Web Services Support in Oracle Enterprise Pack for Eclipse

Learn about the value-add Web Services development support in Oracle Enterprise Pack for Eclipse 11g via a step-by-step tutorial.

By Ronald van Luttikhuizen ACE

Published May 2009

With the emergence of XML and a few arcane ancestors of contemporary WS-* standards in the 1990s, a few frontrunners soon recognized that exchanging data through standard interfaces over the Internet (e.g., XML over HTTP) was the way to go. Initially standards and tooling support was very limited, but later the emergence of industry-wide standards such as SOAP and WSDL gave rise to a “peak of inflated expectations” at which time the use of XML and Web Services became ubiquitous—some would say without good reason.

Eventually, we learned the true value of Web Services: to enable interoperability (where it is needed) and span heterogeneous technology via standards. This nicely fits the idea of service-orientation and reuse where a client can use Web Services to consume a service regardless of the underlying technology and infrastructure.

We now have a wide variety of tools and frameworks to support Web Service development. In the Java world we have seen a fair share of them, such as DOM-based XML parsers and some crude Web Service frameworks that require lots of manual coding yet do not always result in WS-I compliant Web Services. (Several vendors include WS-I support in their tooling, and the nonprofit WS-I offers tools that will verify Web Services compliance. See the Eclipse Wiki for a how-to on testing Web Service artifacts on WS-I compliance.)

The Java Community Process formalized the JAX-RPC (Java API for XML-based RPC; JSR-101) specification in the early 2000s to support XML-based RPC standards in Java. This led to numerous tools and frameworks that were able to generate JAX-RPC Web Services and Web Service clients instead of creating proprietary implementations.

JAX-RPC later evolved into JAX-WS (Java API for XML Web Services; JSR-224)—mainly to reflect that Web Services are more than just RPC and are about communicating XML documents. Some benefits of JAX-WS over JAX-RPC include support for newer standards, improved data binding, better support for attachments using MTOM, support for EJB 3, and an improved and easier programming model using annotations.

JAX-WS also separates the Web Services runtime from data binding functionality, which were intermingled in JAX-RPC, by delegating data binding to JAXB (Java Architecture for XML Binding; JSR-222). JAXB is a framework used to transform XML documents into Java objects—known as marshalling (and vice versa, unmarshalling). This process is referred to as Object-XML Mapping (OXM) and is comparable to Object-Relational Mapping (ORM).

EclipseLink, for example, supports both ORM and OXM through standards such as JPA and JAXB. Note that JAXB can just as well be used on a “standalone basis” to perform data binding without Web Services being involved—for example, when importing XML files and persisting this data.

![Diagram of JAXB process](image)

Figure 1: JAXB’s XJC compiler creates Java classes based on an XML schema. Those classes and the JAXB runtime enable transformation of XML data to Java objects and vice versa at run time. Once these types are generated they can be used as long as the schema does not change. A change in schema requires regeneration of the Java classes.

JAX-WS 2 and JAXB 2 are part of the JEE 5 specification and just like other Java/JEE 5 technologies use annotations and “configuration-by-exception.” That means that you can easily expose a Java class as a Web Service by adding a single @WebService annotation. However, if you want to change the default behavior, you’ll need to use more annotations and/or the JAX-WS API. (The WebLogic Server 10g Rel 3 implementation of JAX-WS is based on the JAX-WS Reference Implementation [JAX-WS RI] version 2.1.4. JAX-WS RI is an open source project that is part of Glassfish. Besides Web Service basics, this release of WebLogic supports various WS-* standards such as WS-Security, WS-Addressing, WS-Policy, SAML, and so on.)

Oracle Enterprise Pack for Eclipse 11g provides a set of certified standards-based plug-ins for the Eclipse IDE that enhances JEE development, debugging, and deployment for Oracle WebLogic Server. At the time of this writing, these plug-ins are available for Eclipse 3.4 (Ganymede) and can be downloaded as all-in-one installer or separate set of plug-ins that can be added to your existing Eclipse environment. Features include Fastswrap (Java Class Redefinition support for WebLogic Server), virtual EAR technology (improving WebLogic Server deployment performance for large applications), graphical editors for WebLogic Server deployment descriptors, and an Oracle Database Plug-in for the Eclipse Data Tools Platform.

Now let’s take a closer look at how Web Service development is done with Oracle Enterprise Pack for Eclipse. Although Web Service development using JAX-WS and JAXB is also possible in default Eclipse distributions, Oracle Enterprise Pack for Eclipse delivers added value by providing additional wizards, project types, and other support that eases Web Service development (especially when using Oracle WebLogic Server as runtime).

Creating a Web Service: How to Choose the Right Approach?
Before we start developing Web Services it is important to know that there are several approaches to create a Web Service. Oracle Enterprise Pack for Eclipse offers support for each one.

- **Top-down or contract first.** The starting point here is the contract of the Web Service: its WSDL. You either design it or it is provided as a "given fact". From the WSDL you generate the Java implementation. If the contract frequently changes, regeneration of the Java code can cause difficulties since the implementation is overridden. If you use this method, make sure you don't change the generated Java classes.

- **Bottom-up or implementation first.** The starting point is the Java implementation; all Web Service artifacts—such as its WSDL—are generated. This is a fast approach when you want to expose existing components as Web Service. However, you need to be careful because you have limited control over the generated Web Service artifacts and it is therefore easy to break an interface if the Web Service is regenerated.

- **Meet-in-the-middle approach.** Here you define both contract and implementation yourself and later on use JAX-WS and JAXB APIs and code to create the glue between them. This is a very flexible approach: you can change both the WSDL and the implementation. It requires more work in the beginning, but is easier to change later on.

Let's begin with a sample app for illustration purposes.

**Sample App: OTN MovieGadget**

This fictitious app is based on an addition to the Oracle’s public wiki (wiki.oracle.com). To extend its social networking capabilities, several gadgets are added. One of these is the "OTN MovieGadget" that lets people interact, discuss, and connect based on their love for movies. It's nice to be linked to others in the Oracle community, but even better to those that love *The Wrestler* or are a huge fan of Brad Pitt or Kate Winslet. To implement such a gadget a Web Service is needed that provides the required services such as searching for movies based on actor or title.

In this step-by-step tutorial you will build this Web Service from scratch using Oracle Enterprise Pack for Eclipse for development and Oracle WebLogic Server as runtime application server. The Web Service will be exposed using JAX-WS. Next you'll create a client for the Web Service and use JAXB to import an XML file containing new movie data. We will use a bottom-up approach in this article since the source code is already provided and bottom-up generation provides a jumpstart for the other tasks in this tutorial. Remember, there's no overall "best" approach; choose the one that best fits your needs.

**Installation and configuration**

Perform the following steps to set up your environment:

- Download and install Oracle WebLogic Server 10g Rel 3 or later (see Downloads portlet).
- Create a new Oracle WebLogic domain called "otn_examples" to which the projects from this tutorial will be deployed and run.
- Download and install Oracle Enterprise Pack for Eclipse 11g (see Downloads portlet).
- Configure a server connection in Oracle Enterprise Pack for Eclipse to the newly created "otn_examples" domain.

Instructions for these steps can be found at the Oracle Enterprise Pack for Eclipse page on OTN.

**Structure**

The following UML Class Diagram depicts the design of the MovieGadgetService component that will implement the gadget.

---

Figure 2: UML Class Diagram showing the design of the MovieGadgetService component
You can find the MovieGadget project that contains the plain Java code in the accompanying zip file. Import it in Oracle Enterprise Pack for Eclipse and run the MovieServicesTester to see how it works. (Note that movie data is stubbed using the MovieServicesDataPopulator class; in a real life project this data would come from a datastore such as a relational database.)

Creating a Web Service project

We start the tutorial by creating a project that will contain our Web Service.

1. Create a new Web Service project. From the menu select File -> New -> Other... In the dialog select WebLogic Web Services -> Web Service Project. Click Next.

![Figure 3: Creating a new Web Service Project in Oracle Enterprise Pack for Eclipse](image-url)

Note that for convenience we create a WebLogic Web Service Project that contains all the right facets, libraries, and so on from the start. In a bottom-up approach, the project containing Java class(es) you want to expose as Web Service usually already exists. In such a case you add JAX-WS and JAXB annotations, add the required libraries and facets, and, if needed, create an EAR project that packages the Java code after which you deploy it on an application server.

2. Enter "MovieGadgetWebService" as project name. If the target runtime is not yet set to "Oracle WebLogic Server 10gR3" then select it from the dropdown list. Select Add project to an EAR. Keep all other default values and click Next.

![New Web Service Project](image-url)
3. Keep all defaults in the **Web Module** dialog and click **Finish** to create the new Web Service Project.

![Creating a new Web Service Project in Oracle Enterprise Pack for Eclipse](image)

**Figure 4:** Creating a new Web Service Project in Oracle Enterprise Pack for Eclipse

4. The new project is created and all relevant facets, libraries, and required deployment descriptors such as web.xml and weblogic.xml are added to the project. Eclipse asks if you want to change the perspective to Java EE. Select **Yes**. This opens all views that are relevant to Java EE software development.

![Changing the perspective in Oracle Enterprise Pack for Eclipse to Java EE](image)

**Figure 5:** Creating a new Web Service Project in Oracle Enterprise Pack for Eclipse - Web Module

**Figure 6:** Changing the perspective in Oracle Enterprise Pack for Eclipse to Java EE

Note that you can change the perspective using the top-right hand side tabs and the **Open Perspective** icon.

**Creating the “MovieGadget” Web Service**

Now let's create the actual Web Service.

1. Add a new JAX-WS Web Service by right-clicking the “MovieGadgetWebService” project and selecting **New -> WebLogic Web Service**.
2. In the **Web Service** dialog enter “MovieGadgetWebService/src/com/oracle/otn/movie/service” as parent folder and “MovieServices” as file name. Click **Finish** to create the Web Service.
Oracle Enterprise Pack for Eclipse generates the “MovieServices.java” class, which is opened in the editor. The class contains the dummy method “hello()”. Note two of the basic JAX-WS annotations: @WebService and @WebMethod. Both annotations are from the javax.jws package. The first annotation indicates that this class should be exposed as Web Service using JAX-WS. The @WebMethod annotation is a method-scope annotation that marks a method to be exposed as Web Service operation. Attributes of these annotations—alongside other generic JAX-WS and WebLogic-specific annotations—can be used to further customize a Web Service. (The use of the @WebService annotation is required as per the JAX-WS specification. If no @WebMethod annotations are used, by default all public methods will be exposed as Web Service operations. When one or more methods are annotated with the @WebMethod annotation, only the annotated methods will be exposed.)

3. Implementation of the service is out of scope, so copy the contents of the “MovieGadgetService/src” directory from the zip file to the “MovieGadgetWebService/src” directory.
4. Refresh the “MovieGadgetWebService” project in Eclipse and open MovieServices.java.
5. Add the @WebService annotation to the class and the @WebMethod annotation to all three methods. Save your modifications.

Adding some annotations is all that is needed to expose a Java class as basic Web Service. Let’s see if it works. (Note that you can use the same mechanism to publish EJB 3 Session Beans as Web Services using JAX-WS and JAXB annotations. This can be convenient when you need to expose existing business logic that is implemented as EJB component as Web Service; for example to integrate it in a SOA-environment. It’s like adding a new interface on the Session Bean besides the local and/or remote interface.)

**Publishing the Web Service to WebLogic**

The next step is to deploy the Web Service on WebLogic and discover what the generated Web Service artifacts such as the WSDL and XSD look like.

Before we start, make sure that Oracle WebLogic Server is started.
1. Add the Web Service Project to WebLogic Server. Right-click on the server in the **Server** view and select **Add and Remove projects...**. In the new dialog, shuffle the **MovieGadgetWebServiceEAR** project to the right and click **Finish**.

![Image of Add and Remove Projects dialog]

**Figure 9:** Adding the Web Service Project to WebLogic Server

Note that publishing the project to the server fails if the “Automatically Acquire Lock and Activate Changes” option is enabled. If so, disable this option using the Oracle WebLogic Server Administration Console. Go to **Preferences**, deselect **Automatically Acquire Lock and Activate Changes**, and click **Save**.

The project is published to the server. If you expand it in the “Server” view you’ll see the project.

![Image of Web Service Project deployed to WebLogic Server]

**Figure 10:** The Web Service Project is deployed to Oracle WebLogic Server

**Testing the Web Service and inspecting the generated Web Service artifacts**

Oracle Enterprise Pack for Eclipse is bundled with an easy-to-use Web Service tester called WebLogic Test Client. The test client is a Web application that ships with WebLogic Server and is integrated in Oracle Enterprise Pack for Eclipse. You might have already noticed it during deployment since the WebLogic Test Client window opened. Let’s see if our Web Service works correctly.

1. If not already opened, start the WebLogic Test Client by clicking the **Launch WebLogic Test Client** button. It should open a test page for the MovieGadget Web Service. All available operations are shown.
Notice that the input arguments are formatted like arg0, arg1, etc. We will change this default behavior in the next section using JAX-WS annotations.

2. Test the getMovieById operation. Enter "1" as value for argument "arg0" and click **getMovieById**. This should return an XML representation of the movie *The Wrestler*.

3. Let's have a look at the generated WSDL and XSDs. Click on the WSDL URL that is displayed in the top of the WebLogic Test Client console. The URL should be something like http://localhost:7001/MovieGadgetWebService/MovieServicesService?WSDL.

Note that a schema definition is also generated and imported by the WSDL. You can view it by opening it in new browser window; the default location is http://host:7001/MovieGadgetWebService/MovieServicesService?xsd=1.
Using annotations to modify default Web Service behavior

If you inspect the WSDL and XSD a little bit closer you may notice some of the generated defaults. This is the same “configuration-by-exception” behavior that is commonly used by more Java/JEE-frameworks such as EJB 3. Let’s use annotations to change parts of this default behavior.

1. The Web Service name is the same as the Java class postfixed with “Service”. The service, port, and operations of the Web Service are in the http://services.movie.otn.oracle.com/ namespace.

   Change the Web Service name to MovieWebService and the target namespace to http://www.oracle.com/otn/movie/.

   In Eclipse replace the @WebService annotation of the MovieServices class with
   @WebService(serviceName="MovieWebService", targetNamespace="http://www.oracle.com/otn/movie/").

2. All input parameters of the Web Service operations are named arg0, arg1, and so on. Also note that these input parameters are located in the “default” or “null” namespace.

   Provide meaningful names to the input parameters and place them in the same namespace as the Web Service and its operations.

   In Eclipse add:
   @WebParam(name="attributeName", targetNamespace="http://www.oracle.com/otn/movie/")

   to every method argument. Replace “attributeName” with the actual name.

3. Republish the project to WebLogic Server and inspect the WSDL using the Test Client. Notice that the Web Service name, target namespaces, and input parameter names have changed.

4. Test all operations using the following arguments:
   a. Operation getMovieById with id “1”. This should return “The Wrestler”.
   b. Operation getMovieByName with name “the”. This should return “The Wrestler”, “The Curious Case of Benjamin Button”, and “Eternal Sunshine of the Spotless Mind”.
   c. Operation searchMovies with searchKey “director” and searchValue “Danny Boyle”. This should return “Slumdog Millionaire” and “28 Days Later...”.

   ![Figure 14: Adding and modifying JAX-WS annotations to change Web Service behavior](image)
A nice thing about the WebLogic Administration Console is that it provides some basic monitoring information on running components. Let’s see how many times our Web Service has been invoked already.

5. Log on to the Oracle WebLogic Server Administration Console and navigate to the MovieGadget Web Service. Select the Monitoring tab and inspect the monitoring information.

An overview of all Web Services Metadata annotations (JSR-181), JAX-WS annotations (JSR-224), JAXB annotations (JSR-222), Common annotations (JSR-250), and WebLogic-specific Web Service annotations can be found here [http://e-docs.bea.com/wls/docs103/webserv_ref/annotations.html]. See the
Generating a Web Service client

The ability to create Web Services is just as important as the ability to create Web Service clients–maybe even more so since there are usually more service consumers than actual services. In this section we will examine Oracle Enterprise Pack for Eclipse’s support for developing Web Service clients by creating a client for the MovieGadget Web Service based on its WSDL.

1. Create a new WebLogic Web Service Project. Enter “MovieGadgetClient” as name and add the project to a new EAR project called “MovieGadgetClientEAR”.

![Image of creating a new WebLogic Web Service Project]

**Figure 18:** Creating a new WebLogic Web Service Project that acts as client to the MovieGadget Web Service

2. Create a Web Service client by right-clicking the “MovieGadgetClient” project and selecting **New -> Other...**. In the dialog select **WebLogic Web Services -> Web Service Client** and click **Next**.

![Image of creating a new Web Service client]

**Figure 19:** Creating a new Web Service client

3. The client code is generated based on the WSDL of the deployed Web Service. Since the Web Service in this tutorial uses a bottom-up approach we don’t have the WSDL locally.

   Select **Remote** and enter the WSDL location that you have obtained in the previous section. This should be **http://host:7001/MovieGadgetWebService**.
Make sure that MovieGadgetClient is selected as client project. Click the Validate WSDL button. The Next and Finish buttons are enabled after successful validation. Click Next.

4. Select the Keep generated Ant script so you can regenerate the client later on. Keep all other defaults and click Next.

5. In the Customization Options dialog select the Copy WSDL into client JAR and click Finish.
The WebLogic clientgen Ant task generates proxy code to invoke the MovieGadget Web Service. This code is packaged in a JAR file and placed in the WebContent/WEB-INF/lib directory of the MovieGadgetClient project. The archive contains both source and compiled classes. This JAR can be used by other applications that also need to invoke the MovieGadget Web Service through packaging or by creating a shared library on the server and importing it.

The clientgen_build.xml Ant file is placed in the project's root directory. The Ant task uses JAX-WS – see for example the @WebServiceClient annotation in MovieWebService.java – to invoke the Web Service. JAXB is used to map the XML datatypes to generated Java objects. See, for example, the JAXB object factory in the ObjectFactory.java.

To demonstrate this we can use the proxy code to invoke the Web Service from a webpage.

6. Create a new Servlet by right-clicking the MovieGadgetClient project and selecting New -> Other... In the dialog select Web -> Servlet and click Next.

7. In the Create Servlet dialog enter “com.oracle.otn.movie.servlets” as package and “MovieGadgetClientServlet” as Class name. Click Finish.

8. Copy the code from the MovieGadgetClientServlet located in the MovieGadgetClient project of the zip file to your newly created Servlet. Notice that the Servlet invokes the Web Service using the classes from the Web Service client JAR.

9. Publish the MovieGadgetClientEAR project to WebLogic Server.

10. Invoke the Servlet by opening a browser window and entering its URL, which should be http://host:7001/MovieGadgetClient/MovieGadgetClientServlet. The following Web page – showing the results of the Web Service invocations – is displayed.
The final part of the tutorial shows Oracle Enterprise Pack for Eclipse's support for data-binding using JAXB.

Object-XML Mapping using JAXB

JAXB is a standard for data binding that can be used on a standalone basis as well as be used from JAX-WS. In a real-life project this capability can for example be used to import XML-data and persist it into a relational database.

In this section we will create a simple application that reads an XML-file containing movie data, translates it to Java objects, and displays the data on a simple Web page.

1. Create a new WebLogic Web Service Project. Enter "MovieGadgetDataImport" as name and add the project to a new EAR project called "MovieGadgetDataImportEAR".
2. Copy the movie.xsd and two example xml files located in the MovieGadgetDataImport project of the zip file to the src directory of the newly created project. Refresh the project in Eclipse.
3. Generate Java classes based on the movie.xsd schema. Right-click the MovieGadgetDataImport project and select New -> Other... In the dialog select WebLogic Web Services -> JAXB Types and click Next.
4. In the New JAXB Types dialog select the MovieGadgetDataImport project and add movies.xsd to the list of schema files. Click Next.
5. In the Generated Artifacts dialog select the Keep generated Ant script option so we can execute the Ant task again if needed. Keep the other defaults.
6. Enter “com.oracle.otn.movies.jaxb.entities” as the Java package in which the JAXB types will be generated. Also select the Include source code in JAR option so you can have a look at the generated code later on. Click Finish.

The XJC binding compiler generates the JAXB types based on the schema and packages these Java classes in the movie.xsd.jar archive together with the ObjectFactory class that allows you to programmatically construct new instances of the Java types, and the package-info class that specifies the top-level package to which the generated types belong.

The movie.xsd.jar archive is located in the WebContent/WEB-INF/lib directory of the MovieGadgetDataImport project.

To demonstrate its functionality we will create a Servlet that transforms an XML file containing movie data to the Java objects that we’ve just created. This is called unmarshalling. The process of generating XML based on JAXB object types (marshalling) is pretty straightforward and is left as an exercise to the reader.

7. Create a new Servlet by right-clicking the MovieGadgetDataImport project and selecting New -> Other... . In the dialog select Web -> Servlet and click Next.
     8. In the Create Servlet dialog enter “com.oracle.otn.movie.servlets” as package and “DataImportServlet” as Class name. Click Finish.

9. Copy the code from the DataImportServlet located in the MovieGadgetDataImport project of the zip file to your newly created Servlet. Notice the JAXB-related statements.

10. Publish the MovieGadgetDataImportEAR project to WebLogic Server.

11. Invoke the Servlet by opening a browser window and entering its URL, which should be http://localhost:7001/MovieGadgetClient /MovieGadgetClientServlet. The following Web page, showing the results of the Web Service invocations, is displayed.
Note that XML that is unmarshalled must adhere to the XML Schema that was used to generate the JAXB types. The advantages of this approach are compile-time type checking (as opposed to for example DOM) and faster XML-processing. If the XML is not valid according to the schema, JAXB will generate a runtime error. If a schema changes frequently or there is no schema at all, you might want to consider an XML framework that does not require the generation of mappings.

Conclusion

We’ll wrap up this tutorial with some best practices for Web Service development:

- Be careful when using vendor-specific annotations (as opposed to those defined in the JAX-WS and JAXB standards). Although vendor-specific annotations can be very powerful they break portability of Web Services and tie them to a specific runtime.
- Don’t expose all methods as Web Service operations by default when using a bottom-up or meet-in-the-middle approach. Only expose those methods that are actually needed by service consumers. This promotes encapsulation and prevents access to “internal” methods.
- Nowadays most products and technologies support Web Services. Keep their pros and cons in mind when deciding what technology to use. Java for example provides better support and a better runtime for Web Service development and XML-processing than relational databases.
- Avoid creating Web Services that receive, process, and/or send very large XML documents. XML processing is resource-intensive and relatively slow and therefore not well equipped for handling bulk data. Use other technologies such as database or ETL tools for that purpose.
- It’s easy to develop basic Web Services—but it’s hard to make them robust, secure, and scalable (enough). Address these non-functional issues in the beginning of the project instead of discovering that requirements are not met at the end of your project.
- Decide upfront, based on the requirements and constraints, what approach for Web Service development best suits your situation: top-down or contract first, bottom-up, or meet-in-the-middle.

Ronald van Luttikhuizen (ronald.vanluttikhuizen@approach-alliance.nl) is an Oracle ACE and architect at Approach Alliance (www.approach-alliance.com), a Netherlands-based information and communications technology consultancy focusing on SOA and BI.