HANDLE dataset-hdl}
Submit	INPUT-OUTPUT {DATASET dataset-name| DATASET-HANDLE dataset-hdl}

All standard operations for a particular Data Object resource must use the same data model. Because of this, all the CRUD operations for a given Data Object resource must use the same parameter type. Therefore, within an ABL resource, you must ensure that all the ABL operations that will correspond to the built-in operations use the same parameter type.

Unlike other operations, the Read operation defines only one input parameter. The parameter must be named “filter”. If it is named differently, it will not function properly when the JavaScript method calls the read operation.

The filter parameter is a string. As the name suggests, you can use the filter in a WHERE clause to filter the records returned by a read operation. However, you do not need to limit yourself to passing only a filter. Since it is a string parameter, you can place additional information in the string and then parse it in the ABL method or procedure corresponding to the read operation.

Semantic types



Semantic types are field-level annotations that determine the type of data that can be stored in temp-table fields. You can specify default/ basic semantic types (such as Text—for CHARACTER, and Boolean—for LOGICAL) and also user-defined semantic types for your temp-table fields.

Suppose you want to ensure that a user, who has to add or update customer records to the Customer table, enters a properly formatted email ID. Then, you must define the semantic type, Email, while creating a Data Object interface using the Define Service Interface wizard. A published Data Object service that has the Email semantic type defined will reject any improperly formatted email IDs. This way you have more control over how data is entered into certain types of fields.

The following table lists each semantic type with its description:

Semantic type	Description
Text	Specify this if the user must enter a single text line.

Semantic type	Description
RichText	Specify this if the user can enter multi-line, formatted text.
Integer	Specify this if the user must enter an integer value.
Number	Specify this if the user must enter a property formatted decimal value.
Boolean	Specify this if the user must enter a Boolean value.
Date	Specify this if the user must enter date.
Datetime	Specify this if the user must enter date with time. This semantic type supports timezone specification.
Currency	Specify this if the user must enter a decimal value with a currency symbol.
Percent	Specify this if you want the value that the user enters to be stored in the database as a fraction (1/100th). If the user enters 10%, the value will be stored in the database as .01.
PercentValue	Specify this if you want the value entered by the user to be stored in the database as it is. If the user enters 10%, the value will be stored in the database as 10.
Email	Specify this if the user must enter text with a single @ character delimiter.
PhoneNumber	Specify this if the user must enter a phone number. This accepts numbers, parentheses, and dashes.
URL	Specify this if the user must enter a URL.
Password	Specify this if what the user enters must be displayed as hidden characters.
Internal	Specify this if a particular field must not be displayed to the user.

Note: If you define the Integer, Number, Currency, Percent, or PercentValue semantic type, you will have to further specify minimum and maximum values for the semantic type.

Using the Define Service Interface wizard

After you ensure that the ABL resource meets all the requirements, you can create a Data Object interface using the Define Service Interface wizard.

Follow these steps to create a Data Object interface using the Define Service Interface wizard:

1. Open Progress Developer Studio for OpenEdge.
2. Right-click the ABL Web App project and then select Progress OpenEdge > Define Service Interface. The Define Service Interface wizard appears.
3. In the Define Service Interface wizard:
   a. Select Data Object as the Definition mode.
   b. Select the ABL resource that needs to be annotated under Workspace resources. This populates the ABL routines section on the right with the list of ABL operations in the selected resource.
   c. Select the ABL operations (create, read, update, submit, etc) that need to be exposed under ABL routines.
   d. Optionally, select Backup selected resources.
   e. Click Next.
4. In the Edit Annotation page:
   a. In the Main annotation tab:
   b. Ensure that singleton is selected as the Execution mode.
   c. Click Browse and select the schema file that is being used by the ABL resource.
   d. Optionally, you can edit the Resource URI. By default, the ResourceURI is the ABL resource name.
5. In the CRUD Annotations tab, select the routine name in the Routine Name column for each operation that needs to be exposed. Note: If you are using the Submit operation, you must select the Before-image option. The Submit option is only applicable to the datasets selected on the Main Annotation tab. It is not applicable to temp-tables.
6. In the Invoke Annotations tab:
   a. Ensure that the Invoke option is selected for all the ABL routines that have to be exposed as Invoke operations.
   b. Ensure that you de-select the Invoke option for all the operations that have been mapped to standard operations (CRUD + Submit).
7. In the Field Annotations tab:
   a. Select the relevant temp table.
   b. Select the temp table field for which you want to define a semantic type.
   c. Select Semantic Type as the annotation.
   d. Select the appropriate semantic type. Note: If you choose Integer, Number, Currency, Percent, or PercentValue as the semantic type, you will have to further specify its minimum and maximum values.
   e. Click Apply.
8. Click Finish. This annotates the class file and the operations that must be exposed.

Using the New Business Entity wizard

Using the New Business Entity wizard, you can create a Data Object interface that contains one of the following:

- A single Read operation
- A full set of CRUD operations
- A full set of CRUD operations plus the Submit operation

Note: You must use the Define Service Interface wizard to create custom Invoke operations.

You can use the New Business Entity wizard to create:

- Operations with empty methods defined with required parameters and annotations
- Fully functional operations for a selected database table

Before you use the New Business Entity wizard to create operations with empty methods, you must define a data model.

Note: If the data model is a ProDataSet with multiple temp-tables, the New Business Entity wizard also defines a separate invoke operation for each temp-table.

Before you use the New Business Entity wizard to create fully functional operations for a selected table, you must start the database and ensure that it is connected to the ABL Web App project in Developer Studio.

Defining the data model

A Data Object resource can support a single data model (schema) that is managed by the standard operations. The data model can be a temp-table or a ProDataSet that contains one or more temp-tables, including any data relations defined for them.

When defining the data model, ensure that it meets the following requirements:

- The fields of the temp-tables do not use the RAW and RECID data types.
- If you want to use the Submit operation, use a ProDataSet and ensure that the temp-tables of the ProDataSet define before-tables to support before-imaging.
- If you want the hierarchy of the parent and child temp-tables of a ProDataSet to be retained, define the data relation between them with the NESTED option.

Using the New Business entity wizard

After you create your data model, you can create a Data Object interface using the New Business Entity wizard.

Follow these steps to create a Data Object interface using the New Business Entity wizard:

1. Open Progress Developer Studio for OpenEdge.
2. Right-click the ABL Web App project and select New > Business Entity. The New Business Entity wizard appears.
3. In the Create a Business Entity class page:
   a. Specify the Package root of the Business Entity class.
   b. Specify the Package.
   c. Enter the Business Entity name.

   d. If you want to use a temp-table to create a new Business Entity, clear the contents of the Inherits field.

   e. Click Next.

4. In the Select a schema file page, select the operation or the set of operations that must be created.
5. Select Write dataset before image if you want the before-image data to be written to a file. Note: The Write dataset before image option is selected by default if you select the CRUD and Submit options. The before-image data is required to perform the Submit operation.
6. If you want to create the Data Object interface for a database table, select the Select database table option and from the drop-down lists, select a Connection and a table.
7. If you want to create the Data Object interface for a schema file, select the Select schema from a file option and then browse to select the schema file. This populates the Schema box. Select the required schema.
8. After you select the schema, in the Using section:
9. Select Include file if the selected schema file is an include (.i) file. Note: The schema file is included in the Business Entity class. No annotations are generated for the temp-table defined in the selected file.
10. Select Schema definition to copy the selected schema definition to a Business Entity class file. This option is enabled if the selected schema is a procedure (.p or .w), a class (.cls), or an include (.i) file.
11. Note: If the schema definition option is selected and the selected file is an include file, the wizard adds the temp-table schema definition from the selected file to the new Business Entity class file along with annotations if the include file does not already have annotations. In such a case, the annotations are generated based on the temp-table definition. If the include file already has annotations, no other annotations are added.
12. Select Class hierarchy if you want to select the schema definition from the class hierarchy.
13. Ensure that Expose as a Data Object service is selected.
14. Optionally, modify the Resource URI. By default, the resource name is displayed with a “/” prefix.
15. Click Finish. A class file with the required methods and annotations, that is, a Data Object interface, is created.

Creating a Data Object service

After you create Data Object interfaces, you can start creating Data Object services using the New ABL Service wizard.

Before you can create Data Object services, you must ensure that the databases and the PAS for OpenEdge instance are running. Follow these steps to create a Data Object service:

1. Open Progress Developer Studio for OpenEdge.
2. Right-click the ABL Web App project for which you want to generate a Data Object service and select New > ABL Service. The New ABL Service wizard appears.
3. On the first page of the New ABL Service wizard:

   a. Select REST as the service type.

   b. Select Create a Data Object Service (Generate a JSDO catalog).

   c. Enter a name for the service. The Service relative URI field is populated.

   d. Optionally, enter a description in the Service description field.

4. On the Create a Data Object service page, select all the Data Object interfaces that you want to expose in the Data Object service and then click Finish. This creates a Data Object service.

Editing a Data Object service

After a Data Object service is created, the service is listed under Defined Services. The entry in the project explorer enables you to modify or delete a service.

If you select a PAS for OpenEdge instance when creating a Data Object service, the service is added to the PAS for OpenEdge instance in the Servers view. You can ensure that a service has been published by looking at the status of the service. When a service is published to the PAS for OpenEdge instance, the status [Synchronized][Published] is displayed.

The entry in the Servers view enables you to add or remove a service from the PAS for OpenEdge instance.

Testing Data Object Services using URIs

A Data Object service has a Data Service Catalog that defines the schema of the data and the operations on the data that are supported by one or more of the Data Object resources. Regardless of whether a Data Object service is WebHandler or REST-based, Data Object service clients can use the following URI syntax for locating resources: `scheme://host:port/web-app/static/service-name.json`

You should test Data Object services before you package them as an ABL Web Application. You can use various REST clients, such as the Postman REST Client, that allow you to access Data Object operations directly as REST resources. To test these services, you must know the absolute REST URI of the WebHandler or REST resource that accesses each Data Object operation.

The syntax for the absolute URI for WebHandler-based Data Object services is:

```
scheme://host:port/web-app/web/pdo/service-name/resource-name[op-element]
```

For example,
`scheme://host:port/web-app/web/pdo/service-name/resource-name?filter=filter-string` is the absolute URI to perform a Read operation using a filter.

The syntax for the absolute URI for REST-based services is:

```
scheme://host:port/web-app/rest/service-name/resource-name[op-element]
```

The following table lists each element in the syntax with its description:

Syntax element	Description
scheme	A URI scheme supported by Data Object services. It can be HTTP or HTTPS.
host	The host name or the IP address of the machine that contains the PAS for OpenEdge instance that hosts the Data Object services.
port	The port number on which the Web Server listens for HTTP requests. The default port number configured for PAS for OpenEdge is 8810.
web-app	The name of the Web Application that contains your Data Object services.
static-resource	The relative URI for a static file or Web page that the Mobile App must access from the Mobile Web Application.

Syntax element	Description
service-name	The relative URI for the Data Object service.
resource-name	The relative URI for the Data Object resource.
op-element	A relative URI or a URI query parameter that is specified to identify specific Data Object resource operations. The syntax is: ?filter=filter-string—For the read operation, this URI query parameter identifies the INPUT filter parameter value passed to the implementing ABL routine, where filter-string is a string value that you must specify. /operation-name—For an Invoke or Submit operation, operation-name by default is the case-sensitive name of the ABL routine that implements the specified Invoke or Submit operation. The operation-name can also be any other value that you annotate for the relative URI of an Invoke or Submit operation routine that uniquely identifies and distinguishes the routine from any other that is defined in the resource.

Updating Business Entities to use server-side preprocessing

Recall that the Read operation supports server-side preprocessing to filter data, sort data, or page through data.

If you are using a Kendo UI client and you want to use server-side preprocessing, you must modify your Business Entities to allow Kendo UI components to access OpenEdge data. To support server-side preprocessing, you must update the Business Entities with additional annotations and code to:

- Handle the filter parameter of the read method (corresponding to the Data Object Read operation) as an OpenEdge JSON Filter Pattern (JFP).
- Create a method that returns the total number of available records and expose it as a Data Object Invoke operation.

Updating a Business Entity to handle an OpenEdge JSON Filter Pattern

To handle the filter parameter of the Data Object Read operation method as an OpenEdge JSON Filter Pattern, you must add the following two additional annotations to the existing annotations of the Read operation.

```
@openapi.openedge.method.property (name="mappingType",
value="JFP").@openapi.openedge.method.property (name="capabilities",
value="ablFilter,top,skip,id,orderBy").
```

The JSON Filter Pattern is a JSON object that contains the following properties:

- ablFilter—Contains the text of an ABL string based on which the OpenEdge database query is filtered before returning the result set.Examples:
- “ablFilter” : “(State = ‘MA’) OR (State = ‘GA’)”“ablFilter” : “Name BEGINS ‘A’ ”
- orderBy—A comma-delimited list of the names of fields used to sort the data in the result set. By default, the result set is in ascending order. You can add desc after any field to indicate that its sort order is descending.Examples:“orderBy”: “Balance, State”—The data will be sorted in ascending order, first by

Balance, and then within Balance by State.“orderBy” : “Country, State desc, City”—The data will be sorted in ascending order by Country, then in descending order by State, and in ascending order by city.

- skip—Specifies the number of records to skip in the result set before returning a page of data in the result set. For example, if the requested page size is 10 and the request is for the 5th page of data, the value of the property is set to 40.
- top—Specifies the number of records to be returned in a single page, that is, the page size of the result set after using skip.

Both the skip and top properties are used for server paging.

You must add ABL code to handle these filter parameters and their behavior in the Read operation of the Business Entity.

Creating a Data Object Invoke operation method that returns the total number of available records

To use server-side paging, you must create an Invoke operation method that returns the total number of records in the server result set. The Invoke operation that returns the total number of records in the server result set must have the following ABL method signature:

[mouse] Roll over a syntax element to learn about it:

```
METHOD PUBLIC VOID countFnName (INPUT filter AS CHARACTER,OUTPUT numRecs AS INTEGER)
```

Where:

countFnName	The name of the method. You have to use this name when you use the Data Object service from Telerik Platform.
filter	A string that contains the same JSON Filter Pattern that is passed to the Read operation method.
numRecs	An integer that you assign to the total number of records in the server result set identified by the filter.

After you create an Invoke method with the required method signature, you must add ABL code that returns the number of records in the result set. Then, you must add annotations to the new Invoke method using the Define Service Interface wizard and replace the URI in the annotation with:

```
/countFnName?filter=~{filter~}
```

3

Agent and Session Management

This section describes how to write and use PAS for OpenEdge event procedures.

For details, see the following topics:

- Agents and sessions
- Agent procedures
- Session procedures
- Managing context for bound and unbound session-managed connections
- Managing client context for session-free and unbound session-managed connections

Agents and sessions

An ABL application in a PAS for OpenEdge instance consists of a session manager that manages multi-session agents. Each agent is an operating system process that launches an ABL Virtual Machine (AVM) and uses it to execute a client application request within a session.

You can write event procedures that are executed during agent or session events (for example, when an agent or session starts or shuts down). The main use of event procedures is to manage user context. Context is any data that needs to be persisted over several interactions between a client application and the service. This includes things like user profile, role information, language preferences, etc.

Agent procedures

There are two types of event procedures that you can write for a PAS for OpenEdge multi-session agent:

- Startup
- Shutdown

Multi-session Agent Startup procedure

PAS for OpenEdge uses a multi-session agent, which means that a single agent can concurrently handle requests from several clients, each in its own session, and each running its own application model. When a multi-session agent starts up, it runs any Agent Startup procedure that you have defined for the PAS for OpenEdge instance. This procedure can be used to perform tasks required by all server sessions that run in the PAS for OpenEdge instance. One common task is to create all self-service database connections that are shared by the server sessions created and managed by a given multi-session agent. Each multi-session agent is a user with its own self-service connection to a given database, and all of its server sessions share that same connection as separate users.

This Startup procedure takes a character string as an input parameter. For example:

```
DEFINE INPUT PARAMETER startup-data AS CHARACTER NO-UNDO.
```

The session that executes an Agent Startup procedure runs effectively session-free. Any global data that is set in this session expires when the session terminates and is not available to other sessions of the multi-session agent. You use the following properties in `openedge.properties` to set them: `agentStartupProc` and `agentStartupProcParam`.

Multi-session Agent Shutdown procedure

When a multi-session agent shuts down, it runs any Multi-session Agent Shutdown procedure that you have defined for the PAS for OpenEdge instance. This procedure can be used to perform tasks required when shutting down the agent and the server sessions that it manages. One common task is to shut down all self-service database connections that were started in the Multi-session Agent Startup procedure.

The session that executes an Agent Shutdown procedure runs effectively session-free. Global data that is set in this session expires when the session terminates and is not available to other sessions of the multi-session agent. You use the following property in `openedge.properties` to set them: `agentShutdownProc`.

Session procedures

There are six types of event procedures that you can write for sessions:

- Startup
- Shutdown
- Activate
- Deactivate
- Connect (session-managed connections only)
- Disconnect (session-managed connections only)

Session Startup procedure

A Session Startup procedure executes as an server session starts up. You can specify the name of an PAS for OpenEdge Session Startup procedure using the OpenEdge Explorer or OpenEdge Management or by setting the `sessionStartupProc` property in the PAS for OpenEdge properties file (`openedge.properties`).

This Startup procedure takes a character string as an input parameter. For example:

```
DEFINE INPUT PARAMETER startup-data AS CHARACTER NO-UNDO.
```

You can set the value for this parameter in the properties file using the `sessionStartupProcParam` property, and you can set it to any arbitrary value. If you do not specify a value for this property, the parameter is set to the Unknown value (`?`) when the PAS for OpenEdge session executes the Startup procedure.

The Session Startup procedure is automatically executed as a persistent procedure within each server session when the PAS for OpenEdge session first starts. The persistent procedure is automatically deleted just before the session terminates. If you do not need the context established by the Startup procedure, you can delete the persistent procedure within a server session context by executing the `DELETE PROCEDURE` (or `DELETE OBJECT`) statement using the appropriate procedure handle.

If the Session Startup procedure completes with no error, the PAS for OpenEdge starts up successfully. If the Startup procedure completes with an error return value (`RETURN ERROR` statement), PAS for OpenEdge instance startup fails and the PAS for OpenEdge session is terminated.

Session Shutdown procedure

A Session Shutdown procedure executes just before the server session is shut down. A Session Shutdown procedure takes no parameters. You can specify the name of a PAS for OpenEdge Session Shutdown procedure using the OpenEdge Explorer or OpenEdge Management or by setting the `sessionShutdownProc` property in the PAS for OpenEdge properties file (`openedge.properties`).

Unlike the Session Startup procedure, the Session Shutdown procedure is run as a non-persistent procedure, and any errors propagated by the Shutdown procedure are simply ignored. The PAS for OpenEdge session terminates immediately after the Shutdown procedure executes.

Activate procedure

The Activate procedure executes immediately before a remote procedure request when the connection is session-free or session-managed in the unbound state. A typical use of Activate procedures for unbound session-managed connections is to retrieve connection context using an application-specific context database or the SERVER-CONNECTION-CONTEXT attribute on the SESSION handle (see Managing context for bound and unbound session-managed connections on page 101). Using the SERVER-CONNECTION-ID attribute, you can uniquely identify (key) the context in the context database. You can then retrieve the connection-identified context from the context database or unmarshal it from the SERVER-CONNECTION-CONTEXT attribute value. You can create the initial context for the connection using the Connect procedure. For more information, see Connect procedure on page 99.

For either session-free or unbound session-managed connections, you can also retrieve client connection context that might be keyed on the value of the ClientContextId property of the Progress.Lang.OERequestInfo class. From the Activate (or any other PAS for OE instance) procedure, you can access this property on the OERequestInfo object that is returned by the CURRENT-REQUEST-INFO attribute on the SESSION handle. For more information, see Managing client context for session-free and unbound session-managed connections on page 103.

You can specify the name of an PAS for OpenEdge instance Activate procedure by setting the sessionActivateProc property in the PAS for OE properties file (openedge.properties). If you specify this procedure, it accepts no parameters and runs as a non-persistent procedure in any PAS for OpenEdge session that executes a remote procedure request for an session-free or unbound session-managed connection.

Note: The Activate procedure only runs prior to remote procedure requests. It does not run in response to connect or disconnect requests.

For the purposes of error handling, the Activate procedure is considered part of the remote procedure request. If the Activate procedure completes with no ABL termination condition (ERROR, QUIT, or STOP), the remote procedure executes immediately afterward. If the Activate procedure completes with a termination condition, it returns to the client application as if the remote procedure had generated the termination condition. Any return values (using the RETURN [ERROR] *string* statement) are passed to the client and the original remote procedure request is not executed.

Deactivate procedure

The Deactivate procedure executes immediately after remote procedure and delete procedure requests when the connection is in the session-free or session-managed in the unbound state. A typical use of Deactivate procedures for unbound session-managed connections is to store connection context to be saved from the session that just completed the current request using an application-specific context database or using the SERVER-CONNECTION-CONTEXT attribute on the SESSION handle. For more information, see Managing context for bound and unbound session-managed connections on page 101. Using the SERVER-CONNECTION-ID attribute, you can uniquely identify (key) the context in the context database. You can then gather the context identified with the connection and its just completed request and store it in the context database or marshal it into the SERVER-CONNECTION-CONTEXT attribute.

For either session-free or unbound session-managed connections, you can also store connection context keyed on the value of the ClientContextId property of the Progress.Lang.OERequestInfo class. From the Activate (or any other PAS for OE instance) procedure, you can access this property on the OERequestInfo object that is returned by the CURRENT-REQUEST-INFO attribute on the SESSION handle. For more information, see Managing client context for session-free and unbound session-managed connections on page 103.

You can specify the name of an PAS for OpenEdge Deactivate procedure by setting the `sessionDeactivateProc` property in the PAS for OpenEdge properties file (`openedge.properties`). If you specify this procedure, it accepts no parameters and runs as a non-persistent procedure after the request executes, but before any response is returned to the client. Also, the Deactivate procedure runs only if a session-managed connection is unbound when the request completes execution.

A session-managed connection can be in the unbound or the bound state when a request is initiated and can change to the opposite state before the request completes execution. If the connection ends up in the bound state during execution of the request, the Deactivate procedure does not run when the request completes. However, if a session-managed connection ends up in the unbound state, the Deactivate procedure runs even if the original request completes with some ABL termination condition (`ERROR`, `QUIT`, or `STOP`).

After a delete procedure request, the Deactivate procedure runs only if a session-managed connection is unbound. The connection can become unbound only if both of the following are true:

- The deleted procedure was the last instantiated remote persistent procedure for the connection
- The `SERVER-CONNECTION-BOUND-REQUEST` attribute has a value of `FALSE`

For more information on how session-managed connections transition between bound and unbound states, see Managing context for bound and unbound session-managed connections on page 101.

Note: The Deactivate procedure only runs following remote procedure and delete procedure requests. It does not run in response to new connection or disconnection requests.

For the purposes of error handling, only return values from the request (or any executed Activate procedure) are passed to the client. Any return values generated by the Deactivate procedure, itself, are ignored.

Connect procedure

A Connect procedure executes in whatever PAS for OpenEdge session handles the client connection request, before the connection request is accepted. If the Connect procedure completes with no error, the connection request from the client application is accepted. If the Connect procedure completes with an error return value (`RETURN ERROR` statement), the connection request is rejected.

The Connect procedure can also return a user-defined string to the client. This is accomplished by using either the `RETURN` *`string`* statement or the `RETURN ERROR` *`string`* statement in the connect procedure. The client can access this string if the Connect procedure executes and returns an error or if the Connect procedure succeeds. This feature allows you to return more detailed error messages or provide diagnostic information about a successful connection.

How the client can access the user-defined error depends on the client type:

- For ABL clients, see Connecting to a PAS for OpenEdge instance
- For SOAP Web Service clients, see *OpenEdge Development: Web Services*
- For .NET Open Clients, see *OpenEdge Development: .NET Open Clients*
- For Java Open Clients, see *OpenEdge Development: Java Open Clients*

You can specify the name of an PAS for OpenEdge Connect procedure by setting the `sessionConnectProc` property in the PAS for OpenEdge properties file (`openedge.properties`). If you specify this procedure, it runs as part of the connection process initiated when an ABL client application executes the `CONNECT( )` method on a server handle or when a Web Service or Open Client executes a corresponding connection method.

You must define the procedure with three character input parameters. The following is an example of how you can define these input parameters:

```
DEFINE INPUT PARAMETER cUserId        AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER cPassword      AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER cConnectInfo   AS CHARACTER NO-UNDO.
```

When the PAS for OpenEdge Connect procedure runs, the values of these three parameters are the same as the corresponding parameters specified via the `CONNECT( )` method. If a value is not specified for a parameter to the `CONNECT( )` method, the PAS for OpenEdge Connect procedure receives the `Unknown value (?)` for that parameter.

OpenEdge does not examine the values passed from the `CONNECT( )` method to an PAS for OpenEdge Connect procedure. The name of each character input parameter might suggest the content you define for it. For example, `cUserId` can contain user name information, `cPassword` can contain password information, and `cConnectInfo` can contain other user-defined information needed by the PAS for OpenEdge Connect procedure to complete a connection. However, the requirement to specify a particular value for the `CONNECT( )` method is based only on the implementation of your PAS for OpenEdge Connect procedure. After examining the values provided to the PAS for OpenEdge Connect procedure, you can reject the connection request from the client application by terminating the procedure with the `RETURN ERROR` statement.

You can use this procedure to authenticate a client application's request for a connection, or to perform any other initialization steps that you want executed each time a client application initiates an explicit connection request. For example, you typically begin by checking the `cAppServerInfo` procedure parameter to identify if this is to be a bound or unbound connection and set the `SERVER-CONNECTION-BOUND-REQUEST` attribute on the `SESSION` handle appropriately. You might then initialize context for the connection using an application-specific context database or using the `SERVER-CONNECTION-CONTEXT` attribute on the `SESSION` handle (see Managing context for bound and unbound session-managed connections on page 101). Using the `SERVER-CONNECTION-ID` attribute, you can uniquely identify the connection. You can then gather the context identified with the connection and store it in the context database or marshal it into the `SERVER-CONNECTION-CONTEXT` attribute. Any PAS for OpenEdge procedure that executes on behalf of the connection can restore and re-save this context as appropriate. .

Disconnect procedure

A Disconnect procedure executes in whatever PAS for OpenEdge session handles a client disconnection request, but before the connection is terminated. You can use the PAS for OE Disconnect procedure to specify logic that you want executed at this time. For example, you might want to use it to disconnect from a client-specific database or from other PAS for OpenEdge instances that were connected on behalf of the client.

You can specify the name of the PAS for OpenEdge Disconnect procedure by setting the `sessionDisconnProc` property in the PAS for OpenEdge properties file (`openedge.properties`). This procedure takes no parameters.

For a bound client connection, the Disconnect procedure runs after the PAS for OE instance returns the response to the disconnect request to the client application. For an unbound client connection, the procedure runs just before the PAS for OpenEdge returns this disconnect response, where it is typically used to delete connection-specific context. Any raised conditions that you do not handle in this procedure are ignored by OpenEdge.

Managing context for bound and unbound session-managed connections

On a bound session-managed connection, all remote procedures run in the session of a PAS for OpenEdge instance. The context at the initiation of a client request is exactly as it was when the previous request from the same client completed.

Unbound session-managed connections generally require more complex context management than for bound connections. The trade-off is potentially higher throughput for the effort. Because any ABL session in a PAS for OpenEdge session pool can handle any remote procedure request on any unbound connection, the context created by one remote procedure request is generally unavailable to the next request in the same connection sequence. To help you manage context for an unbound session-managed connection, OpenEdge provides the following ABL elements:

- `SERVER-CONNECTION-CONTEXT` attribute
- `SERVER-CONNECTION-ID` attribute
- `SERVER-CONNECTION-BOUND-REQUEST` and `SERVER-CONNECTION-BOUND` attributes

Other than in Using the SERVER-CONNECTION-BOUND attribute on page 103, most of the remaining information on context management applies only to session-managed connections in the unbound state.

To support context management for either session-free or unbound session-managed connections, OpenEdge also provides the `ClientContextId` property on the `Progress.Lang.OERequestInfo` class. For more information, see Managing client context for session-free and unbound session-managed connections on page 103.

Using the SERVER-CONNECTION-CONTEXT attribute

When a client application establishes a session-managed connection with a PAS for OE instance, the instance session manager creates a connection context area in memory for the connection. A PAS for OpenEdge procedure can access this context area using the `SERVER-CONNECTION-CONTEXT` attribute of the `SESSION` handle. This is a readable and writable attribute of type `CHARACTER`. The initial value of `SERVER-CONNECTION-CONTEXT` is the `Unknown value (?).`

When any remote procedure executes in a session over a session-managed connection, it can set the `SERVER-CONNECTION-CONTEXT` attribute to any application-specific value. Each time OpenEdge assigns a request from a session-managed client to a PAS for OpenEdge session, OpenEdge ensures that the `SERVER-CONNECTION-CONTEXT` attribute for that session is set to the last value assigned in a previous server session that serviced the same client connection. This value is also available to any Connect procedure, Activate procedure, Deactivate procedure, or Disconnect procedure that you configure for the PAS for OpenEdge instance. Thus, each ABL session that services a an unbound session-managed client connection can pass context information to the next.

Using the SERVER-CONNECTION-ID attribute

For more complex context than you might want to store in the `SERVER-CONNECTION-CONTEXT` attribute, you might choose to use a context database. In this case, you can use the `SERVER-CONNECTION-ID` attribute as a primary key to help you store and retrieve the context for each unbound session-managed connection. A context database is an application database that is connected to and accessible from each that services a session-managed client..

Because the SERVER-CONNECTION-ID is a globally unique ID, OpenEdge ensures that no two connections on a computer network have the same value. Before a session-managed request is executed within a client connection, OpenEdge also ensures that the SERVER-CONNECTION-ID is set to the ID for the connection on which the request was received. This value is also available to any Connect procedure, Activate procedure, Deactivate procedure, or Disconnect procedure that you configure for the PAS for OpenEdge instance.

Using the SERVER-CONNECTION-BOUND-REQUEST

A session-managed client connection to a PAS for OpenEdge instance is in one of two states:

- **Unbound** — Where remote procedure requests over the connection can be handled by any available PAS for OpenEdge session
- **Bound** — Where remote procedure requests over the connection are handled by a single PAS for OpenEdge session that is dedicated to the connection

By default, a session-managed connection is unbound.

Making a connection bound

An unbound connection can become bound in two ways:

- A client application can call a remote persistent procedure (see Programming ABL Client Applications) or instantiate an Open Client ProcObject, in the case of an Open Client or SOAP Web service client. This call runs and instantiates a persistent procedure in the PAS for OpenEdge session that handles the remote procedure request. The connection thus becomes bound, dedicating this session to handle all future client requests over the bound connection.

  Note: It is the successful instantiation of a remote persistent procedure that forces the connection to transition from the unbound state to the bound state. If a client fails in its attempt to instantiate a remote persistent procedure on an unbound connection, the connection remains unbound.

- A PAS for OpenEdge session handling a request from a client can run a procedure that sets the SERVER-CONNECTION-BOUND-REQUEST attribute on the SESSION handle to TRUE. The connection thus becomes bound, dedicating the session that sets this attribute to handle all future client requests over the bound connection.

Making a connection unbound

You can request that a bound connection become unbound by setting the SERVER-CONNECTION-BOUND-REQUEST attribute on the SESSION handle to FALSE. However, the connection only becomes unbound as long as there are no remote persistent procedures still active in the PAS for OpenEdge session.

Once a connection transitions from the bound to the unbound state, the PAS for OpenEdge session can handle the next request sent by any connected client, whether it is session-managed or session-free. Conversely, the next request sent by the client on the previously bound connection can be handled by any available PAS for OpenEdge session.

If remote persistent procedures are active on a bound connection, setting the SERVER-CONNECTION-BOUND-REQUEST attribute to FALSE has no effect until all remote persistent procedures in the bound session are explicitly deleted. When all remote persistent procedures are deleted, the connection either remains in the bound state or becomes unbound depending on the last setting of the SERVER-CONNECTION-BOUND-REQUEST attribute.

Using the SERVER-CONNECTION-BOUND attribute

A session-managed connection can be in a bound state because of active remote persistent procedures even if the SERVER-CONNECTION-BOUND-REQUEST attribute is set to FALSE. To easily determine if a connection is bound, even if the SERVER-CONNECTION-BOUND-REQUEST attribute is set to FALSE, check the value of the SERVER-CONNECTION-BOUND attribute.

When set to TRUE for a session running in a session-managed connection, the read-only SERVER-CONNECTION-BOUND attribute indicates that the connection is currently bound, either from having active remote persistent procedures or from having the SERVER-CONNECTION-BOUND-REQUEST attribute set to TRUE.

Managing client context for session-free and unbound session-managed connections

In the session-free or the (especially unbound) session-managed application models of the PAS for OpenEdge instance, any available PAS for OpenEdge session handles each remote request from a PAS for OpenEdge client. The session can use a *client context identifier* provided by OpenEdge as a key to retrieve a previously initialized client context from a context store in order to execute the request on behalf of the originating client. On completion of the request, the agent can then update the client context in the context store for access by the client when the request returns, or (acting as a client) the session can make its own request to another PAS for OE instance, passing the same client context identifier on behalf of the original client.. Sessions across one or more PAS for OpenEdge instances can thus use this same client context identifier to access the context store on behalf of the original client request..

OpenEdge generates an initial, globally unique value for this client context identifier when it first creates the server object that a client uses to connect to a PAS for OpenEdge instance. The client can then assign its own value for this identifier if necessary, possibly one that has been previously initialized for a different PAS for OpenEdge connection.

For example, this identifier is typically used to identify the current user login session that a client initially establishes through a PAS for OpenEdge instance or OpenEdge PAS for OpenEdge instance that implements the authentication service for the client. This value is then propagated in subsequent requests sent between the client and any OpenEdge Application Server involved in requests for that same login session.

Passing the client context identifier between client and server

OpenEdge provides a class, Progress.Lang.OERequestInfo, and its properties to hold the value of the client context identifier and related context information. OpenEdge provides instances of OERequestInfo for reference by attributes both on the server object handle that an ABL client uses to make remote requests, and on the SESSION system handle of the server session that processes the request. These attribute references include:

- On a client's server object handle (or a returned asynchronous request object handle):
  - REQUEST-INFO — Referencing context information that is sent with a client request to the server associated with this handle
  - RESPONSE-INFO — Referencing context information that is returned to the client with the response from the most recent request sent to the server associated with this handle

- On the server session's SESSION handle:
  - CURRENT-REQUEST-INFO — Referencing context information that is received from the current client request to the server. This is the same information that the client sends from the REQUEST-INFO attribute on the server object handle associated with the request.
  - CURRENT-RESPONSE-INFO — Referencing context information that is returned from the server in response to the current client request. This is the same information that the client receives in the RESPONSE-INFO attribute on the server object handle associated with the request.

The ClientContextId property on an OERequestInfo object holds the client context identifier value that is generated and passed between a client and server. Additional properties identify other information about the request, including the remote procedure call (ProcedureName), a unique ID that OpenEdge generates for each request (RequestId), and OpenEdge release information (VersionInfo) about the client that sends the request or the server that returns the response, depending on the handle and attribute that provides the information.

As noted previously, the ClientContextId property value typically serves as a primary key to the client context information maintained in a context store for a user login session, which might be initialized by an authentication service that handles the user login. Subsequently, when the client executes a remote procedure on the server, it again propagates the client context identifier (and related information) from the REQUEST-INFO attribute on the server handle, and the server session can then retrieve the identifier from its SESSION handle using the CURRENT-REQUEST-INFO attribute. Before the remote procedure returns, it can modify the context identifier referenced by the CURRENT-RESPONSE-INFO attribute on the SESSION handle, and the client can retrieve the updated identifier returned using the RESPONSE-INFO attribute on the server handle (or on the returned asynchronous request object handle for an asynchronous request).

For more information on these attributes and examples that access the ClientContextId property, see the reference entries for these attributes in *OpenEdge Development: ABL Reference*. For more information on using the ProcedureName property, see Accessing the name of the current remote procedure.

Generating a client context identifier

The client context identifier is typically a globally unique character string. For performance, the maximum length of the string should be less than 36 characters. The client context identifier can also be a blank ("") value or the Unknown value (?).

The following code fragment shows a recommended ABL assignment statement for generating a unique value for the client context identifier:

```
DEFINE VARIABLE ccid AS CHARACTER NO-UNDO.
ccid = SUBSTRING(BASE64-ENCODE (GENERATE-UUID), 1, 22).
```

This is only one of the methods you can use to generate a client context identifier, but it does generate a globally unique value.

The default method of generating the client context identifier is to allow the clients (ABL, Java, and .NET) to automatically generate the value and send it to the server. Optionally, the client or the server session can create their own client context identifier for advanced application logic.

For example, the client context identifier in an ABL session needs to be reset in order to indicate the end of the client user login session. This is typically done when the server invalidates the current client login session by invoking the `LOGOUT( )` method on the client-principal object that maintains the session, and by removing the client login context from the context store. The server then typically sets the `ClientContextId` property on the appropriate `Progress.Lang.OERequestInfo` class instance (available through the `CURRENT-RESPONSE-INFO` attribute on the `SESSION` handle) to the `Unknown value (?)`. This tells the client that its current login session has been terminated, and any subsequent (presumably invalid) requests from the client on the same server connection will contain a client context identifier with the `Unknown value (?)`. For more information on client-principal objects and using the `ClientContextId` property to manage context for user login sessions, see Implementing multi-tier security models in this manual and the sections on authenticating the user identity for multi-tier applications in *OpenEdge Development: Programming Interfaces*.

Managing context from the server with OERequestInfo objects

The entire application can manage the value of the client context identifier in the `ClientContextId` property. For example, a server session might control the client context identifier for a client using the `OERequestInfo` instance returned to the client in the server response, or act as a client to another server using the `OERequestInfo` instance sent to that other server in its own remote request.

In addition to the client context identifier, instances of `OERequestInfo` provide additional context information that depends on whether the client or server is originating the instance.

OpenEdge context information accessible by a server session provides:

- **OpenEdge version information for the current server session** — The server session can access the `LOCAL-VERSION-INFO` attribute on its `SESSION` system handle to find its OpenEdge Release information. This object reference attribute returns a `Progress.Lang.OEVersionInfo` class instance with `OEMajorVersion`, `OEMinorVersion`, and `OEMaintVersion` properties that provide this information. This Release information is also returned in the response to each client request.
- **Server version information returned to a client** — This identifies the OpenEdge Release information for the server returned to the client from the server session in response to the current client request. This is the same information provided to the agent using its own `LOCAL-VERSION-INFO` attribute. The server session provides this information to the client using the `VersionInfo` property on the `OERequestInfo` instance referenced by the `CURRENT-RESPONSE-INFO` attribute on the session's `SESSION` handle. The client can then access this information using the `RESPONSE-INFO` attribute on the server object handle it uses to make a remote request.
- **OpenEdge version information of the client application** — This identifies the OpenEdge Release of the client application that executed the current request. The session can read this information using the `VersionInfo` property on the `OERequestInfo` instance referenced by the `CURRENT-REQUEST-INFO` attribute on the agent's `SESSION` handle.
- **Client context identifier for the current request** — This is the value of the `ClientContextId` property on the `OERequestInfo` instance referenced by the server handle's `REQUEST-INFO` attribute when the client makes a remote request. The server session can retrieve this property value on the `OERequestInfo` instance referenced by the `CURRENT-REQUEST-INFO` attribute on its `SESSION` handle.
- **Identifier for the current request only** — The client automatically generates a globally unique identifier for the current request in the `RequestId` property on the `OERequestInfo` instance referenced by the server handle's `REQUEST-INFO` attribute. The server session can read this property value on the `OERequestInfo` instance referenced by its `SESSION` handle's `CURRENT-REQUEST-INFO` or `CURRENT-RESPONSE-INFO` attribute. This value is read-only and cannot be changed by any application code.
- **Client context identifier in response to the current request** — Before completing its current request, the server session handling the request can set a different client context identifier value as a part of its

response to the client by setting the `ClientContextId` property available through the `CURRENT-RESPONSE-INFO` attribute of its `SESSION` handle. Otherwise, by default, the existing property value available through its `CURRENT-REQUEST-INFO` attribute is used. The client can read this value after the remote request completes using the `RESPONSE-INFO` attribute on the server handle.

For more information on these class properties and handle attributes, see *OpenEdge Development: ABL Reference*.

For session-free connections, the typical way for both the client and server to manage context is for both the client and server to manage context. A client request can be executed, not only on any available session on the server, but on any one of many servers that support the business application for a single user login session. You can also use this same configuration for one or more servers that support a single user login session for an unbound session-managed application..

Because none of the context management attributes provided for a session-managed application work for a session-free application (except for the `SERVER-CONNECTION-BOUND` attribute), the context database must:

- Be shared by all server instances that support the user login session
- Use an entirely application-defined means (such as the `ClientContextId` property) to identify and distinguish user login session and to key the context database for this purpose

Together with this context database, you can use the Activate and Deactivate configuration procedures to restore context for every client request or provide the necessary context database management solely within each session handling the remote requests.

For more information on managing context from the ABL client, see Accessing client context regardless of application model.

4

Troubleshooting

This section covers how to troubleshoot problems in your ABL services by setting up ABL application logging.

For details, see the following topics:

- ABL application logging

ABL application logging

The OpenEdge Logger framework provides ABL classes and interfaces that enable you to implement logging in your ABL application in a configurable way. Using this framework, you define custom log messages at various severity levels in your ABL application code. When the code runs, the framework produces log events containing the defined log messages. The log messages are then formatted and written to log files based on logging configurations defined in a configuration file.

Because it uses configuration files, the Logger framework enables you to control the logging behavior (what gets written and where) without having to change your ABL application code.

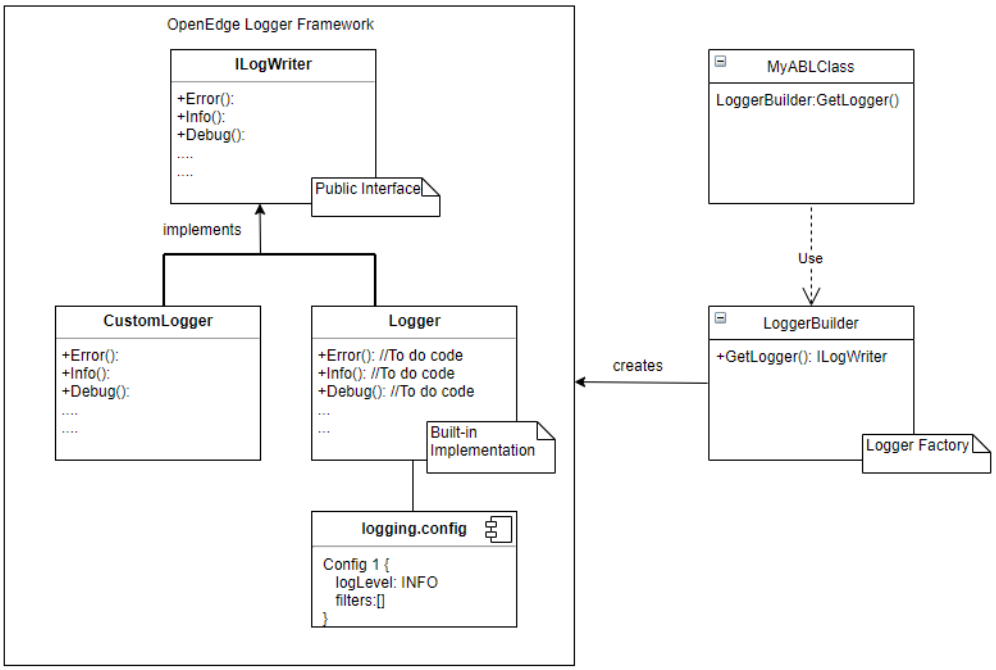
The Logger framework is modeled on the SLF4J pattern.

Key components

The OpenEdge Logger framework provides the following components:

- An ABL interface called `ILogWriter`. This interface provides methods like `Error()`, `Info()`, `Debug()`, etc, that you use to define log messages at various severity levels from within your ABL code.
- An ABL class called `Logger` that implements the `ILogWriter` interface. The `Logger` uses a configuration file named `logging.config` to process log events.

- A factory class called `LoggerBuilder` that enables you to instantiate the `Logger` from within your ABL code.
- An ABL class called `LogEvent`. A `LogEvent` object is created by the `Logger` when an `ILogWriter` interface method (`Error()`, `Info()`, etc) is invoked. The `LogEvent` object contains the log message, the timestamp of the event, etc.
- Filter classes that format and write log events. These filters are used by the `Logger` to process log events.
- Extension points that enable you to write custom filters.
- An enumerated type named `LogLevelEnum` that defines log levels (`OFF`, `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG`, `TRACE`). These log levels correspond to the `ILogWriter` interface's methods.



How it works

You instantiate the `Logger` through the `LoggerBuilder` and then use the `ILogWriter` interface's methods to define log messages at various severity levels in your ABL class. When your ABL class executes, a `LogEvent` object is produced for each log message defined in your code. These log events are processed by the `Logger` based on configurations defined in a `logging.config` file.

You must create the `logging.config` file manually and write the configurations in JSON. Each configuration has properties such as a name to identify itself, the log level (`INFO`, `DEBUG`, `TRACE`, etc), and references to filters that format and write log events.

By separating the configuration of logging from its implementation, you get the benefit of being able to modify logging behavior without having to modify ABL code. At runtime, an administrator can simply update configurations in the `logging.config` file to change log levels, specify log message formats, etc.

How to use it

To implement basic logging behavior, you need to perform the following tasks:

1. Define log messages in your ABL classes.
2. Write logging configurations in a `logging.config` file.

In addition, you can do the following:

- Set up a logger hierarchy, where a logging configuration applies to groups of ABL classes.
- Create a custom filter, if the built-in filters do not meet your requirements.

Define log messages in your ABL class

The OpenEdge Logger framework comprises a set of ABL classes and interfaces that are provided in the `OpenEdge.Logging` package in the **OpenEdge.Core.pl** procedure library, which gets added to an ABL application's PROPATH by default.

To define log messages, you mainly need the `ILogWriter` interface and the `LoggerBuilder` class. Use them as follows:

1. (Optional) Import the following packages in your ABL class:

```
using OpenEdge.Logging.ILogWriter.
using OpenEdge.Logging.LoggerBuilder.
```

2. Define a variable for the `ILogWriter` interface:

```
define variable logger as ILogWriter no-undo.
```

3. Use the LoggerBuilder's `GetLogger()` factory method to instantiate the `Logger` by passing the name of a logging configuration:

```
logger = LoggerBuilder:GetLogger("my.ABL.Class").
```

   The name that you specify here (`my.ABL.Class` in this example) should match the name of a logging configuration in the `logging.config` file. This instructs the `LoggerBuilder` to use that configuration to when instantiating the `Logger`. It is a good practice to use the fully qualified name of your ABL class as the logging configuration name (in the `logging.config` file and also when instantiating the Logger). This makes it easy for administrators to know which configuration to modify to control logging for an ABL class. In addition, this may enable the configuration of a logger hierarchy in the `logging.config` file. To learn more, see Set up a logger hierarchy.

   Note: If the `LoggerBuilder` does not find a matching logging configuration in the `logging.config` file, it either instantiates a `VOID` logger (which acts as a sink for log events), or writes events to the `Log-Manager`, depending on the client type.

4. Write log messages at different severity levels using the following interface methods:

```
logger:Fatal()
logger:Error()
logger:Warn()
logger:Info()
logger:Debug()
logger:Trace()
```

   These methods correspond to log levels defined in the `LogLevelEnum` enumerated type. See Log levels for guidelines on how to use these log levels. When the ABL code runs, a `LogEvent` object is created for each of these methods. They are processed based on the log level set in the logging configuration. To know more, see Write logging configurations.

   Each of the interface methods has two signature styles—string and object. They are described below.

String style

The string style signature takes the following parameters:

Parameter	Description	Required?
CHARACTER *MessageGroup*	A name of a message group.	No
CHARACTER *Message*	The log message that needs to be written.	Yes
Progress.Lang.Error *ErrorObject*	An error object.	No

Examples:

```
logger:Info("Customer record found").
logger:Warn("CUSTOMER-MESSAGES", "Customer record missing").
logger:Error("An exception occurred", ErrorObject)
```

Object style

The Logger wraps the string style message in a LogMessage object when it produces a log event, but you can create your own LogMessage object as well. The main reason for doing this is if you want to pass arguments in the message that get their values from elements in an array. To specify a LogMessage object, you need to use the object style signature, which takes the following parameters:

Parameter	Description	Required?
OpenEdge.Logging.LogMessage. *LogMessageObject*	A LogMessage object that contains a group name, message, and an array of character arguments.	Yes
Progress.Lang.Error *ErrorObject*	An error object.	No

How to create a LogMessage object

Create a LogMessage object using its class constructor, which accepts the following parameters:

Parameter	Description
CHARACTER *GroupName*	A name of a message group.
CHARACTER *BaseText*	A string that can contain arguments denoted by &1, &2, &3.. etc, whose values are retrieved from elements in an array. Up to 9 arguments can be specified.
CHARACTER EXTENT *ArgumentArray*	A character array. The value of elements in the array will substitute the arguments in the BaseText parameter. You can either specify an array name or specify a comma-separated list of strings.

Example:

```
using OpenEdge.Logging.LogMessage.

...
...

define variable cArgs as character extent 3.

...
...
cArgs[1] = "3".
cArgs[2] = "Hoops".
cArgs[3] = "Atlanta".
...
...
logger:Info(new LogMessage("CUSTOMER-MESSAGES", "Customer found. ID: &1. Name: &2. City:
 &3.", cArgs)).
```

Alternatively, you can supply the substitution values as follows:

```
logger:Info(new LogMessage("CUSTOMER-MESSAGES", "Customer found. ID: &1. Name: &2. City:
 &3.", "3", "Hoops", "Atlanta")).
```

This will produce an output log message that looks like this:

```
"Customer found. ID: 3. Name: Hoops. City: Atlanta."
```

Note: For this to work, you need to use an `ABL_SUBSTITUTE_FORMAT` filter in the logging configuration. See Log filters to learn more.

Write logging configurations in a logging.config file

Log messages that are defined in an ABL class are processed by the `Logger` based on configurations in a `logging.config` file.

Where to put the `logging.config` file

You need to create the `logging.config` file manually and write logging configurations in it in JSON format. The file should be placed in the ABL application's PROPATH. The `Logger` uses only the first `logging.config` file that it finds in the ABL application's PROPATH, so logging configurations for all ABL classes in an ABL application must be written in the same `logging.config` file.

Structure of the `logging.config` file

The `logging.config` file must begin with a JSON object named `logger`. Under this root object, you can create as many logging configurations as you need.

```
{
  "logger": {
      "my.ABL.class.1": {},
      "my.ABL.class.2": {},
      "my.ABL.class.3": {}
  }
}
```

Each logging configuration has a name (`my.ABL.class.1`, `my.ABL.class.2`,... in this example). This name is used when instantiating the `Logger` using the `LoggerBuilder:GetLogger()` method in an ABL class.

Inside each logging configuration, you define two things—a `logLevel` and an array of `filters`, as shown in this example:

```
{
  "logger": {
      "my.ABL.class.1": {
          "logLevel": "INFO",
          "filters": [
             "FULL_TEXT_FORMAT",
             {
               "name": "NAMED_FILE_WRITER",
               "fileName": "Class1.log",
               "appendTo": true
             }
          ]
      },
      "my.ABL.class.2": {...},
      "my.ABL.class.3": {...}
  }
}
```

To learn more about log levels, see Log levels. To learn more about filters, see Log filters.

Logging behavior when no `logging.config` file is found

If no `logging.config` file is found, the Logger framework acts as follows:

- For applications where the `Log-Manager` is in use, such as applications deployed on PAS for OpenEdge, the Logger framework builds a logger with the following filters:
  - `STACK_WRITER_FORMAT`
  - `ABL_SUBSTITUTE_FORMAT`
  - `ERROR_FORMAT`
  - `LOG_MANAGER_FORMAT`
  - `LOG_MANAGER_WRITER`

    For a description of each of these filters, see Log filters.

- For applications where the `Log-Manager` is not in use, the Logger framework builds a logger with a `VOID_WRITER` filter that acts as a sink for log events.

Log filters

Filters process log events that satisfy the log level defined in the same configuration. The OpenEdge Logger framework provides a number of built-in filters. They fall into two broad categories:

- Format filters that format the log message. You can define multiple format filters in the same logging configuration. In this case, the filters are applied to each log event sequentially, in the order in which you define them.
- Writer filters that write the log message to a file or console. Just like format filters, you can define multiple writer filters in the same logging configuration. In this case, the log message gets written by each writer filter.

You must use at least one format filter before a writer filter. Here is a list of built-in filters for each category:

Format Filters

ABL_SUBSTITUTE_FORMAT

Use this filter if the log event's LogMessage object contains arguments whose values are retrieved from elements in an array. Specify the filter name as shown in this example. No additional properties need to be set.

```
"filters": [
   "ABL_SUBSTITUTE_FORMAT",
   ...
   ...
]
```

LOG_MANAGER_FORMAT

Use this filter if you want the same log message format that is used in the Log-Manager. In this format, the log level is prefixed in the log message. Specify the filter name as shown in this example. No additional properties need to be set.

```
"filters": [
    "LOG_MANAGER_FORMAT",
    ...
    ...
]
```

STACK_WRITER_FORMAT

This filter applies to log events produced by the Debug() method. The log event that is produced includes a property named CallStack. This property (populated only for log events produced by Debug()) contains log stack information about the classes and methods that led to the log event. Specify this filter as shown in this example. No additional properties need to be set.

```
"filters": [
    "STACK_WRITER_FORMAT",
    ...
    ...
]
```

Note: In addition to log events produced by the Debug() method, the STACK_WRITER_FORMAT gets used if the debug-alert flag on the session handle is set to true (SESSION:DEBUG-ALERT = TRUE).

FULL_TEXT_FORMAT

This filter produces a predefined format that includes the timestamp, message group (if available), log level, the logging configuration name, and the log message. Here is a sample output:

```
[2018-11-02T14:20:29.021-04:00] CUSTOMER-MESSAGE-GROUP INFO: This is a test
message
```

Specify this filter as shown in this example. No additional properties need to be set.

```
"filters": [
    "FULL_TEXT_FORMAT",
    ...
    ...
]
```

ERROR_FORMAT

This filter applies only to log events produced by the `Error()` method. The log event that is produced includes a property named `Error`. This property (populated only for log events produced by `Error()`), is of the type `Progress.Lang.Error` and contains a description of the error.

Note: Using the `ERROR_FORMAT` filter sets the session's `error-stack-trace` property to `true`.

Specify this filter as shown in this example. No additional properties need to be set.

```
"filters": [
    "ERROR_FORMAT",
    ...
    ...
]
```

`TOKEN_FORMAT`

Use this filter if you want to use tokens in the log message. A token is like a placeholder or a variable name, whose value is 'resolved' at runtime. For example, if you use the token `${t.now}`, the `TOKEN_FORMAT` filter resolves the token to the current timestamp (the time at which the filter was applied). You cannot add tokens while defining log messages in the ABL class. You can, however, define a token in the `TOKEN_FORMAT` filter as shown in this example.

```
"filters": [
    {
       "name": "TOKEN_FORMAT",
       "format": "${t.now} ${msg.level} ${msg}"
    }
]
```

The `msg` token gets its value from the log message defined in the ABL class. The output looks something like this:

```
2018-11-05T14:00:07.054-05:00 INFO Customer found.
```

If the token cannot be resolved, the Logger framework prints the token as is. This also means that you can specify any custom strings, such as XML tags, in the token format. Here is an example:

```
"filters": [
    {
       "name": "TOKEN_FORMAT",
       "format": "<LogEntry>${t.now} ${msg.notARealProperty}</LogEntry>"
    }
]
```

The output, based on this format, would look like this:

```
<LogEntry>2018-11-05T14:00:07.054-05:00 ${msg.notARealProperty}</LogEntry>
```

A complete list of all tokens is described in the Logging Tokens section.

Writer filters

LOG_MANAGER_WRITER

Use this filter to write log messages to the log files used by the ABL Log-Manager (typically the PAS for OpenEdge agent log). Specify this filter as shown in this example.

```
"filters": [
    "FULL_TEXT_FORMAT",
    "LOG_MANAGER_WRITER"
]
```

NAMED_FILE_WRITER

Use this filter to specify the name of a log file to write the log message to. Specify this filter as shown in this example.

```
"filters": [
    "FULL_TEXT_FORMAT",
    {
       "name": "NAMED_FILE_WRITER",
       "fileName": "CustomerLog.log",
       "appendTo": true
    }
]
```

You use the fileName property to set the log file name. By default, if no path is specified, as in this example, the log file gets created in the PAS for OpenEdge instance's work directory. You can also use tokens in the filename to set up rotating log files. Here is an example:

```
"filters": [
    "FULL_TEXT_FORMAT",
    {
       "name": "NAMED_FILE_WRITER",
       "fileName": "${session.temp-dir}/${t.today}-CustomerLog.log",
       "appendTo": true
    }
]
```

The ${session.temp-dir} token is substituted by the pathname of the temp directory used by PAS for OpenEdge at runtime. The ${t.today} token is replaced by the date.

VOID_WRITER

Use this filter if you want to turn off writing log messages. The VOID_WRITER filter acts as a sink for the LogEvent object.

DEFAULT_WRITER

Use this filter to use the default logging system. For PAS for OpenEdge, this resolves to the LOG_MANAGER_WRITER. For other client types, it resolves to VOID_WRITER.

Log events

The OpenEdge Logger framework creates a `LogEvent` object for each log message defined using `ILogWriter` interface methods like `Info()`, `Warn()`, `Error()`, etc. Each `LogEvent` object has the following properties:

Property	Type	Required?	Description
`Logger`	`OpenEdge.Logging.ILogWriter`	No	The name of the logging configuration.
`LogLevel`	`OpenEdge.Logging.LogLevelEnum`	Yes	The log level of the event, as defined in the `ILogWriter` interface method.
`TimeStamp`	`datetime-tz`	No	The timestamp of the event.
`Message`	`OpenEdge.Logging.LogMessage`	Yes	Log message object containing the log message.
`Error`	`Progress.Lang.Error`	No	Errors, if any are thrown.
`CallStack`	`Character array`	No	The current stack trace of where the log event occurred. This property is populated only when the log level is DEBUG or TRACE.
`LoggerBy`	`handle`	No	The user logging this event. This is obtained from the `SECURITY-POLICY` system handle.
`LoggerShortName`	`character`	No	The short-name of the logging configuration that is responsible for processing the log event. The short name is the logger name with the `ShortNameFormat` applied.
`ShortNameFormat`	`character`	No	The short name format of the logging configuration that is responsible for processing the log event.

Log levels

Log levels in the Logger framework are defined in the `OpenEdge.Logging.LogLevelEnum` enumeration type. The `ILogWriter` interface methods (`Error()`, `Info()`, etc) as well as the log levels that you set in logging configurations in the `logging.config` file correspond to these log levels.

When you are defining log messages in an ABL class using these levels, it may be helpful to keep the following guidelines in mind:

Log level	Guidelines for usage
FATAL	Severe errors that cause premature termination. These should be immediately visible on a status console.
ERROR	Other runtime errors or unexpected conditions. These should be immediately visible on a status console.
WARN	Use of deprecated APIs, events that are 'almost' errors, other runtime situations that are undesirable or unexpected, but not necessarily wrong. These should be immediately visible on a status console.
INFO	Interesting runtime events such as startup or shutdown.
DEBUG	Detailed information on the flow through the system. These should be written to log files only.
TRACE	Most detailed information. These should be written to log files only.

At runtime, a log event (a `LogEvent` object) is produced for each `ILogWriter` interface method used in the ABL class. These events are processed based on the log level that is set in the logging configuration. Log levels are arranged in a hierarchy, starting from FATAL (the lowest level) to TRACE (the highest level). When you set a log level, all log events leading up to that level are processed. For example, if you set the log level to INFO, log events produced by `Fatal()`, `Error()`, `Warn()`, and `Info()` methods get processed.

The log levels work as follows:

Log Level	What log events gets processed
FATAL	`Fatal()` events only
ERROR	`Error()` and `Fatal()` events
WARN	`Warn()`, `Error()`, and `Fatal()` events
INFO	`Info()`, `Warn()`, `Error()`, and `Fatal()` events
DEBUG	`Debug()`, `Info()`, `Warn()`, `Error()`, and `Fatal()` events
TRACE	All events

Logging tokens

When you use the `TOKEN_FORMAT` filter in a logging configuration, you can supply tokens whose values are resolved at runtime (when a `LogEvent` object is processed). The resolved values are then used in the log message that is written to a log file. Here is an example.

```
"filters": [
    {
       "name": "TOKEN_FORMAT",
       "format": "${t.now} ${msg.level} ${msg}"
    }
]
```

The token has the syntax `${<token>}`. Some tokens have a group and an argument, separated by a period "`.`". For example, the group `t`(short for time), has arguments like `today` (the current date), `now` (the current timestamp), etc.

Here is a list of all available tokens that you can use:

Group	Argument	Description
msg		The log message defined in an ILogWriter interface method and produced in the LogEvent.
	level	The log level (ERROR, WARN, INFO, DEBUG, etc).
	logger	The logging configuration name.
session	Any readable attribute of the session handle	You can specify any readable attribute of the session system handle as an argument. For example, ${session.temp-dir}, ${session.client-type}, etc. Refer to the Session system handle documentation for a list of these attributes.
env	Any environment variable	You can specify any environment variable that is available via the OS-GETENV() ABL function. For example, ${env.path} would resolve to a comma-separated list of the pathnames set in the path environment variable.
ver	major	The major version of PAS for OpenEdge. For example, if you are using version 12.0.1, this will return 12.
	minor	The minor version of PAS for OpenEdge. For example, if you are using version 12.0.1, for example, this will return 0.
	maint	The service pack number of PAS for OpenEdge. For example, if you are using 12.0.1, for example, this will return 1.
t	today	The current date in ISO-DATE DATE format. For example, 2018-11-05.
	now	The current timestamp in ISO-DATE DATETIME-TZ format. For example, 2018-11-05T14:00:07.054-05:00.
	YYYY	The current year, including the century. For example, 2018.
	YY	The current year, without the century. For example, 18 (for 2018).
	BB	The full name of the current month. For example, November.
	B	The shortened name of the current month. For example, Nov.
	MM	The integer value of the current month with a leading 0, if required. For example, 01 (for January).
	M	The integer value of the current month with no leading 0. For example, 1 (for January).
	DD	The integer value of the current day of the month, with a leading 0, if required. For example, 05 (for the 5th day of the current month).
	D	The integer value of the current day of the month, with no leading 0. For example, 5 (for the 5th day of the current month).
	HH	The integer value of the current hour, in 24-hour clock format. For example, 18 (for a time between 6 PM and 7 PM).
	H	The integer value of the current hour, in 12-hour clock format. For example, 6 (for a time between 6 AM and 7 AM or between 6 PM and 7 PM).
	MMM	The current minute value in the timestamp, with a leading 0.
	SS	The current seconds value in the timestamp, with a leading 0.

Group	Argument	Description
	SSS	The millisecond value in the current timestamp, with a leading 0.
	Z	The timezone, based on the current session, with a leading +/-.
	PP	AM or PM.
	P	A for AM or P for PM.
	AA	The full name of the day of the week. For example, Tuesday.
	A	The shortened name of the day of the week. For example, Tue.
	W	The integer day of the week, starting from Sunday. If the day of the week is Tuesday, this argument returns 3.
req	tpt	The transport type for the request (REST, WEB, or APSV).
	ccid	The client context ID.
	id	The current request ID.
	session	The current session ID.
	thread	The current thread ID.
web	webapp	The web app name. (Valid only if the session client-type is WebSpeed or Multi-Session-Agent.)
cp	uid	The current user ID, obtained from the Client-Principal.
	quid	The current qualified user ID, obtained from the Client-Principal.
	domain	The domain name of the current user, obtained from the Client-Principal.

Group	Argument	Description
name	[right-segment-index] [case-expr].<Dot. Separated. String>	The ${name.[right-segment-index][case-expression].<String>} token enables you to shorten strings based on the dot "." separator in the string. For instance, you can trim a logging configuration name like Company.Order.Sales to C.O.S. Each segment in the <string>, can be trimmed by specifying a right-segment-index. All segments, except for those specified in the right-segment-index, get trimmed. The case (uppercase, lowercase, etc) for every trimmed segment can be changed using the case-expression. Specify an integer value in the right-segment-index to define how many segments, starting from the right, are retained in their original form, while others are trimmed. For example: • 0—all segments of the dot-separated string are trimmed. Company.Order.Sales becomes C.O.S. • 1—only the last segment of the string is retained in its original form. All other segments are trimmed. Company.Order.Sales becomes C.O.Sales. • 2—all segments except for the last two are trimmed. Company.Order.Sales becomes C.Order.Sales. And so on. Specify a case expression that applies to the trimmed segments as follows: • U—uppercase • L—lowercase • K—retains the original case of the trimmed segments • C—camel case; if any of the trimmed segments has a camel case, each uppercased letter gets concatenated. For example, OpenEdge becomes OE. **Examples**: ${name.1L.Company.Order.Sales} becomes c.o.Sales. ${name.0C.Company.Order.OrderLine} becomes C.O.OL.

Set a default logging configuration

You can set a default logging configuration in the `logging.config` file. When the Logger framework is unable to find a matching name or a parent configuration[8] for a configuration specified in the `LoggerBuilder:GetLogger()` method, it uses the default logging configuration to process log events. Use the `DEFAULT_LOGGER` property to specify a default logging configuration as shown in this example:

```
{
  "DEFAULT_LOGGER": "my.ABL.class.1",
  "logger": {
      "my.ABL.class.1": {
          "logLevel": "INFO",
          "filters": [
             "FULL_TEXT_FORMAT",
             {
               "name": "NAMED_FILE_WRITER",
               "fileName": "Class1.log",
               "appendTo": true
             }
          ]
      },
      "my.ABL.class.2": {...},
      "my.ABL.class.3": {...}
  }
}
```

Set up a logger hierarchy

You can set up a logger hierarchy within a `logging.config` file using dotted strings. The Logger framework interprets a string like `Company.Order` as being the parent of `Company.Order.Sales`, `Company.Order.Customer`, etc. This enables you to set up a logger hierarchy with logging configurations that apply to groups of ABL classes.

For example, assume that you have the following ABL classes:

- `Company.Order.Sales`
- `Company.Order.Customer`
- `Company.Internal.Employee`
- `Company.Internal.Finance`

When you instantiate the Logger, you typically pass the fully qualified class name in the `LoggerBuilder` (for example, `LoggerBuilder:GetLogger('Company.Order.Sales')`).

In the `logging.config` file, instead of specifying a logging configuration with a matching name for each of these classes, you could specify logging configurations named `Company.Order` and `Company.Internal`. The Logger framework determines a parent-child relationship and uses the `Company.Order` logging configuration for the `Sales` and `Customer` classes and `Company.Internal` for the `Employee` and `Finance` classes.

[8] You can set up a logger hierarchy based on dotted strings. See Set up a logger hierarchy

Create a custom filter

If the built-in filters do not meet your requirements, you can extend the OpenEdge Logger framework by writing your own custom filter.

Overview of steps to create a custom filter

Follow these steps to create a custom filter:

1. Create an ABL class that implements the `ILoggerFilter` interface.
2. Register the filter using the `LoggerFilterRegistry`.

Note: If the custom filter requires additional properties to be set in the `logging.config` file, you need to perform additional steps. See Create a custom filter with additional properties.

Implement the `ILoggerFilter` interface

Create an ABL class that implements the `OpenEdge.Logging.Filter.ILoggerFilter` interface. This interface provides a method named `ExecuteFilter()` that takes a `LogEvent` object as an input parameter.

Override this method and write your custom filter code inside it. Typically, you will want to manipulate the `LogEvent` object's properties. In the following example, the custom filter code retrieves the log message from the `LogEvent` object's `Message` property and reverses the order of the words in the message:

```
using OpenEdge.Logging.Filter.ILoggerFilter.
using OpenEdge.Logging.LogEvent.
using OpenEdge.Core.StringConstant.

class Filters.ReverseWordsFormat implements ILoggerFilter:

    method public void ExecuteFilter(input poEvent as LogEvent):

        define variable loop as integer no-undo.
        define variable cnt as integer no-undo.
        define variable newMessage as character no-undo.

    //Get the number of words in the log message
        assign cnt = num-entries(poEvent:Message:Message, StringConstant:SPACE).

    //Reverse the order of the words and create a new message string
        do loop = cnt to 1 by -1:
            assign newMessage = newMessage
                              + entry(loop, poEvent:Message:Message,
StringConstant:SPACE)
                              + StringConstant:SPACE.
        end.

    //Copy the reversed message string to the LogEvent's message property
        assign poEvent:Message:Message = newMessage.
    end method.

end class.
```

To learn more about properties of the `LogEvent` object, see Log events.

Register the custom filter using the `LoggerFilterRegistry`

After you have written your custom filter class, you need to register it using the `OpenEdge.Logging.LoggerFilterRegistry` class. You should do this early in your application code. For

example, if your application is to be deployed to a PAS for OpenEdge instance, the code that registers your filter class should be put in a session startup procedure.

You register a custom filter using the `LoggerFilterRegistry`'s `Put()` method, which takes two parameters:

- A name (a character data type) for the filter. This is the filter name that will be used in the `logging.config` file.
- A reference to the custom filter class.

Here is an example:

```
LoggerFilterRegistry:Registry:Put("reversed_words_filter",
get-class(Filters.ReverseWordsFormat)).
```

The `LoggerFilterRegistry` binds the filter name with the custom filter class. You can then use the filter name in a `logging.config` file, just like any other filter.

Example: Using the custom filter

Building on the previous examples, here is how you would use the custom filter using the filter name (`reversed_words_filter` in our example) in a `logging.config` file:

```
{
   "logger": {
      "Company.Order.Sales": {
         "logLevel": INFO,
         "filters": [
             "reversed_words_filter",
     {
                "name": "NAMED_FILE_WRITER",
                "fileName": "${msg.logger}-${today}.log"
                "appendTo": true
             }
         ]
      }
   }
}
```

For a log message produced from a statement like `logger:Info("Customer found.")`, the formatted log output would be "`found. Customer`".

Create a custom filter with additional properties

If your custom filter requires additional filter properties that need to be set in the `logging.config` file, you need to perform some extra steps. In addition to writing your custom filter class and registering it using the `LoggerFilterRegistry`, you need to:

1. Create a custom filter builder that inherits the `OpenEdge.Logging.Filter.LogFilterBuilder` class.
2. Register the filter builder using the `LogFilterBuilderRegistry`.

The following example creates a custom filter that translates a log message into another language via a REST API. For this example, assume that the filter requires three properties to be set in the `logging.config` file:

- `toLang`—the language that the message is to be translated in.
- `serviceURI`—the URI of the REST service that performs the translation.
- `apiKey`—an API key required by the translation service.

Create the custom filter

The first step is to create the custom filter class and have it implement the `ILoggerFilter` interface. The ABL code that executes the filter's operations must be put inside the interface's `ExecuteFilter()` method. This step needs to be performed for all custom filters, regardless of whether the filter requires additional properties.

In the following example, the filter class (`TranslatedMessageFormat`) contains three properties—the translation service URI, the api key, and the language to translate the message to. These properties are used in the `ExecuteFilter()` method, along with the log message, to build an HTTP request that is sent to the translation service. The actual values for these properties are to be obtained from configurations in the `logging.config` file.

```
class Example.Filters.TranslatedMessageFormat implements ILoggerFilter:

    // The translation service URI
    define public property TranslationService as URI no-undo
        get.
        set.

    // The API key for making requests
    define public property ApiKey as character no-undo
        get.
        set.

    // The language to translate the message to
    define public property To as character no-undo
        get.
        set.

    // reusable HTTP client
    define variable httpClient as IHttpClient no-undo.

    constructor public TranslatedMessageFormat():
        assign httpClient = ClientBuilder:Build():Client.
    end constructor.

    method public void ExecuteFilter( input poEvent as LogEvent):
        define variable req as IHttpRequest no-undo.
        define variable resp as IHttpResponse no-undo.
        define variable msg as JsonArray no-undo.
        define variable txt as JsonObject no-undo.
        define variable transArray as JsonArray no-undo.
        define variable translation as JsonObject no-undo.
        define variable loop as integer no-undo.
        define variable cnt as integer no-undo.

        TranslationService:AddQuery('to':u, this-object:To).

        assign req = RequestBuilder:Post(TranslationService)
                        :AddHeader('Ocp-Apim-Subscription-Key':u, ApiKey)
                        :AcceptJson()
                        :ContentType('application/json':u)
                        :Request.

        assign msg        = new JsonArray()
               req:Entity = msg
               txt        = new JsonObject()
               .

        txt:Add('Text':u, poEvent:Message:Message).
        msg:Add(txt).

        assign resp = httpClient:Execute(req)
               msg  = cast(resp:Entity, JsonArray)

               transArray = msg:GetJsonObject(1):GetJsonArray('translations':u)
```

```
                cnt        = transArray:Length
                .
        do loop = 1 to cnt:
            assign txt = transArray:GetJsonObject(loop).
            if txt:GetCharacter('to':u) eq this-object:To then
            do:
                assign poEvent:Message:Message = txt:GetCharacter('text':u).
                leave.
            end.
        end.
    end method.

end class.
```

Register the custom filter

The next step is to register the custom filter using the `LoggerFilterRegistry`.

```
LoggerFilterRegistry:Registry:Put('translation_filter':u,
      get-class(Example.Filters.TranslatedMessageFormat))
```

The filter name, in this example, is `translation_filter`. This is the name that will be used in the `logging.config` file.

Create a custom filter builder

The properties defined in the custom filter class need to be bound to properties in the `logging.config` file. To do this, you need to create a custom filter builder that inherits the `OpenEdge.Logging.Filter.LogFilterBuilder` class and override its `NewFilter()` method. The method must return an `ILoggerFilter` object.

In this example, a `JsonObject` variable named `options` is used to map the properties defined in the custom filter class (`To`, `ApiKey`, `TranslationService`), with logger properties (`toLang`, `apiKey`, `serviceURI`) that will be used in a logging configuration in the `logging.config` file.

```
class Example.Builders.TranslatedMessageFormatBuilder inherits LogFilterBuilder:
    /* Constructor */
    constructor public TranslatedMessageFormatBuilder(input filterName as character):
        super(filterName).
    end method.

    /* Creates the instance.

       @return ILoggerFilter A filter instance  */
    method override protected ILoggerFilter NewFilter(  ):

        define variable logFilter as ILoggerFilter no-undo.
        define variable tmf as TranslatedMessageFormat no-undo.
        define variable options as JsonObject no-undo.

        // Use the default
        assign logFilter = super:NewFilter().

        Assert:IsType(logFilter, get-class(TranslatedMessageFormat)).

        assign tmf     = cast(logFilter, TranslatedMessageFormat)
               options = cast(GetOptionObjectValue('loggerOptions':u), JsonObject)
               //Map filter class properties to logging configuration properties
              tmf:To                 = options:GetCharacter('toLang':u)
               tmf:ApiKey             = options:GetCharacter('apiKey':u)
              tmf:TranslationService = URI:Parse(options:GetCharacter('serviceURI':u))

               .
```

```
        return logFilter.
    end method.

end class.
```

Register the custom filter builder

Finally, register the custom filter builder using the `LogFilterBuilderRegistry`. Just like when you are registering a custom filter, you should register the custom filter builder early in the application code, such as in a session startup procedure.

```
LogFilterBuilderRegistry:Registry:Put('translation_filter':u,
get-class(Example.Builders.TranslatedMessageFormatBuilder)).
```

Example: Using the custom filter with additional properties

After the custom filter and the custom filter builder have been registered, you can use the filter and specify its properties in the `logging.config` file as shown in this example:

```
{
   "logger": {
      "Company.Order.Sales": {
         {
            "name": "translation_filter",
            "toLang": "fr",
            "serviceURI":
"https://api.cognitive.microsofttranslator.com/translate?api-version=3.0",
            "apiKey": "<APIKey>"
         },
         {
            "name": "NAMED_FILE_WRITER",
            "fileName": "${msg.logger}-${today}.log",
            "appendTo": true
         }
      }
   }
}
```

For a log message produced from a statement like `logger:Info("Customer found.")`, the translated log output would be "`Client retrouvé.`".