BEA WebLogic Workshop™ Documentation

Introducing WebLogic Workshop

WebLogic Workshop is the industry's leading visual development environment for building enterprise−class web services. With WebLogic Workshop, you can focus on building business logic into your web service rather than on complex implementation details. Whether you are an application developer with a problem to solve and no J2EE experience or a J2EE code jockey, WebLogic Workshop makes it easy to design, test, and deploy web services.

WebLogic Workshop's intuitive user interface lets you design your web service visually. Controls make it simple to connect to enterprise resources, like databases and Enterprise JavaBeans, without writing a lot of code. Conversations handle the job of keeping track of application state. And WebLogic Workshop's support for asynchronous processes makes it easy to build highly reliable web services for the enterprise.

The following topics are good places to start if you are new to web services and WebLogic Workshop.

What Are Web Services?

What are web services, and what technologies do they rely on?

Tutorial: Your First Web Service

Build your first web service quickly with our hands−on tutorial. Then explore WebLogic Workshop in−depth to learn how to build powerful services.

Building Web Services with WebLogic Workshop

Gives an overview of how you build services with WebLogic Workshop.

How Do I? Topics

Provides links to information about common tasks in building web services.

Introduction to Java

Covers the basics of Java programming for the new Java developer.

Introduction to XML

Introduces Extensible Markup Language (XML), the language with which web services communicate.

Sample Web Services

Lists and describes samples included with WebLogic Workshop.

An Overview of Web Services

A web service is a set of functions packaged into a single entity that is available to other systems on a network. The network can be a corporate intranet or the Internet. Other systems can call these functions to request data or perform an operation. Because they rely on basic, standard technologies which most systems, they are an excellent means for connecting distributed systems together.

Web services are a useful way to provide data to an array of consumers over the Internet, like stock quotes and weather reports. But they take on a new power in the enterprise, where they offer a flexible solution for integrating distributed systems, whether legacy systems or new technology.

The following topics offer an overview of web services and the standard technologies they rely on.

What Are Web Services?

Explores how web services can offer a new approach for existing challenges

Inside Web Services

Discusses the standard technologies underlying web services.

What Are Web Services?

The following topic gives a non−technical introduction to the basic concepts behind web services: what web services are, how they work and what problems they are intended to solve.

The Problem with Existing Applications

Today, companies rely on thousands of different software applications each with their own role to play in running a business. To name just a few, database applications store information about customers and inventories, web applications allow customers to browse and purchase products online, and sales tracking applications help business identify trends and make decisions for the future.

These different software applications run on a wide range of different platforms and operating systems, and they are implemented in different programming languages. As a result, it is very difficult for different applications to communicate with one another and share their resources in a coordinated way.

Take, for example, a company that has its customer data stored in one application, its inventory data stored in another application, and its purchasing orders from customers in a third. Until now, if this company wanted to integrate these different systems, it had to employ developers to create custom bridging software to allow the different applications to communicate with one another. However, these sorts of solutions are often piecemeal and time consuming. As soon as a change is made to one application, corresponding changes have to made to the other applications linked to it and to the bridges that link the applications together.

The Web Services Solution

To solve the problem of application−to−application communication, businesses need a standardized way for applications to communicate with one another over networks, no matter how those applications were originally implemented. Web services provide exactly this solution by providing a standardized method of communication between software applications. With a standardized method of communication in place, different applications can be integrated together in ways not possible before. Different applications can be made to call on each other's resources easily and reliably, and the different resources that applications already provide can be linked together to provide new sorts of resources and functionality.

Moreover, application integration becomes much more flexible because web services provide a form of communication that is not tied to any particular platform or programming language. The interior implementation of one application can change without changing the communication channels between it and the other applications with which it is coordinated. In short, web services provide a standard way to expose an application's resources to the outside world so that any user can draw on the resources of the application.

Related Topics

What Are Web Services?

The following topic describes what a web service is, the basic technologies that underlie a web service, and how a web service build with WebLogic Workshop is really an enterprise level web application.

Web Services and Networks

Essentially, a web service makes software application resources available over networks in a standardized fashion. Other technologies have done the same thing, such as Internet browsers, which make web pages available using standard Internet technologies such as HTTP and HTML. However, these technologies are generally used as a way for human users to view data on a web server and, on their own, are not well suited to enabling application to application communication and integration. What is new and exciting about web service technology is its ability to allow software applications to talk to one another and utilize each other's resource. Using web service technology, one application can call on another to perform simple or complex tasks, even if the two applications are running on different operating systems and are written in different languages. In other words, a web service makes its resources available in such a way that any client application, regardless of its internal implementation, can operate and draw on the resources provided by the web service.

Basic Technologies

Web services are able to expose their resources in this generally accessible way because they adhere to the following communication standards:

1. A web service publicly describes its own functionality through a WSDL file.
2. A web service communicates with other applications via XML messages.
3. A web service uses a standard network protocol such as HTTP.

WSDL

A Web Service Description Language (WSDL) file describes how the web service is operated and how other software applications can interface with the web service. Think of a WSDL file as the instruction manual for a web service explaining how a user can draw on the resources provided by the web service. WSDLs are generally publicly accessible and provide enough detail so that potential clients can figure out how to operate the service solely from reading the WSDL file. If a web service translates English sentences into French, the WSDL file will explain how the English sentences should be sent to the web service, and how the French translation will be returned to the requesting client. For more information on WSDL files see WSDL Files: Web Service Descriptions.

XML and SOAP

Extensible Markup Language (XML) messages provide the common language by which different applications can talk to one another over a network. To operate a web service a user sends an XML message containing a request for the web service to perform some operation; in response the web service sends back another XML message containing the results of the operation. Typically these XML messages are formatted according to SOAP syntax.

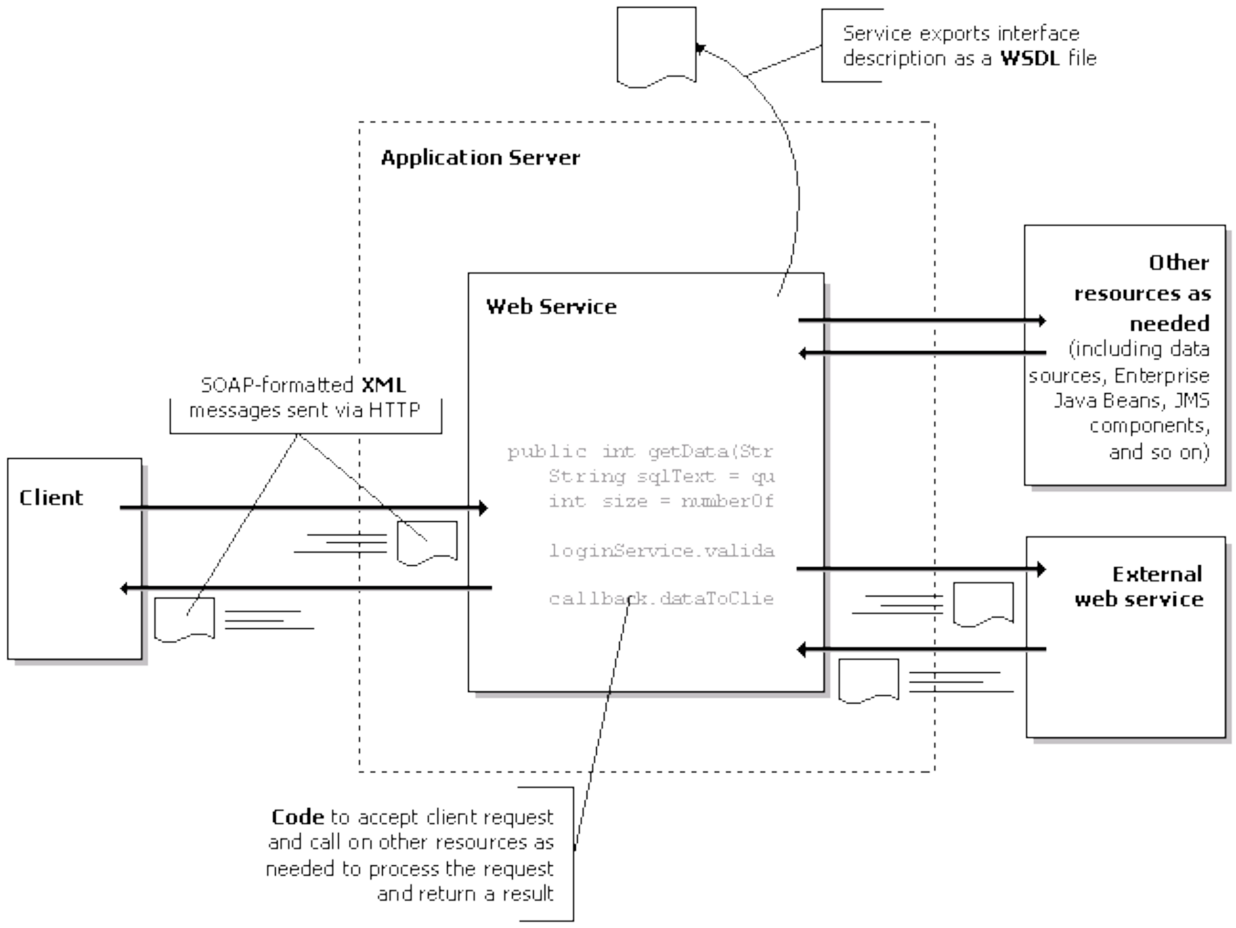
Simple Object Access Protocol (SOAP) specifies a standard format for applications to call each other's methods and pass data to one another. Note that other non−SOAP forms of XML messages are possible, depending on the specific requirements of the web service. But, in any case, the sort of XML message and the specific syntax required can be found in the WSDL file, making the web service generally available to any client application capable of sending and receiving the appropriate XML messages. For more information about XML see Introduction to XML.

HTTP

To make it accessible to other applications across networks, such as the Internet and in−house intranets, web services receive requests and send responses using widely used protocols such as HyperText Transfer Protocol (HTTP) and Java Message Service (JMS).

Web Service Architecture

The following illustration shows the relationship between a web service (in the center), its client software applications (on the left), and the resources it uses, including databases, other web services, and so on (on the right). A web service communicates with clients and resources over standard protocols such as HTTP by exchanging XML messages. The WebLogic Server on which the web service is deployed is responsible for routing incoming XML messages to the web service code that you write. The web service exports a WSDL file to describe its interface, which other developers may use to write components to access the service.


Service exports interface description as a **WSDL** file
Application Server
Web Service
Other resources as needed (including data sources, Enterprise Java Beans, JMS components, and so on)
SOAP-formatted **XML** messages sent via HTTP
Client
public int getData(Str
String sqlText = qu
int size = numberOf
loginService.valida
callback.dataToClie
External web service
Code to accept client request and call on other resources as needed to process the request and return a result

Other Characteristics of Web Services

The following characteristics of web services make them well−suited for creating integrated systems on the Internet or intranets.

Reusable

Just like other component technologies, once you have written code and exposed it as a web service, you or others may use that code again and again from various applications. Once you've written and tested your code, it’s easy to make use of it in new applications. Because web services use standard XML protocols, these components can be consumed by a much wider variety of users than with previous component based technologies.

Flexible

Because web services communicate using extensible XML documents (instead of small pieces of data), they can more easily accommodate changing data and content over time. This also encourages packaging many activities into a single method call over the network. This greatly improves the efficiency of your applications, especially over high−latency networks such as the Internet or corporate Wide Area Networks (WANs).

Standards−Based

Web services are built on standard technologies such as HTTP and XML. All web service messages are exchanged using a standard XML−messaging protocol known as SOAP, and web service interfaces are described using documents in the WSDL standard. These standards are all completely agnostic of the platform on which the web services were built.

WebLogic Workshop and Enterprise Web Services

WebLogic Workshop provides a complete framework for easily building web services applications that automatically leverage the power, reliability and scalability of WebLogic Server. Moreover, Workshop isolates developers from the low−level details of communicating via XML messages, creating web services descriptions (WSDL files), and accessing enterprise level back end resources.

Workshop also provides additional web service functionality in order to meet the requirements of using web services in the enterprise. In particular, WebLogic Workshop helps you build web services which are:

Asynchronous

Many business processes take more than a few moments to complete; traditional architectures make it hard to handle long−running tasks efficiently. WebLogic Workshop helps you architect asynchronous web services easily by introducing the notion of conversations and callbacks. Conversations help manage the typical problems in asynchronous messaging, namely correlating messages and managing some information or state between message exchanges. Workshop also allows a web service to notify a client when the results of the operation are ready using a callback. A callback is a web service message sent from your web service to the client. These two features help you implement long−running business processes efficiently and easily.

In addition, Workshop supports the use of JMS queues as message buffers to ensure that web service messages are not lost regardless of server load. JMS can also be used by Workshop web services to communicate with back end resources.

Loosely−Coupled

Web services by their very nature are somewhat loosely coupled — information is exchanged using XML which isolates clients of a web service from the interior implementation of the web service. This lets a web service describe an interface or "public contract" which determines how users can interact with the web service. As long as the messages and interfaces described by this contract are maintained, the internal implementation of the web service (or the client) is free to change at will. Maintaining this contract requires a simple technology for changing how Java is mapped into XML (and vice versa) so that the public contract of your web service can be maintained with minimal work when the internal implementation changes.

Reliable, Available, and Scalable

While the web services you create in WebLogic Workshop are simple to create, ultimately they are compiled to standard J2EE applications. This means your web services will have all the reliability, scalability, and availability you've come to expect from J2EE applications running on WebLogic—absolute requirements for web services deployed in the enterprise.

Multiple Protocols

WebLogic Workshop helps you create web services that run not just over HTTP but also over JMS queues. This allows you to use web services to communicate easily with internal systems and intranet resources.

Related Topics

Building Web Services with WebLogic Workshop

Building Web Services with WebLogic Workshop

Although XML, SOAP and WSDL form the core technologies behind a web service, a developer working with the WebLogic Workshop is for the most part freed from the cumbersome details of these technologies. WebLogic Workshop simplifies web service development by allowing the developer to focus on the application logic of the web service, rather than focusing on the complex syntax of XML, SOAP and WSDL files. With WebLogic Workshop you can take a functional approach to development tasks: you tell WebLogic Workshop how you want your web service to function, and WebLogic Workshop implements the core web service technologies according to your instructions. WebLogic Workshop's visual development environment provides a graphical representation of the web service under development, giving the developer a complete picture of the web service's different parts and how the web service connects to other software applications and data resources. The topic below explains the basic concepts required to begin building web services with WebLogic Workshop.

JWS Files

The Java Web Service (JWS) file is the core of your web service. It contains the Java code that determines how your web service behaves. In fact, a JWS file is nothing other than an ordinary Java class file. In this respect, you can think of a web service built on WebLogic Workshop as a Java class which communicates with the outside world through XML messages. If you are unfamiliar with programming in Java, see Introduction to Java.

The WebLogic Workshop user interface offers two ways of viewing JWS files: Source View and Design View. Source View offers a direct view of the Java code within a JWS file; Design View offers a graphical representation of the code within a JWS file.

Just as you can view a web service in either Design View or Source View, you can also edit a web service in either Design View or Source View. As you make changes to a web service in Design View, WebLogic Workshop will make the corresponding changes to the underlying code; similarly, if you make changes to the code directly in Source View, those changes will automatically be reflected in Design View.

For more information on JWS files see JWS Files.

Working with Design View

When you view a JWS file in Design View, the web service itself is depicted in the middle of the screen, the client application is depicted on the left side, and the web service's data resources are depicted on the right side. The client, the application which calls your web service, might be a Java application, a VisualBasic application, or another web serviceadov−−>any application designed to send and receive XML messages. Similarly, the data resources for your web service can be any sort of application, such as a database or another web service, provided it can send and receives XML messages. As a general rule, when you are building a new web service, you should begin in Design View. In the initial Design View stage you can define what data resources your web service will draw from, and how your web service interacts with its client application and its data resources. Once you have defined the basic shape of your web service in Design View, then begin working in Source View, where you can define the details of the interior implementation of your web service.

For more information about Design View, see User Interface Reference.

Client Interface

If you examine any web service (that is, any JWS file) in Design View you will see arrows bridging the gap between the client application and the web service. These arrows represent the interface, or "public contract", between the client and the web service. Arrows pointing from the client toward the web service represent the methods through which the client application invokes the functionality of the web service. (Note that methods which are exposed to the client are sometimes referred to as operations to distinguish them from other methods in your web service.) Arrow pointing from the web service toward the client represent callbacks. Callbacks are one of the ways that your web service can send information back to the client application. Callbacks are especially useful for communication in network contexts, because callbacks make it possible for a client application to continue with its own processes instead of halting and waiting for a response from a slow or otherwise delayed web service. For example, if the web service is called upon to perform a time consuming operation, it can free up the client's own processes by immediately sending an acknowledgement of the client's request instead of the full result of the operation. Once the operation is complete the web service can send the full result via a callback to the client. This style of inter−application communication is called asynchronous communication, as opposed to synchronous communication, which requires that a client synchronize its own processes with the processes it invokes on another machine. Hence in synchronous communication, the client application may have to halt and wait for the full results to be returned whenever an operation it invokes an operation on an external machine.

For a more detailed treatment of asynchronous communications in web services see Using Asynchrony to Enable Long−Running Operations.

Controls

When you build a web service that must draw data from other resources—such as databases, legacy applications, and other web services—you can incorporate those resources into your design by using controls. Controls manage the communication between a web service and a data resource by means of control methods and callback handlers. Control methods (depicted as arrows pointing from the web service, through the control, to the data resource) allow the web service to invoke the functionality of the data resource; callback handlers (depicted as arrows pointing from the data resource, through the control, to the web service) allow the web service to listen for and receive callbacks from the data resource.

Note that the underlying Java code for a control is not incorporated directly into a web service's JWS file, instead the code for a control appears in a separate CTRL file. This kind of file allows you to reuse the same control in many different web services without repeatedly writing the same Java code.

Controls greatly simplify the complex task of making your web service inter−operate with other applications. Instead of building piecemeal, ad hoc interfaces between your web service and other applications, controls provide a single model for interfacing with a wide variety of different applications. Whether the application is a database or another web service, controls let your web service access the application simply by calling control methods directly from the Java code of your web service. Moreover, WebLogic Workshop provides powerful tools for autogenerating controls. For example, WebLogic Workshop will automatically generate a CTRL file based on a WSDL file, allowing your web service easy interface with any other web service on the internet.

WebLogic Workshop supports the following kinds of controls:

Service control — A Service control provides an interface to another web service, allowing your service to invoke the methods and handle the callbacks of the other service. The other web service can be one developed with WebLogic Workshop or any web service for which a WSDL file is available.

Timer control — A Timer control notifies your web service when a specified period of time has elapsed or when a specified absolute time has been reached.

EJB control — An EJB control provides an interface for access to Enterprise Java Beans (EJBs). EJBs are Java software components of enterprise applications.

Database control — A Database control provides access to a relational database, allowing a web service to query the database by calling Java methods and operating on Java objects . The Database control automatically performs the translation from database queries to Java objects and vice versa.

JMS control — A JMS control enables web services to easily interact with messaging systems that provide a Java Message Service (JMS) implementation.

For more information about controls, see Controls: Using Resources from a Web Service.

Setting Properties

Another way that WebLogic Workshop lets you focus on application logic rather than the complexities of the underlying web service technologies is through setting properties of your web service directly through the graphical user interface. Instead of writing complex Java code to make you web service support enterprise level features, you can simply use WebLogic Workshop to make your web service implement these features. Each item in your service design, from individual methods, up through controls, and including the whole web service itself, exposes its own set of properties which can be modified directly through Design View.

Examples of functionality you can use merely by setting properties include:

- Message buffering, a technology that places incoming and outgoing XML messages into a queue for better handling of high demand.
- Conversations, a technology through which WebLogic Server maintains your service's state and correlates its communications. For more information about conversations see Maintaining State with Conversations.
- Protocols (HTTP, JMS, etc.) that may be used to access your service.

Setting properties in Design View adds Javadoc annotations to your source code. Originally a technology for extracting in−source documentation (hence the name), Javadoc technology is ideal for associating properties with aspects of your web service.

Javadoc tags used in WebLogic Workshop begin with an @jws prefix. The @ symbol is a Javadoc convention that signals to the compiler the presence of a word that should be interpreted as a Javadoc tag. The "jws" prefix stands for "Java Web Service" and differentiates web service tags from other Javadoc tags (for instance, you may use your own Javadoc tags to embed documentation about your web services).

The following example contains three Javadoc tags: @jws:operation specifies that the requestReport method should be exposed to be called by clients; @jws:conversation specifies that the requestReport method starts a conversation; and @jws:message−buffer specifies that the XML messages to and from the requestReport method should be queued.

```
/**
* @jws:operation
* @jws:conversation phase="start"
* @jws:message−buffer enable="true"
*/
public void requestReport(String id)
{...}
```

For reference information on Javadoc tags, see Javadoc Tag Reference.

XML Maps

Because a web service communicates with external applications via XML messages, much of the work of building a web service involves translating back and forth between XML and whatever language implements the interior application logic of your web service. If your web service's interior implementation is written in Java, then outgoing messages need to be translated from Java method calls into the appropriate XML message; likewise incoming XML messages need to be translated into the appropriate Java method calls. In the case of WebLogic Workshop all of this translation work is done for you—you don't need to write any scripts to parse apart incoming XML messages or put together outgoing XML messages. However, if necessary, WebLogic Workshop does allow you to access and customize the process of XML−Java translation via the use of XMLMaps and ECMAScript.

For more information about XML maps see Handling and Shaping XML Messages with XML Maps.

Using Web Services: A Real World Example

The following topic describes a problem faced by a fictional company and provides a sketch of a web service solution.

The Scenario

Imagine that you work for a company that evaluates credit applications for other clients, such as banks and mortgage lenders. These clients come to your company for help in evaluating an applicant's credit history and credit worthiness. Once your company has researched and evaluated an applicant's credit worthiness, you return a credit worthiness score to your client, who then decides whether or not to extend credit to the applicant.

The credit evaluation process involves three steps. First, an applicant is checked against an in−house database of bankruptcies. This database is maintained by your company and is accessible over the company's intranet. Second, a credit history for the applicant is retrieved from an external vendor. The external vendor has set up a web service that provides information about an applicant's credit cards, including how many credit cards she holds and the available credit on each card. Finally, the information retrieved in the first two steps is passed to a credit evaluation formula that returns a credit score indicating whether the applicant is a good or poor credit risk. For this example, the formula is an operation already implemented as an EJB (Enterprise Java Bean) embedded in another software application.

The Problem

The problem scenario described above contains the following three resources:

1. An in−house database that supplies bankruptcy data
2. Another web service that supplies credit card histories
3. A credit evaluation formula

In order to make the evaluation process consistent and automated, your company now wants to build a credit evaluation process that ties together all of the relevant resources available both externally and internally. WebLogic Workshop is an ideal tool to apply to this problem because WebLogic Workshop is good at two things: (1) Workshop makes it easy to access and coordinate a wide variety of external resources into a single web service, and (2) Workshop makes it easy to expose this web service over networks to other users and software applications. When approaching the particular problem described above, a web service developer can use WebLogic Workshop to create the following solution.

The Solution

The web service developer begins by deciding what sort of operations she wants her web service to expose to the outside word. In this particular case, the goal is to collect and evaluate information on a particular credit applicant, so the developer decides to expose a single operation of her web service and creates a single method that takes the applicant's Social Security number as a parameter and returns the applicant's credit worthiness score. She then establishes connections between the web service itself and its external resources.

Controls

To establish these connections between resources, the developer uses tools in WebLogic Workshop, called controls, which manage the communications between a web service and its external resources. The developer would build three separate controls: one to retrieve data from the in−house database, another to retrieve data from the web service providing credit card histories, and a finally one to access the EJB which calculates a credit worthiness score. Once these controls are in place, the developer can write Java code to coordinate all three controls and the pass the final credit score back to the clients of your company.

User Interface

Because a web service typically does not require a very sophisticated user interface, the developer does not need to spend time developing one. Web services themselves typically have thin user interfaces designed for other software applications to access the web service, rather than for direct human access. For the particular web service described above, there is no sophisticated GUI providing an interface between the web service and a human user. Instead the clients to the web service are responsible for building user interfaces that provide access to the web service.

Network Latency

Because web services are involved with transmitting data across networks, a web service developer should address the issue of network latency. Latency refers to the time a client must wait for a request to be processed and returned over the network. A number of factors go into determining network latency. A network may have more traffic at certain times of the day, thereby increasing the network latency; or a web service may not be able to complete a process until a human intervenes to input some data or authorize some step in the process.

The web service developer can cope with network latency by building an asynchronous web service, which does not require that the client application halt and wait for the web service to process and return the requested data. Instead, an asynchronous web service immediately returns a simple acknowledgment to the client that a request has been received, and then, at a later time, the web service sends a callback to the client containing the full response. For more information about asynchronous web services see Using Asynchrony to Enable Long−Running Operations.

Client Applications

The developer should also keep in mind the sort of client applications that will access the web service. The client application that calls the web service might be a JSP page, a Java application, a VisualBasic application, another web service or any software application that can communicate using XML messages. For the web service described above, the developer should consider what sort of client is likely to call the web service and what its capabilities are. If a client of the web service cannot support callbacks, the developer might consider providing a polling interface to allow such clients to access the web service. A polling interface is one where the client periodically calls on the web service until the web service has completed a full response to the client. Once the web service has completed the full response, it is returned to the client as a return value, rather than as a callback. For more information on polling interfaces see Using Polling as an Alternative to Callbacks. For more information about callbacks see Using Callbacks to Notify Clients of Events.

To see step by step instructions for building a web service like the one described above, see Tutorial: Your First Web Service

Web Service Development Cycle

This topic describes the process of developing web services with WebLogic Workshop. The steps in developing a web service are the same as for any software development: design, implement, test, debug and deploy.

Designing a Web Service

When designing a new web service, you specify the names and parameters of all of the service's exposed operations and callbacks. In WebLogic Workshop, you accomplish this task in the Design View. To lean more about WebLogic Workshop's Design View, see Design View.

The specific steps you follow to create and define a new web service are:

- Create a new JWS file in an appropriate folder within your project. To learn about WebLogic Workshop projects, see WebLogic Workshop Projects.
- Add the methods your web service will expose and configure the methods' parameters.
- Add any callbacks your web service will expose and configure the callbacks' parameters. To learn about callbacks, see Using Callbacks to Notify Clients of Events.
- Determine and configure the conversation phase of each method and callback. To learn more about conversations, see Maintaining State with Conversations.

The set of methods and callbacks you define for your web service is called the public interface or public contract. Once you have defined the public interface of your web service, you are ready to implement the web service's internal logic. Implementation is described in the next section.

Implementing Web Service Code

You implement the internal logic of your web service in WebLogic Workshop's Source View. In Source View, you can edit the contents of the JWS file that defines your web service. If you added methods or callbacks in Design View, you will see the definitions of those methods and callbacks in Source View. To implement your web service's logic, add code to the body of the various methods, and invoke callbacks where appropriate.

It may be necessary or desirable for your web service to utilize other resources, such as a database or another web service. In WebLogic Workshop, you accomplish this via controls. Controls provide a simple, common model for interacting with resources from within your web service. To learn more about controls, see Controls: Using Resources from a Web Service.

While WebLogic Workshop web services are written in the Java programming language, WebLogic Workshop strives to make it possible to implement Java web services without being a Java expert. If you are competent in any programming language and are familiar with common programming concepts like variable declarations, method declarations, control structures (if−then−else statements, for loops, etc.) you should be able to implement web services in WebLogic Workshop.

If you are completely new to the Java programming language, see Introduction to Java.

Testing a Web Service

Once you have implemented the logic of a web service, you will want to test it. WebLogic Workshop provides a test environment for the web services you develop. This environment is called Test View.

Test View runs in a browser. Remember that your web service's methods are typically invoked via the HTTP web protocol (thus the name web services). Real clients will typically invoke your web service by sending HTTP requests and receiving HTTP responses (the same thing your browser always does). However, the responses from web service method invocations are not typically HTML pages, they are typically XML messages.

Test View provides a way for you to invoke a web service's methods from a browser and view the XML messages that are exchanged. Test View keeps a log of activity while testing a web service so that you can examine the details of the interaction between the client and web service at any point.

Test View can be reached directly via the Start or Start with Debug actions in WebLogic Workshop's user interface. However, you may also enter Test View directly by entering the URL of your web service in the address bar of a browser (assuming the WebLogic Server hosting your web service is running).

To learn more about testing web services in Test View, see Test View.

Debugging a Web Service

Since web services are application components containing code, and code is seldom correct the first time it is written, you need a way to debug your web service's code. WebLogic Workshop provides full debugging capabilities for all code that implements a web service. This includes the Java code in your web service's methods as well as any ECMAScript code you may be using to map XML to Java types and vice versa.

WebLogic Workshop provides an efficient edit−compile−debug cycle so that you can arrive at correct web service code quickly and easily. To debug, begin by setting breakpoints at desired locations in your web service's source code. Next execute your service. When execution reaches one of your break points, execution is suspended and you can examine the state of your service's variables and environment. When you are ready, you can continue execution, perhaps stopping at another breakpoint. When you locate a programming error, you may correct the error in the program source and re−execute the web service to test the new code.

To learn more about how to debug WebLogic Workshop web services, see Debugging Web Services.

Deploying a Web Service

One you have designed, implemented and debugged a web service, you are ready to make that web service available to potential clients. Clients may be customers, business partners or other software components within your organization. The process of making your web service available on a production server and publishing its location is known as deployment.

At a basic level, deployment of WebLogic Workshop services is very simple. You merely package the web service as an EAR file, and copy the EAR file to a production server on which the WebLogic Workshop runtime environment is installed. Your web service is immediately available.

In enterprise environments, the deployment story is typically more complicated, involving various levels of staging and testing to ensure the code being deployed won't adversely affect other applications on the eventual production server.

To learn more about deployment of WebLogic Workshop web services, see Deploying Web Services.

Related Topics

What Are Web Services?

Building Web Services with WebLogic Workshop

Structure of a JWS File

How Do I: Start WebLogic Workshop?

WebLogic Workshop is a visual development environment for building enterprise−class web services. Once you have installed the WebLogic platform, you can start WebLogic Server using the following steps.

To Start WebLogic Workshop on Microsoft Windows

- Click Start−−>Programs−−>BEA WebLogic Platform 7.0−−>WebLogic Workshop−−>WebLogic Workshop.

To Start WebLogic Workshop on Linux

1. Using a file system browser or shell, locate the Workshop.sh file at:

$HOME/bea/weblogic700/workshop/Workshop.sh

2. Run Workshop.sh as follows:

- From the command line, use the following command:

```
sh Workshop.sh
```

- From a file system browser, double−click Workshop.sh.

Related Topics

Tutorial: Your First Web Service

Java and XML Basics

What if you don't know Java and XML? Fortunately, you don't need to be an expert in either to build web services in WebLogic Workshop. If you have some prior programming experience in another programming language, you should be able to learn the basics of Java quickly. And XML, although it may look complex, is not even a programming language—it's just a structured way to describe data.

This section provides a brief overview of Java and XML to get you started. If you need more, there are a number of excellent books available on both Java and XML.

Introduction to Java

Learn the basics of programming in Java.

Introduction to XML

Explore the parts of an XML document.

Introduction to Java

This section outlines the basic rules of Java syntax and gives a brief introduction to Java concepts useful for the WebLogic Workshop developer. If you are familiar with another high−level programming language, such as C, C++, Visual Basic, or even JavaScript, you should have no trouble learning Java quickly. If you want a more detailed and thorough introduction to Java, there are a number of excellent books and courses available to you.

Basic Java Syntax

The following sections cover some some of the basics of Java syntax.

Primitive Types and Wrapper Classes

In Java, you must declare a variable before you use it, providing both its name and type. While this may seem like more work up front than simply using the variable when you need it, the extra bit of effort pays off, because the compiler can check and enforce type restrictions. This can save you debugging time later on.

For example, you can declare a new variable to store an integer as follows:

```
int x;
```

The following declares a new variable to hold a text string:

```
String strName;
```

After you’ve declared the variable, you can assign a value to it:

```
x = 5;
```

or

```
strName = "Carl";
```

The prefix str in the variable above makes it easy for the developer to remember that it is a variable of type String.

Java provides several primitive types that handle most kinds of basic data. The primitive types are:

- boolean
- byte
- char
- double
- float
- int
- long
- short

In Java, you use primitive types to handle basic values and Java objects to store more complex data. Every Java object is derived from a basic Object class. A class is a template for creating an object; you can think of a class as a cookie cutter, and the objects created from it as the cookies.

It is important to keep in mind the difference between a primitive type and an object. An Integer object is different from an int, even though both store an integer value. A method that accepts one as an argument will not accept the other unless it has been specifically defined to do so. Also, Java objects always have methods which you use to work with them; a primitive type cannot have a method.

Note that in Java, the memory size of the primitive types is well−defined. An int, for example, is always a 32−bit value.

Java Tip: Many useful classes in Java that store and manage collections of objects (lists, trees, arrays, etc.) cannot operate on primitive types, because the primitive types are not derived from the basic Java Object class. However, Java includes wrapper classes for the primitive types, which you can use to temporarily create an object. You can then use that object in the desired operation. The wrapper classes have similar names to the primitive types, but with an uppercase first letter:

- Boolean
- Byte
- Char
- Double
- Float
- Integer
- Long
- Short

It is easy to create a wrapper class object from a primitive value, as shown below for Integer and int:

```
int myInt;


Integer myInteger = new Integer(myInt);
```

The primitive value can then be retrieved from the wrapper object as follows:

```
myInt = myInteger.intValue();
```

Statements

A statement is a logical line of code. A statement may assign a value to a variable or call a function. A statement is the basic building block of your Java program. For example, the following statement assigns the value 5 to the variable x:

```
x = 5;
```

This statement calls a function:

```
calculateIT(x);
```

As you may notice above, statements must end in a semicolon. Exceptions to this rule are statements that begin new blocks of code – that is, sections of code enclosed in curly braces ({}).

-

Blocks

Curly braces ({}) are used to enclose blocks, as shown below:

```
for( i=0; i< 100; i++)
{
    <one or more statements>
}
```

The position of lines breaks is not important. The following example is logically equivalent to that above:

```
for(i=0; i<100; i++) { <one or more statements> }
```

However, most programmers would consider the latter example to be poor style since it hampers readability.

Blocks group a set of statements together. In the above example, using a block here indicates that all the statements in the block should be executed in turn as part of the for loop.

Special Operators

The following section discusses some of the operators you need to understand to write Java code.

The new Operator: Object Creation

To use an object in Java, you must follow a two−step process. First, you must declare an object variable, providing its name and type; second, you must create a new instance of the object’s class. The second step creates a new object for you in memory and assigns it to the variable you’ve declared. You use the new keyword to create a new instance of a class, together with a class constructor.

A Java class always has one or more constructors, which are methods that return a new instance of the class. Every Java class provides a basic constructor method that takes no arguments. Consider the following example:

```
class MyClass
{
   int firstElement;
   float secondElement;
}
MyClass myClassInstance;         // at this point, myClassInstance is null
myClassInstance = new MyClass(); // now myClassInstance refers to an object
```

A Java class may also have constructor methods that take several arguments; these constructors are generally provided as a convenience for the programmer using the object. You decide which one to use based on how you want to store data in the object.

The . Operator: Object Member Access

The . (dot) operator is used to access information or to call an object's methods. For example, given the class definition and variable declaration above, the members of myClassInstance object can be accessed as follows:

```
myClassInstance.firstElement = 1;
myClassInstance.secondElement = 5.0f;
```

The [] Operator: Array Access

In Java, arrays are accessed by integer array indices. Indices for arrays begin with zero (0). Each element of an array can be accessed by its index using the [] (array) operator.

For example, to store an array of six names, you can declare an array as follows:

```
String strNames[5];
```

You can then set the value of first element in the array by assigning an appropriate value:

```
strNames[0] = "Carl";
```

Collections and Iterators

The following section explains collections and iterators.

Collections

Java defines a set of classes known as collection classes. These classes are defined in the java.util package (see more on packages below). The most commonly used collection classes are:

- ArrayList
- HashMap
- LinkedList

Objects created from these classes provide convenient management of sets of other objects. An advantage of a collection over an array is that you don't need to know the eventual size of the collection in order to add objects to it. The disadvantage of a collection is that it is generally larger than an array of the same size. For example, you can create a LinkedList object to manage a set of objects of unknown size by saying:

```
LinkedList l = new LinkedList();



l.Add("Bob");
```

```
l.Add("Mary");


l.Add("Jane");


…
```

Note that in versions of Java before 1.2, the Vector and Hashtable classes were often used to manage collections of objects. These classes, however, are synchronized, meaning they are safe for multi−threaded use. Synchronization has performance implications. If you do not require synchronization behavior, you will achieve better performance by using the newer ArrayList and HashMap classes.

Iterators

Some collection classes provide built−in iterators to make it easy to traverse their contents. The built−in iterator is derived from the java.util.Iterator class. This class enables you to walk a collection of objects, operating on each object in turn. Remember when using an iterators that it contains a snapshot of the collection at the time the iterator was obtained. It's best not to modify the contents of the collection while you are iterating through it.

String and StringBuffer Classes

Java provides convenient string manipulation capabilities via the java.lang.String and java.lang.StringBuffer classes. One of the most common performance−impacting errors new Java programmers make is performing string manipulation operations on String objects instead of StringBuffer objects.

String objects are immutable, meaning their value cannot be changed once they are created. So operations like concatenation that appear to modify the String object actually create a new String object with the modified contents of the original String object. Performing many operations on String objects can become computationally expensive.

The StringBuffer class provides similar string manipulation methods to those offered by String, but the StringBuffer objects are mutable, meaning they can be modified in place.

Javadoc Comments

Java defines a mechanism for attaching information to program elements and allowing automatic extraction of that information. The mechanism, called Javadoc, was originally intended to attach specially formatted comments to classes, methods and variables so that documentation for the classes could be generated automatically. This is how the standard Java API documentation is produced.

Javadoc requires a special opening to Javadoc comments: they must start with /** instead of /*.

Java Tip: While the comment may contain arbitrary text before the first Javadoc tag in the comment, all text after the first Javadoc tag must be part of a tag value. You cannot include free comment text after Javadoc tags; doing so will cause compilation errors.

In addition to the special opening of the comment, Javadoc defines several Javadoc tags that are used to annotate parts of the code. Javadoc tags always start with the @ character. Examples of standard Javadoc tags are:

- @param: The value is a description of a parameter to a method.
- @see: The value is a reference to another class, which will become a hyperlink in the documentation.
- @since: The value is the version number in which this class first appeared.

Although originally intended to produce documentation, Javadoc tags have now found other uses, as they are a convenient and systematic way to attach information to program elements that may be automatically extracted.

WebLogic Workshop uses Javadoc tags to associate special meaning with program elements. For example, the @jws:control tag on a member variable of the class in a JWS file tells WebLogic Workshop to treat the annotated member variable as a WebLogic Workshop control.

Exceptions

Java defines a common strategy for dealing with unexpected program conditions. An exception is a signal that something unexpected has occurred in the code. A method throws an exception when it encounters the unexpected condition. When you call a method that throws an exception, the Java compiler will force you to handle the exception by placing the method call within a try−catch block (see below).

An exception is a Java object, which makes it easy to get information about the exception by calling its methods. Most Java exception objects inherit from the basic java.lang.Exception class. Specific exception classes contain information pertinent to the particular condition encountered. For example, a SQLException object provides information about a SQL error that occurred.

Try−Catch−Finally Blocks

You use try−catch−finally blocks to handle exceptions. The try−catch−finally block allows you to group error handling code in a single place, near to but not intermingled with the program logic.

The following example demonstrates a try−catch−finally block. It assumes that method doSomething() declares that it throws the BadThingHappenedException:

```
public void callingMethod()
{
   try
   {
      doSomething();
   }
   catch (BadThingHappenedException ex)
   {
      <examine exception and report or attempt recovery>
   }
   finally
   {
      <clean up any work that may have been accomplished in the try block>
   }
   return;
}
```

If doSomething() completes normally, program execution continues at the first statement after the try−catch−finally block. Note that the finally block is not executed.

If doSomething()throws the BadThingHappenedException, it will be caught by the catch block. Within the catch block, you can perform whatever action is necessary to deal with the unexpected condition.

The finally block is executed if any portion of the try block executed and an exception was thrown. This give you the opportunity to clean up any partial work that was performed. For example, if a file had been opened in the try block before the exception occurred, the finally block could include code to close the file.

Garbage Collection

Unlike its predecessor compiled languages like C and C++, Java is never compiled all the way to machine code that is specific to the processor architecture on which it is running. Java code is instead compiled into Java byte code, which is machine code for the Java Virtual Machine (JVM). This is the feature that gives Java its inherent portability across different operating systems.

Since all Java code is run in the JVM, the JVM can include capabilities that would be difficult to implement in a traditional programming language. One of the features the JVM provides is garbage collection.

The JVM keeps track of all references to each object that exists. When the last reference to an object is removed, the object is no longer of any use (since no one can reference it, no one can use it). Periodically, or when memory resources are running low, the JVM runs the garbage collector, which destroys all unreferenced objects and reclaims their memory.

For the programmer, this means that you don't have to keep track of memory allocations or remember when to free objects; the JVM does it for you. While Java includes the new keyword, which is analogous to the malloc function in C/C++ and the new operator in C++, Java does not include a free() function or a delete keyword. It is still possible to leak memory in Java, but it is less likely than in other high−level languages.

Packages

The following sections explain Java packages and their relationship to Java classes.

Declaring Packages

All Java code exists within what is know as the package namespace. The package namespace exists to alleviate conflicts that might occur when multiple entities are given the same name. If the entities exist in different portions of the namespace, they are unique and do not conflict.

A package statement may optionally be included at the top of every Java source file. If a file does not specifically declare a package, the contents of the file are assigned to the default package.

Every class and object has both a simple name and a fully qualified name. For example, the Java class String has the simple name String and fully qualified name java.lang.String, because the String class is declared in the java.lang package.

There is a relationship between the package names and the directory hierarchy within which the files exist. A file containing the statement

```
package security.application;
```

must reside in the directory security/application or the Java compiler will emit an error.

Note that packages allow the same class or object name to exist in multiple locations in the package hierarchy, as long as the fully qualified names are unique. This is how the standard Java packages can contain a List class in both the java.util and java.awt packages.

If a file includes a package statement, the package statement must be the first non−comment item in the file.

Importing Packages

Before you can refer to classes or objects in other packages, you must import the package or the specific entities within the package to which you want to refer. To import all classes and objects in the package from the previous example, include the following statement in your file:

```
import security.application.*;
```

Note that the following statement imports only the classes and objects that are declared to be in the security package; it does not import the security.application package or its contents.

```
import security.*;
```

If UserCredentials is a class within the security.application package and it is the only class or object in the package to which you want to refer, you may refer to it specifically with the following statement:

```
import security.application.UserCredentials;
```

If you are only importing a few classes from a large package, you should use the specific form instead of importing the entire package.

Import statements must be the first non−comment items in the file after the optional package statement.

Related Topics

Introducing WebLogic Workshop

Introduction to XML

Extensible Markup Language (XML) is a way to describe structured data in text. Of the many roles XML plays in web services, perhaps the most visible to you as a developer is providing the format for the messages a web service sends and receives. A web service receives method calls, returns values, and sends notifications as XML messages. As you test and debug your service, for example, you might construct XML documents that represent sample messages your service could receive, then pass these documents to your service and view the results.

For more information about the XML standard, see Extensible Markup Language (XML) 1.0 (Second Edition) at the web site of the W3C. The W3C also provides a page with useful links to more information about XML at Extensible Markup Language (XML).

Note: While WebLogic Server handles the task of converting data in Java to and from XML messages, you can exert greater control over your service's use of these messages through XML maps. For more about XML mapping, see Why Use XML Maps?

Anatomy of an XML Document

Let's start with a sample XML document. Imagine that you are writing a web service to return information about employees from a database. Based on first and last name values sent to your service, your code searches the employee database and returns contact information for any matches. The following example is XML that might be returned by your service. Even if you have no experience with XML, you can probably see structure in the example.

```
<!-- Information about the employee record(s) returned from search. -->
<employees>
```

```
    <employee>
        <id ssn="123−45−6789"/>
        <name>
            <first_name>Gladys</first_name>
            <last_name>Cravits</last_name>
        </name>
        <title>Busybody</title>
        <address location="home">
            <street>1313 Mockingbird Lane</street>
            <city>Anytown</city>
            <state>IL</state>
            <zip>12345</zip>
        </address>
    </employee>
</employees>
```

Elements

In XML, text bracketed by < and > symbols is known as an element. Elements in this example are delimited by tags such as <employees> and </employees>, <name> and </name>, and so on. Note that most elements in this example have a start tag (beginning with <) and an end tag (beginning with </). Because it has no content the <id> element ends with a /> symbol. XML rules also allow empty elements to be expressed with start and end tags, like this: <id ssn="123−45−6789"></id>.

Root Elements

Every XML document has a root element that has no parent and contains the other elements. In this example, the <employees> element is the root. The name of the root element is generally based on the context of the document. For example, if your service is designed to return merely an addressadov−−>an <address> element and its contentsadov−−>then the root element is likely to be <address>.

Parent/Child Relationships

Elements that contain other elements are said to be parent elements; the elements that parent elements contain are known as child elements. In this example, <employees>, <employee>, <name>, and <address> are parent elements. Elements they containadov−−>including <employee>, <id>, <name>, and <city>adov−−>are children. (Note that an element can be both a parent and a child.)

Note: This example uses indentation to accentuate the hierarchical relationship, but indentation is not necessary.

Attributes

Attributes are name/value pairs attached to an element (and appearing in its start tag), and intended to describe the element. The <id> element in this example has an ssn attribute whose value is 123−45−6789.

Content

The content in XML is the text between element tags. Content and attribute values represent the data described by an XML document. An element can also be empty. In the example above, “Gladys” is the content (or value) of the <first_name> element.

Comments

You can add comments to XML just as you would with HTML, Java, or other languages. In XML, comments are bracketed with <!−− and −−> symbols (the same symbols used in HTML). The first line of the example above is a comment.

Namespaces

A namespace in XML serves to define the scope in which tag names should be unique. This is important because XML's wide use and textual nature make it likely that you will see occasions where element names with different meanings occur in the same document. For example, namespaces and prefixes are used in XML maps, where a prefix differentiates the tags that are needed for mapping from those associated with the mapped method's XML message.

Schemas

As you gain more experience with XML you may become exposed to schemas. Schemas are not required but can be used to help define what should be in an XML document used for a certain purpose. Whereas XML itself defines basic rules for syntax and structure, a schema defines more stringent rules for the structure and content of an XML document. It can specify which elements and attributes can appear in a document, which elements should be children of others, the sequence of child elements, and so on. If an XML document meets XML's general rules it is called well−formed; if it meets a schema's more specific rules, it is said to be valid.

The following example XML document describes an order of oranges and apples. Compare this document with the schema that follows it.

```
<?xml version="1.0"?>
<produceOrder>
    <item>
        <name>Oranges</name>
        <amount>4</amount>
        <price>2.0</price>
    </item>
    <item>
        <name>Apples</name>
        <amount>12</amount>
        <price>3.6</price>
    </item>
</produceOrder>
```

The following schema is designed to enforce not only the hierarchy of the produce order description, but also the data types allowable for each element. XML schemas use XML syntax, and as a result they can look somewhat like the documents to which they apply.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="produceOrder" type="item"/>
<xs:complexType name="item">
    <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="amount" type="xs:integer"/>
        <xs:element name="price" type="xs:decimal"/>
    </xs:sequence>
</xs:complexType>
```

In this simple example, the hierarchy specifies that the <name>, <amount>, and <price> elements should be children of the <item> element, and the order in which they should appear. The structure of an XML documentadov−−>including its hierarchy, the order of elements, and so onadov−−>is sometimes called its shape.

The xs prefix specifies the namespace against which the tag names should be evaluated for uniqueness. The namespace declaration, which begins with xmlns (for XML namespace), maps the xs prefix to the Uniform Resource Identifier (URI), http://www.w3.org/2001/XMLSchema. In practice, there need not be anything special at the other end of the URIadov−−>it need only be a path that is unique.

The xs prefix also lends specific meaning to the value of type attributes where it is used. Here, types whose values have an xs prefix are known as "simple types," and are defined by the W3C XML Schema specification. Finally, this schema also defines a complex type called item. According to this schema, the item type consists of three XML elements in a specific sequence.

You can use a schema to validate messages received or sent by your web service. When you do so, you can mark your service method's implementation with a custom Javadoc tag that specifies the schema to use. For reference information, see @jws:parameter−xml Tag or @jws:return−xml Tag.

Related Topics

Handling and Shaping XML Message with XML Maps

Handling XML with ECMAScript Extensions

Tutorial: Your First Web Service

This tutorial provides a tour of the major features of WebLogic Workshop, a visual development environment for building web services. Through the tutorial, you build a web service called Investigate, which is designed to receive client requests for a credit report, perform the required research and calculation, and return the report. The tutorial takes a little over three hours to complete.

Tutorial Goals

Through this tutorial, you will learn how to create and test a web service with WebLogic Workshop. Along the way, you will also learn how to create methods that expose a service's functionality, become acquainted with WebLogic Workshop's support for asynchronous communication and loose coupling, and learn about its controls to speed the design process.

Tutorial Overview

The Investigate web service you will build with this tutorial is designed to collect credit−related information about the applicant, compute a credit worthiness score and rating, then return the combined information to the client.

There are six actors in this scenario:

- The client of your web service. Clients of Investigate are loan or credit application processing systems. An example might be a credit card application processing system at a department store. These systems will use your Investigate web service to determine the applicant's credit worthiness. They will supply you with the applicant's taxpayer ID, and your service will compute and return a credit score.
- Your Investigate web service. The service will receive a taxpayer ID as part of requests from clients and respond with information about the applicant's credit worthiness.
- An database containing bankruptcy information about the applicant.

- A credit card reporting web service with information about the applicant's credit history.
- A credit scoring application designed to calculate a credit score based on information you've collected.
- An Enterprise Java Bean (EJB) designed to provide a credit rating based on the score.

This tutorial guides you through the process of adding functionality in increments, and shows you how to test your web service as you build it.

Steps in This Tutorial

Step 1: Begin the Investigate Web Service adov−−> 30 minutes

In this step you build and run your first WebLogic Workshop web service. You start WebLogic Workshop and WebLogic Server, then create a web service that has a single method.

Step 2: Add Support for Asynchronous Communication adov−−> 25 minutes

You will work around a problem inherent in communication across the web: the problem of network latency. Network latency refers to the time delays that often occur when transmitting data across a network.

Step 3: Add a Database Control adov−−> 20 minutes

You add a control for access to a database, then query the database for bankruptcy information.

Step 4: Add a Service Control adov−−> 15 minutes

You use a ServiceControl to invoke another web service, collecting credit card data on the applicant.

Step 5: Add a JMS and EJB Control adov−−> 20 minutes

You will take the data you have gathered and request help from an in−house application to retrieve a credit score. You will then send the credit score to an Enterprise Java Bean (EJB) that will use it to measure the applicant's credit risk.

Step 6: Add Script for Mapping adov−−> 15 minutes

You will apply an XML map to translate data stored within the Investigate service into a particular XML shape for delivery to the client.

Step 7: Add Support for Cancellation and Exception Handling adov−−> 15 minutes

You enhance the Investigate web service to better handle these possibilities such as the client's desire to cancel the request, an overly long wait for a response from the credit card service, and exceptions thrown from your service's methods.

Step 8: Client Application adov−−> 20 minutes

You will build a Java console client to invoke your web service. You also modify your web service to support a polling interface, an alternative to its current design.

Step 9: Deployment and Security adov−−> 10 minutes

You modify your web service for greater security, exposing it through HTTPS (Secure HTTP) instead of HTTP. You will also package the web service for deployment on a production server.

Summary: Your First Web Service adov−−> 15 minutes

You review the concepts and technologies covered in the tutorial. This step provides additional links for more information about each area covered.

To begin the tutorial, see Step 1: Begin the Investigate Web Service.

Click the arrow to navigate to the next step.

Step 1: Begin the Investigate Web Service

In this step, you will learn to start WebLogic Workshop and use it to begin a new web service. WebLogic Workshop is the graphical tool you use to create, implement, test, and debug web services. When working in WebLogic Workshop, you use projects to group files associated with a web service or related web services.

With WebLogic Workshop, you can also start and stop WebLogic Server. WebLogic Server provides functionality that supports web services you build with WebLogic Workshop, and so must be running while you are building a web service.

In this tutorial, you will learn how:

- To start WebLogic Workshop on Microsoft Windows
- To start WebLogic Workshop on Linux
- To open the samples project
- To start WebLogic Server
- To create a new web service
- To add a method
- To add a parameter and return value
- To add code for returning a value
- To launch Test View
- To test the requestCreditReport method

Starting WebLogic Workshop

The first step in any process is to start the program. These sections explain how to do that on Microsoft Windows, Linux, and UNIX platforms.

To Start WebLogic Workshop on Microsoft Windows

- From the Start menu, choose Programs−−>WebLogic Platform 7.0−−>WebLogic Workshop.

To Start WebLogic Workshop on Linux

1. Open a file system browser or shell.
2. Locate the **Workshop.sh** file at the following address:

```
$HOME/bea/weblogic700/workshop/Workshop.sh
```

3. In the command line, type the following command:

sh Workshop.sh

When you start WebLogic Workshop for the first time after installing it, it opens to display the samples project. This project contains sample web services included with WebLogic Workshop. After it has been used for the first time, WebLogic Workshop displays the project last opened.

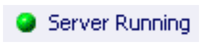
For this tutorial, you will use the samples project.

To Open the Samples Project

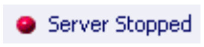
1. Choose File−−>Open Project. The Open Project dialog appears.
2. In the Projects on your development server box, select samples.
3. Click Open.

Once you have opened the samples project, you must ensure that WebLogic Server is running while you build your service. All of the web services you create using WebLogic Workshop run on WebLogic Server on your development machine until you deploy them. When your web service is called by other components, WebLogic Server is responsible for invoking it.

You can confirm whether WebLogic Server is running by looking at the indicator in the status bar at the bottom of WebLogic Workshop. If WebLogic Server is running, there is a green ball as pictured here:


Server Running

If instead you see the a red ball, as in the following picture, then you must start WebLogic Server before proceeding:


Server Stopped

To Start WebLogic Server

In WebLogic Workshop, choose **Tools−−>Start WebLogic Server**.

Creating a New Web Service

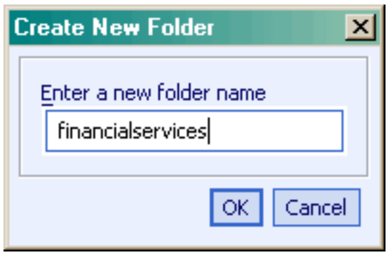
Once WebLogic Workshop and WebLogic Server are running, you can create a new web service. When you create a new web service, you also create a new JWS file. A JWS file is very much like a JAVA file in that it contains code for a Java class. However, because a file with a JWS extension contains the implementation code intended specifically for a web service class, the extension gives it special meaning in the context of WebLogic Server.

To Create a New Web Service

1. In the **Project Tree**, right−click **Project 'samples'** and select New Folder The Create New Folder dialog appears.

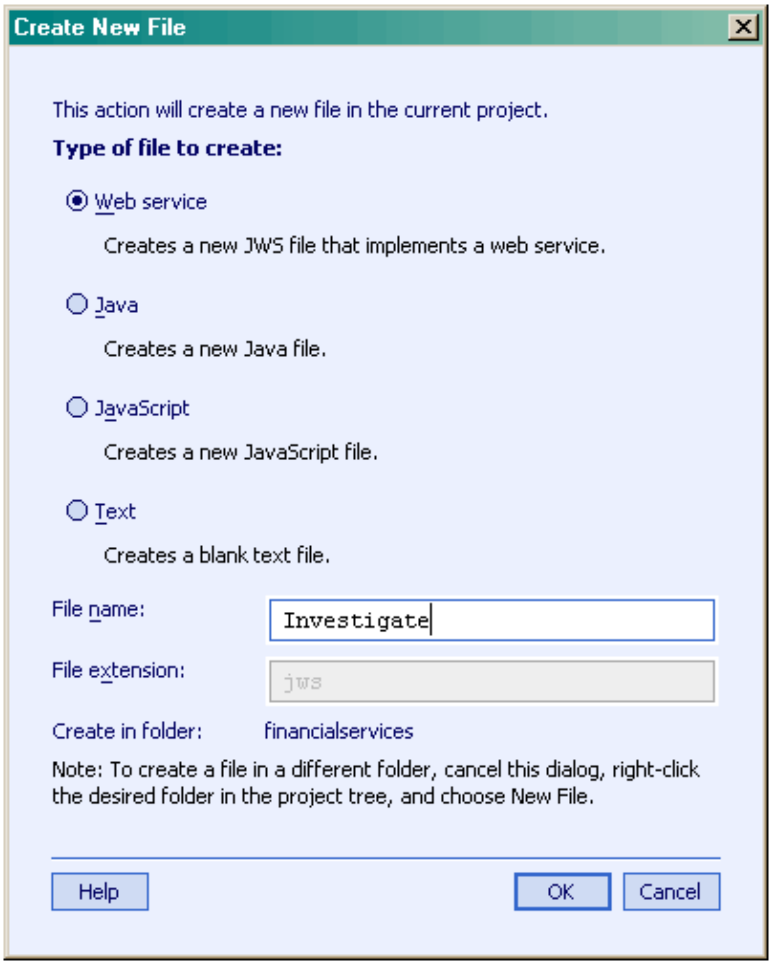
Note:The Project Tree is located at the left side of the screen.

2. In the **Enter a new folder name** box, type financialservices , as shown here:



financialservices

3. Click **OK**. The **financialservices** folder appears in the project tree.
4. Right−click the **financialservices** file and select **New File**. The **Create New File** dialog appears.
5. Confirm that the Web service radio button is selected, as shown in the following picture.



This action will create a new file in the current project.

Type of file to create:

● Web service

Creates a new JWS file that implements a web service.

○ Java

Creates a new Java file.

○ JavaScript

Creates a new JavaScript file.

○ Text

Creates a blank text file.

File name: Investigate

File extension: jws

Create in folder: financialservices

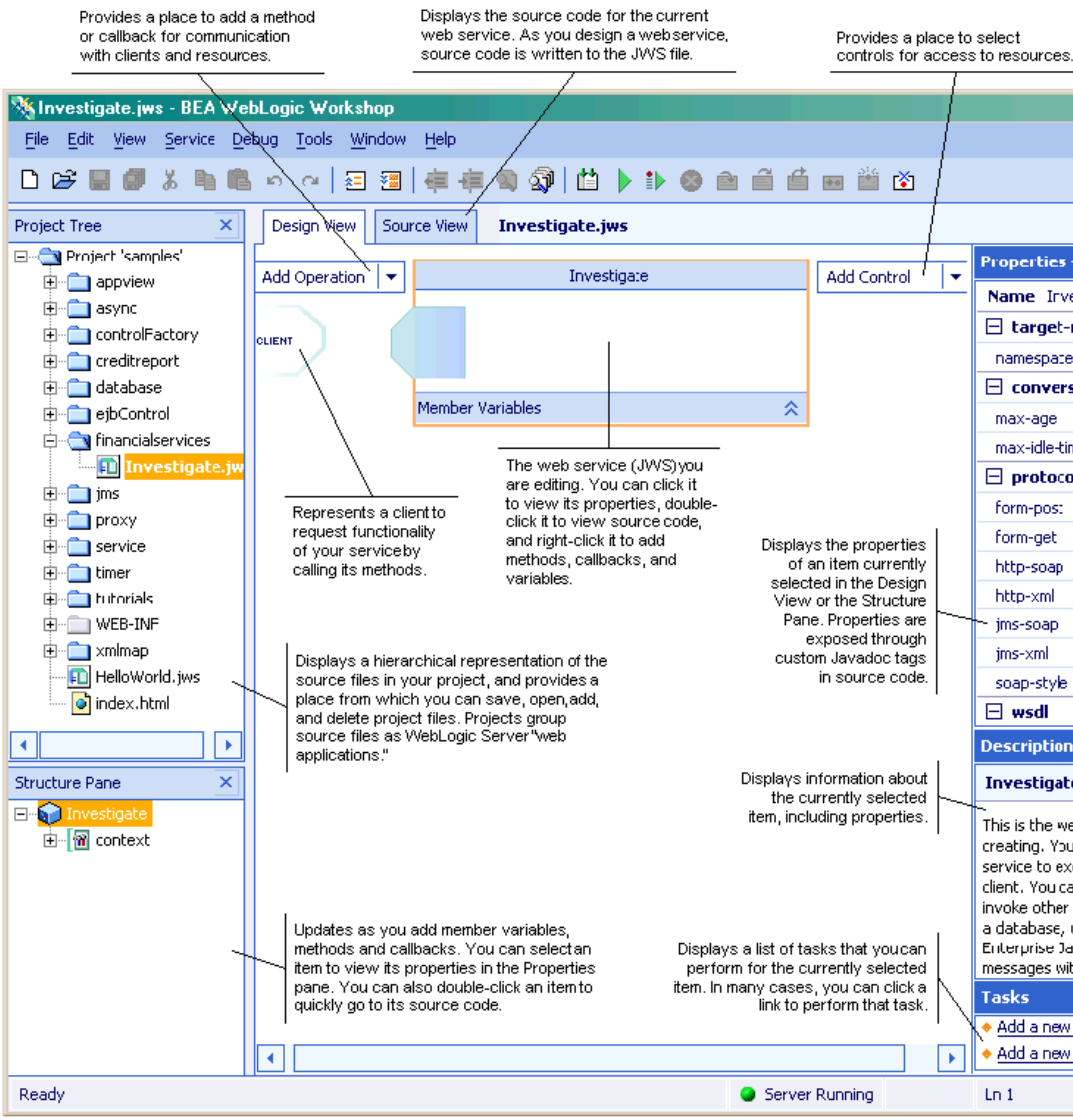
Note: To create a file in a different folder, cancel this dialog, right-click the desired folder in the project tree, and choose New File.

Note: The Java, JavaScript, and Text options provide a way for you to add those kinds of files to your project. For now, however, leave Web service selected.

5. In the File name field, type Investigate. This will be the name of your web service class.

6. Click OK.

Once you create a new service project, WebLogic Workshop displays your new JWS file, entitled Investigate.jws, in Design View. The following screen shot provides a quick tour of your project in Design View.



Provides a place to add a method or callback for communication with clients and resources.

Displays the source code for the current web service. As you design a web service, source code is written to the JWS file.

Provides a place to select controls for access to resources.

Represents a client to request functionality of your service by calling its methods.

The web service (JWS) you are editing. You can click it to view its properties, double-click it to view source code, and right-click it to add methods, callbacks, and variables.

Displays the properties of an item currently selected in the Design View or the Structure Pane. Properties are exposed through custom Javadoc tags in source code.

Displays a hierarchical representation of the source files in your project, and provides a place from which you can save, open, add, and delete project files. Projects group source files as WebLogic Server "web applications."

Displays information about the currently selected item, including properties.

Updates as you add member variables, methods and callbacks. You can select an item to view its properties in the Properties pane. You can also double-click an item to quickly go to its source code.

Displays a list of tasks that you can perform for the currently selected item. In many cases, you can click a link to perform that task.

Graphical Design of Services

In Design View you can design your service in a graphical manner. You create what is effectively a drawing of your service and its interactions with clients and other resources. As you design your service, WebLogic Workshop generates source code that you can edit in the Source View. Through this source code you can add logic behind your design. This enables you to create web services by focusing on application logic rather than on code for infrastructure.

While designing your service, you can add methods and events. You can add controls to represent resources such as other services, databases, and Enterprise Java Beans. You can also specify support for powerful features of the underlying server by setting properties for items in your design.

Note: A good way to get a sense of the design tasks you can perform is to click an item in your design, then view the tasks listed in the **Tasks** pane in the lower right corner of Design View.

One of the key features of WebLogic Workshop is tight integration between tasks you do while designing your web service and the changes you make to source code. For example, if you add a member variable to source code in Source View, your new variable will appear in Design View, and vice versa. This integration is also reflected in the Structure Pane. There you can find a list of items in your service's design, including methods, events, variables, and so on. You do not need to save or refresh the file to cause changes in one view to appear in other views.

For more complete information about projects and the files they contain, see WebLogic Workshop Projects.

Adding the requestCreditReport Method

Web services expose their functionality through methods, which clients invoke to make requests. In this case, clients will use a method you create to request credit reports. The service you are building will eventually collect credit information from other sources. But for now, to keep things simple, you will provide a method that returns a simple report immediately. This web service will provide clients with credit ratings for loan or credit applications. So your web service will need to provide the client with a way to make the initial request. You can do this by adding a requestCreditReport method to the Investigate web service.

To Add a New Method

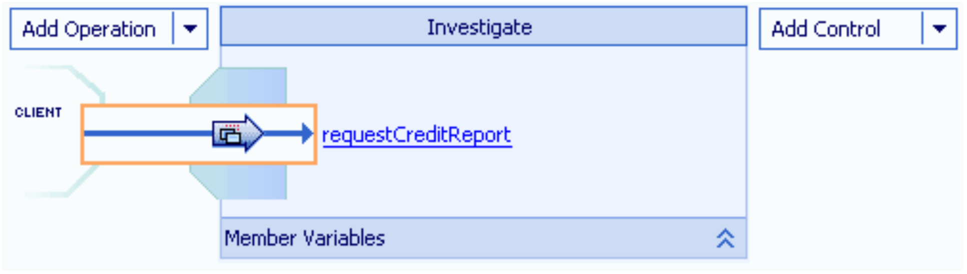
1. If it is not selected already, click the **Design View** tab to ensure that you are viewing the design for the Investigate web service.

Note: It is not necessary to be in Design View while adding methods, but it is helpful to see how changes update the design of your service.

2. From the **Add Operation** drop−down list, select Add Method.
3. In the space provided, replace method1 with requestCreditReport and press Enter.

Note: If you switched from WebLogic Workshop after adding the method (for example, to read this topic), the method name may no longer be available for editing. To re−open it for editing, right click its name, click Rename, then type requestCreditReport. Then press Enter.

Your design of the Investigate web service should now resemble the following illustration:



Adding a Parameter and Logic

When making a request, the client needs to supply information about the applicant in order for the Investigate file to do its work. For this scenario, the client provides a taxpayer ID. For simplicity, you will enable this method to receive the number as a string. You will also add a return value so that the method can send back to the client the requested results.

To Add a Parameter and Return Value

1. In Design View, double−click the arrow corresponding to the **requestCreditReport** method.

The **Edit Maps and Interface** dialog appears.

2. In the **Java** pane in the lower section of the dialog, edit the text so that it appears as follows:

```
public String requestCreditReport(String taxID)
```

This text is known as the method's declaration. The first occurrence of "String" is the return data type; the text "String taxID" is the parameter data type and parameter name.

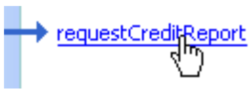
3. When you have finished editing the method declaration, click **OK**.

Now you can add a little code to make the method do something. The code you add here will return a string value every time the requestCreditReport method is called. Obviously, this is not very useful, but you will add more intelligent functionality later.

You change a method's functionality by editing its source code.

To Add Code for Returning a Value

1. In Design View, click the name of the method, as shown in the following illustration.



The following source code appears:

```
/** * @jws:operation */ public String requestCreditReport(String taxID) { }
```

WebLogic Workshop added this source code while you worked in Design View.

2. Edit the requestCreditReport method code so that it returns a String. The returned code resembles the following:

```
/** * @jws:operation */ public String requestCreditReport(String taxID) {    return "applicant: " + taxID + " is approved."; }
```

3. Press Ctrl+S to save your work.

As it is now written, the code for the **requestCreditReport** method returns a string containing the taxpayer ID the client sent with the method call and a note that the applicant was approved. Remember to press Ctrl+S to save your changes to the web service.

Notice that a comment precedes the method that contains an @jws:operation tag. This is a Javadoc comment (Javadoc comments always begin with /** and end with */). Briefly, Javadoc is a simple system for including program structure−related information in comments, indicated by tags, that may be automatically extracted to produce documentation or other material related to structure of a Java class. Its original use was to specify inline documentation that could be extracted to HTML. WebLogic Workshop uses a variety of Javadoc tags to indicate how web service code should be treated by WebLogic Server.

The @jws:operation tag indicates that the method to which it is attached should be exposed as a web service method that clients can call. The term *operation* refers to the actions web services expose to clients, such as the **requestCreditReport** method you have added here. For more information on Javadoc, please refer to the Javadoc pages at java.sun.com.

Launching a Browser to Test the Service

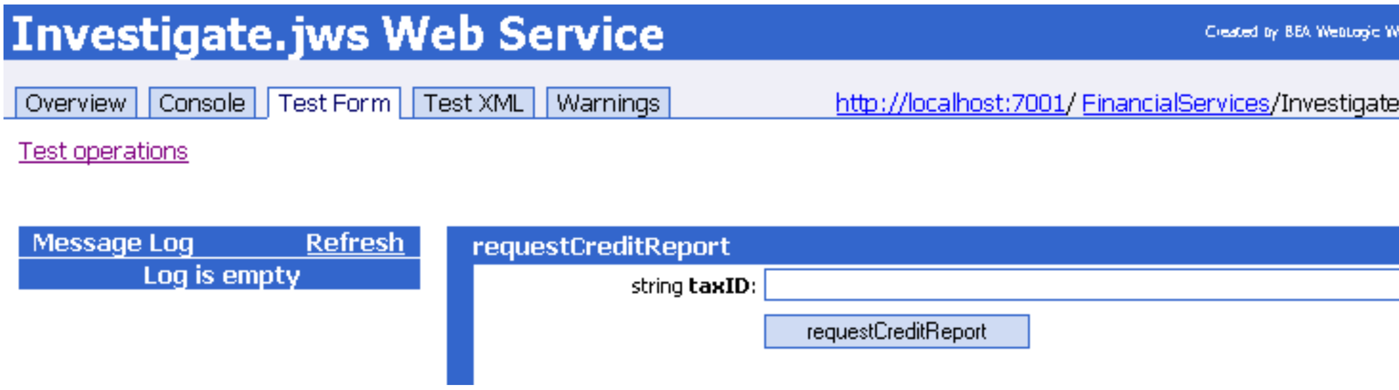
The best way to test a web service while you're writing and debugging code is to use the Test View, a browser−based tool through which you can call methods of your web service. In this tutorial, you will be using two buttons in WebLogic Workshop to control display of the Test View.

To Launch Test View

- Click the Start button, which looks like the following illustration:

Note: If you want to stop Test View, click the Stop button shown here:

Clicking the Start button prompts WebLogic Workshop to build your project, checking for errors along the way. WebLogic Workshop then launches your web browser to display a page through which you can test the service method with possible values. The following illustration shows the Test Form page that appears after you click the Start button.



As you can see, this page provides tabs for access to other pages of the Test View. Test Form and Test XML provide alternative ways to specify the content of the call to your service's methods. Overview and Console include additional information and commands useful for debugging a service. The Warnings tab provides feedback about your service. These tabs are described later in this topic. First, here is a little more information about what you are seeing.

Exploring the Test Form Tab

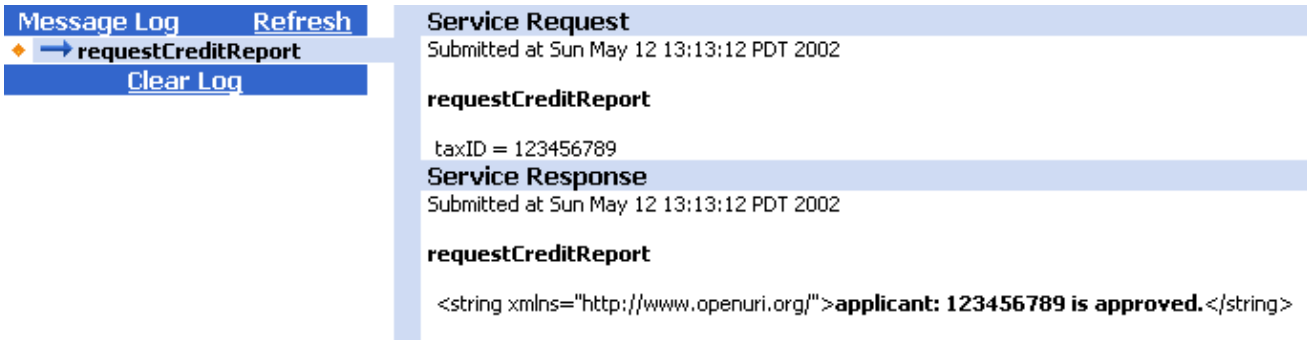
The Test Form tab provides a place for you to test a service with values. You can enter values for the parameters of your methods. For the Investigate service, there is one method with one parameter; as a result, the page provides a box into which you can enter a value for the taxID parameter. When you click the requestCreditReport button, WebLogic Workshop invokes the requestCreditReport method, passing the contents of the text box as a parameter to the method.

To Test the requestCreditReport Method

1. If it is not already selected, click the **Test Form** tab.
2. In the **string taxID** field, enter a value. This value can be any string value, but for the sake of the test it should be something that at least looks like a taxpayer ID, such as 123456789.

3. Click requestCreditReport.

The Test Form page refreshes to display a summary of your request parameters and your service's response, as shown here:



Service Request
Submitted at Sun May 12 13:13:12 PDT 2002

requestCreditReport

taxID = 123456789

Service Response
Submitted at Sun May 12 13:13:12 PDT 2002

requestCreditReport

<string xmlns="http://www.openuri.org/">**applicant: 123456789 is approved.**</string>

Under Service Request, the summary displays the essence of what was sent by the client (you) when the method was called. It shows the parameter values passed with the method call, as in the following example:

taxID = 123456789

Under Service Response, the summary displays what was returned by the Investigate service. The response is formatted as a fragment of Extensible Markup Language (XML), as communications with web services are formatted as XML messages. You can see that the method returned the correct result for the code you provided, as shown here:

```
<string xmlns="http://www.openuri.org/">applicant: 123456789 is approved.</string>
```

To the left of the request and response is the Message Log area. This area lists a separate entry for each test call to the service method. Entries in the Message Log correspond to your service's methods, updating with a new entry for each test you make.

To try this out, click the Test operations link to return to the original Test Form page, then enter a new value in the string taxID box and click requestCreditReport. When the page refreshes, request and response data corresponding to your second test is displayed as a second entry in the Message Log. You can click each log entry to view the data for the corresponding test. Click Clear Log to empty the list of log entries and start fresh.

That's it for the test! With this simple test, you have pretty well exercised this service's abilities adov−−> so far.

But before you move on, take a look at the other information available from Test View. For example, the URL at the upper right corner of the page next to the Warnings tab should appear something like this:

http://localhost:7001/samples/financialservices/Investigate.jws

This is the URL used to call the web service. Each portion of the URL has a specific meaning. The following list explains each portion:

- http://locahost:7001/ adov−−> This means that your browser's requestadov−−>that is, the call to the serviceadov−−>will be intercepted by WebLogic Server, which is listening on port 7001 of your local machine. In your case, the term "localhost" is probably replaced by the name of your computer.
- samples/financialservices/ adov−−> "samples" refers to the web application of which the service is a part. When you create a project in WebLogic Workshop, you are also creating a WebLogic Server "web application." The project name becomes part of the URL for all web services in that project. You will want to keep that in mind when naming new projects so that the resulting web service URLs are meaningful and appropriate.

- Investigate.jws adov−−> This is the name of the web service's JWS file. WebLogic Server is configured to recognize the JWS extension and respond appropriately by serving the request as a web service rather than, for example, an HTML page or a Java servlet.

Exploring the Overview Page

The Overview page deserves special attention because it provides access to all of the files you and your service's clients will need to use your web service. As you update your service's design and source code, you will be able to easily get current versions of these files. As a result, you may find the Overview page to be an invaluable resource as you begin testing and using it in a deployed application.

The following list describes some of the prominent features you will find on the Overview page:

Web Service Description Language files adov−−> Web Service Definition Language (WSDL) files describe your service's interface, giving potential clients what they need to know in order to use your service's functionality.

Web Service Clients adov−−> The following list provides brief descriptions of these links. They will be covered in more detail later in this tutorial.

- The Workshop Control link displays the contents of a CTRL file. A CTRL (or "control") file provides your service's interface in a format specific to WebLogic Workshop.
- The Java Proxy link downloads a JAR file containing code a Java client can use to call the Investigate service.
- The Proxy Support Jar link downloads a JAR file containing supporting classes for clients written in Java.

Service Description adov−−> A list of the methods and callbacks (if any) that are exposed by the web service. This is for verification purposes. If you don't see all of the methods you expected in this list, it indicated a problem with the source code for the web service. Here, you can see that the Investigate web service currently has only the requestCreditReport method and no callbacks.

Useful Links adov−−> A link to a topic on building client proxies, as well as links to the specifications behind the standards for the following languages:

- WSDL, a language (based on the rules of Extensible Markup Language, or XML) for describing a service's interface.
- Simple Object Access Protocol (SOAP), a protocol for formatting the XML−based messages that web services use to communicate.
- XML namespaces, a means to prevent conflicts among XML element and attribute names (but ensuring that names are unique within a given scope of use).

- URIs, strings used to identify a resource, including XML namespaces and schemas.

Exploring the Console Page

The Console page provides information and commands related to the instance of WebLogic Server running on your computer. Through the settings and buttons on this page, you can control aspects of your local instance of WebLogic Server while you are testing your service.

Exploring the Test XML Page

When your web service is deployed and receiving requests from clients, those requests will come in the form of XML messages carrying request−specific information. The Test XML page provides a place for you to test your service with something like the body of an XML message that might be sent to your service. You can try it out now by replacing the placeholder value with the Taxpayer ID you tested with earlier, then clicking requestCreditReport.

```
<requestCreditReport xmlns="http://www.openuri.org/">
  <taxID>
    111111111
  </taxID>
</requestCreditReport>
```

You have successfully added code to a web service’s method and tested the method for correct operation. In the next step, Step 2: Add Support for Asynchronous Communication, you will revise your service's design to solve a problem inherent in communications across the Internet adov−−> latency.

Related Topics

WebLogic Workshop Projects

Design View

Test View

Click one of the following arrows to navigate through the tutorial.

Step 2: Add Support for Asynchronous Communication

In this step, you will solve a problem inherent in communication across the web: the problem of network latency. Network latency refers to the time delays that often occur when transmitting data across a network.

This can be a particular problem on the internet, which has highly unpredictable performance characteristics and uncertain reliability. There may be additional latency if it takes a long time for a web service to process a request (for example, if a person "behind" the service must review the applicant's credit before a response can be returned).

So far, the design of the Investigate web service forces the client calling the requestCreditReport method to halt its own processes and wait for the response from the web service. This is called a synchronous relationship, in which the client software invokes a method of the web service and is blocked from continuing its own processes until it receives the return value from the web service. As you might imagine, this won't work well when it is difficult to predict the time it will take for the client to receive a response.

You will solve the latency problem by enabling your web service to communicate with its clients asynchronously. In asynchronous communication, the client communicates with the web service in such a way that it is not forced to halt its processes while the web service produces a response. In particular, you will add a method to the web service that immediately returns a simple acknowledgement to the client, thereby allowing the client to continue its own processes. You will also add a callback that returns the full results to the client at a later time. Finally, you will implement support for conversations that enable your web service to remember which client to send the full response to via the callback.

The tasks in this step are:

- To add a class and a member variable to the web service

- To add a requestCreditReportAsynch method to the web service
- To add an onCreditReportDone callback to the web service
- To add code that sends data back to the client
- To add a buffer and conversation support to the web service
- To test the web service

To Add a Class and Member Variable to the Web Service

The code you add below has two parts: a class, called Applicant, and a member variable, called m_currentApplicant. This variable is an instance of the Applicant class, which serves as a data structure to record the information about credit applicants. Its fields hold information about an applicant's name, currently available credit, and so on. While the member variable m_currentApplicant stores data about a particular applicant, the web service builds a profile about that particular applicant.

1. Click the **Source View** tab. From this tab you can view the web service's Java code.
2. Place the following code within public class Investigate:

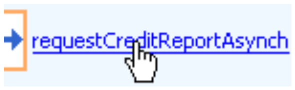
```
public static class Applicant implements java.io.Serializable
{
    public String taxID;
    public String firstName;
    public String lastName;
    public boolean currentlyBankrupt;
    public int availableCCCredit;
    public int creditScore;
    public String approvalLevel;
    public Applicant(String taxID)
    {
        this.taxID = taxID;
    }
    public Applicant() {}
}
Applicant m_currentApplicant = new Applicant();
```

To Add a requestCreditReportAsynch Method to the Web Service

Next you will add a method to be invoked by client software applications that receives a report on an applicant's credit worthiness. The method itself simply returns an acknowledgement to the invoking client without returning any substantial information on the applicant. The finished report on the applicant's credit worthiness is returned in the onCreditReportDone callback, which you will add later.

1. Click the Design View tab.
2. From **Add Operation** drop−down list, select **Add Method**.

3. In the input box that appears, type the method name requestCreditReportAsynch and press Enter.
4. Click the method name, as shown here:



The method code appears in Source View.

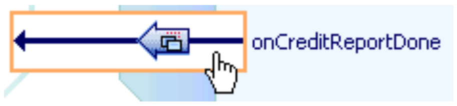
5. Edit the requestCreditReportAsynch method code so that it appears as follows:

```
/** * @jws:operation */ public void requestCreditReportAsynch(String taxID)    {
      m_currentApplicant.taxID = taxID;    }
```

To Add an onCreditReportDone Callback to the Web Service

Once the web service is finished building a credit profile of a credit applicant, you want to add a callback to send a credit report back to a client application. The callback you add below will return the finished report on an applicant's credit worthiness.

1. Click the Design View tab to return to Design View.
2. From the Add Operation drop−down list, select Add Callback.
3. In the input field that appears, type the callback name onCreditReportDone and press Enter.
4. Double−click the arrow associated with the onCreditReportDone callback. The Edit Maps and Interface dialog box appears.



5. In the Java pane, edit the text so that it appears as follows:

```
public void onCreditReportDone(Applicant m_currentApplicant, String responseMessage)
```

6. Click OK. The Edit Maps and Interface dialog closes.

To Add Code that Sends Data Back to the Client

Next, you can edit the requestCreditReportAsynch method so that it initiates the onCreditReportDone callback .

1. If you are not in Source View, click the Source View tab.
2. Modify the requestCreditReportAsynch method to look like the following:

```
/** * @jws:operation */ public void requestCreditReportAsynch(String taxID) {
      m_currentApplicant.taxID = taxID;       callback.onCreditReportDone(m_currentApplicant, null); }
```

To Add a Buffer and Conversation Support to the Web Service

Now you are ready to add a buffer and conversation support to your web service.

The buffer, which you will add to the requestCreditReportAsynch method, serves two purposes. First, it saves client requests in a message queue which prevents requests from being lost during server outages. Second, it immediately returns an acknowledgement to the requesting client, which allows the client to continue processing without waiting for the full response from the web service.

Adding conversation support to your web service lets the client know for sure that the result your service sends back through the onCreditReportDone callback is associated with the request the client originally made. If the same client makes two independent calls with the same taxpayer ID (say, for separate loan applications from the same applicant), how will it know which returned result corresponds to which request? With all the interaction back and forth between the client and your web service, your service needs a way to keep things straight.

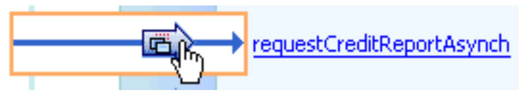
Also, if the server goes down (taking the m_applicantID member variable with it), the data that the client sent as a parameter is lost. The service not only loses track of the client's request, but of who the client was in the first place.

You can solve these problems by associating each exchange of information (request, response, and any interaction needed in between) with a unique identifier—an identifier known to both the client and the service.

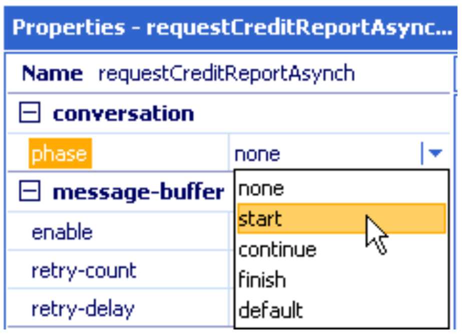
In web services built with WebLogic Workshop, you do this by adding support for a conversation. For services participating in a conversation, WebLogic Server stores state−related data, such as the member variable you added, on the computer's hard drive and generates a unique identifier to keep track of the data and of which responses belong with which requests. This infrastructure remains in place as long as the conversation is ongoing. When the conversation is finished, the resources allocated to the infrastructure are released.

When you add methods to a web service that supports conversations, you can indicate whether each method starts, continues or finishes a conversation. Adding support for a conversation is very easy. With methods that represent both the first step of the transaction (requestCreditReportAsync) and the last step (onCreditReportdone), you want to indicate when the conversation starts and when it finishes. To add a buffer and conversation support to your web service follow the steps below.

1. Click the Design View tab to return to Design View.
2. Edit the Properties Pane associated with the requestCreditReportAsynch method. To edit the Propeties Pane for this method click the arrow next to the method name requestCreditReportAsynch. Note: do not click the method name requestCreditReportAsynch, instead click its associated arrow, as shown here:



3. In the Properties pane, from the phase drop−down list, select start, as shown here:


Properties - requestCreditReportAsync...

Name requestCreditReportAsynch

conversation

phase: none

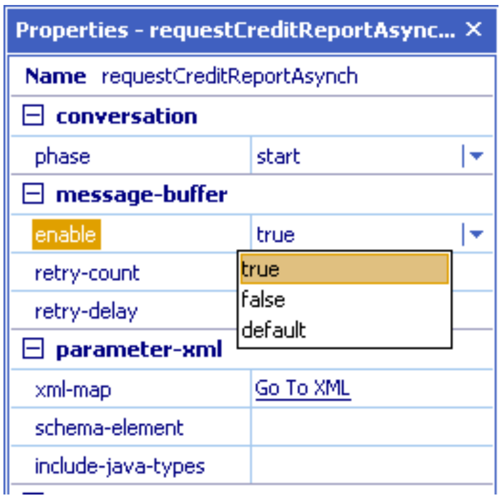
message-buffer

enable

retry-count

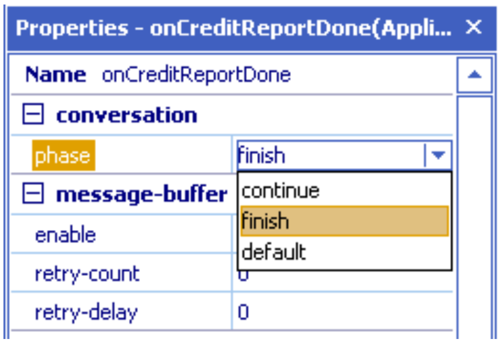
retry-delay

4. In the message−buffer section, from the enable drop−down list, select true, as shown here:


Properties - requestCreditReportAsync...

Name	requestCreditReportAsynch
⊟ conversation	
phase	start
⊟ message-buffer	
enable	true
retry-count	
retry-delay	
⊟ parameter-xml	
xml-map	Go To XML
schema-element	
include-java-types	

5. Click the arrow next to the onCreditReportDone callback.
6. In the Properties pane, from the phase drop−down list, select finish, as shown here:


Properties - onCreditReportDone(Appli...

Name	onCreditReportDone
⊟ conversation	
phase	finish
⊟ message-buffer	
enable	
retry-count	0
retry-delay	0

7. Press Ctrl+S to save the service.

To Test the Web Service

Now you are ready to compile and test the service.

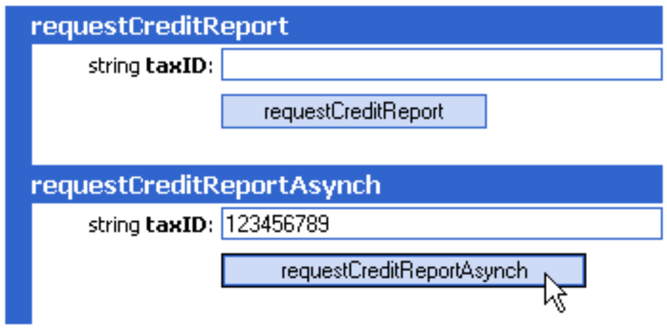
1. Compile and test the web service by clicking the Start button, shown here:

Your web service compiles and a browser window is launched that displays the Test page.

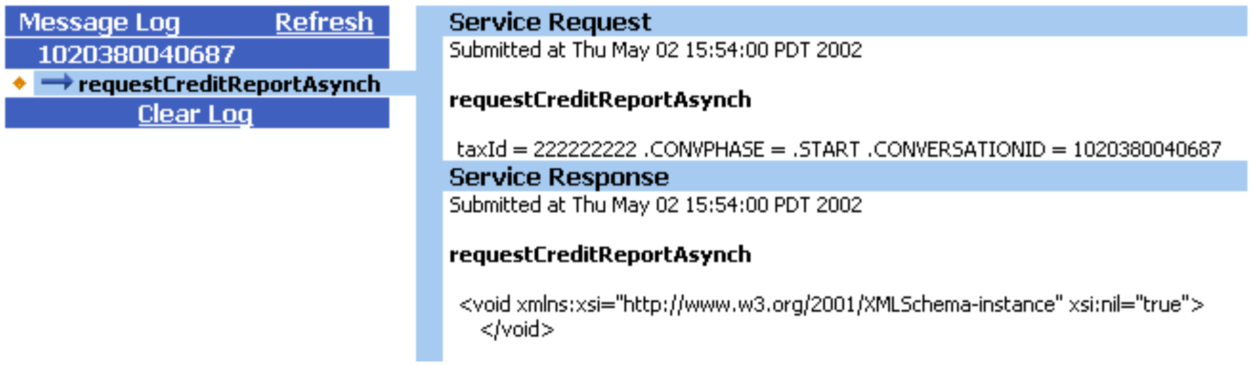
2. Type a number value in the taxID field, as shown below.

Note: Use one of the following (9 digit) taxID's to test your web service throughout the tutorial:

123456789, 111111111, 222222222, 333333333, 444444444, and 555555555.



3. Click requestCreditReportAsynch. The Test page refreshes to display a summary of the request parameters sent by the client and your service's response, as shown here:



Service Request
Submitted at Thu May 02 15:54:00 PDT 2002

requestCreditReportAsynch

taxId = 222222222 .CONVPHASE = .START .CONVERSATIONID = 1020380040687

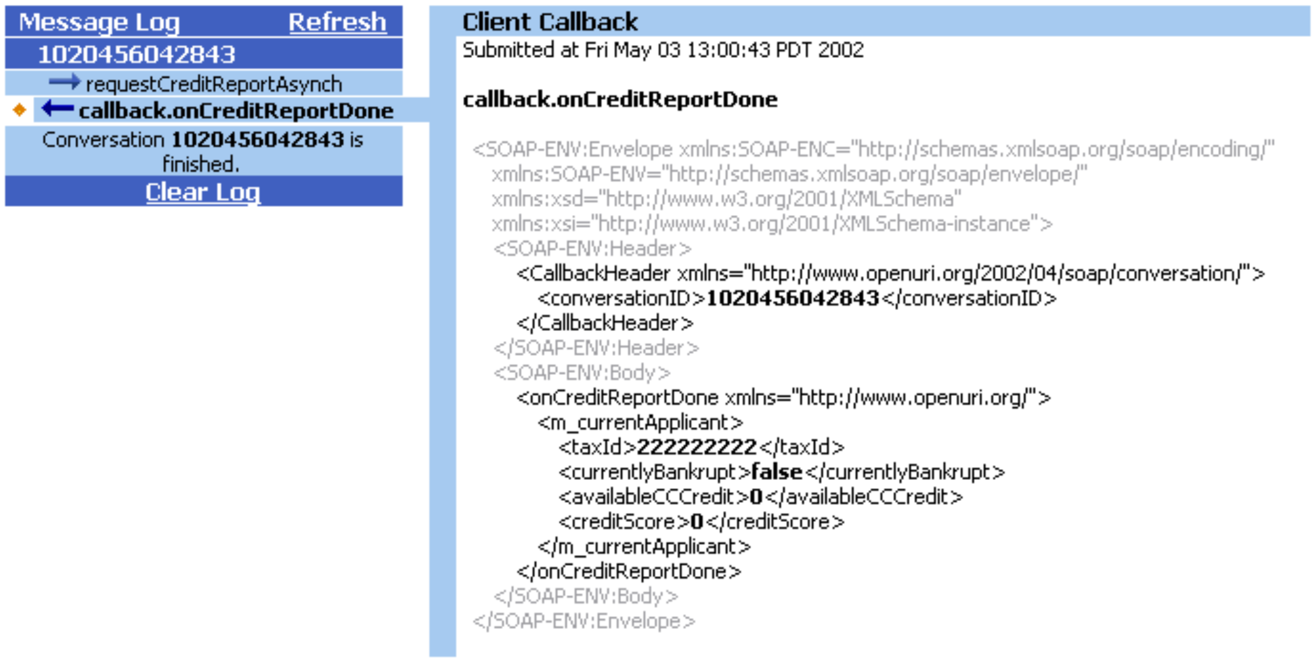
Service Response
Submitted at Thu May 02 15:54:00 PDT 2002

requestCreditReportAsynch

```
<void xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:nil="true">
  </void>
```

Note: Under Service Request, the summary displays what the client sent to the web service to invoke the requestCreditReportAsynch method. Notice that is displays the taxID which you sent to the web service as well as the conversation identification number, which the service uses to associate the initial client request and the callback response it will later send back to the client.

4. Under Service Response, the summary displays what the web service has sent back (so far) to the client. In this case it sends back an XML file serving as an acknowledgement to the client that its request has been received.
5. Refresh the browser. Notice that the Message Log has the new entry callback.onCreditReportDone.
6. Click callback.onCreditReportDone to view the contents of the callback sent from the web service to the client, as shown here:



Conversation 1020456042843 is finished.

Client Callback

Submitted at Fri May 03 13:00:43 PDT 2002

callback.onCreditReportDone

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <CallbackHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
      <conversationID>1020456042843</conversationID>
    </CallbackHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <onCreditReportDone xmlns="http://www.openuri.org/">
      <m_currentApplicant>
        <taxId>222222222</taxId>
        <currentlyBankrupt>false</currentlyBankrupt>
        <availableCCCredit>0</availableCCCredit>
        <creditScore>0</creditScore>
      </m_currentApplicant>
    </onCreditReportDone>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The contents display the SOAP message that the client receives, including data about the current applicant. Of course, since our web service has no way of learning anything about a credit applicant, the information send back to the client consists of the taxID originally entered, along with default values for the other fields.

In the next step of the tutorial you will add a database control to your web service, which will enable your web service to acquire substantive data about a credit applicant.

Related Topics

[User Interface Reference]

[Overview: Conversations]

Step 3: Add a Database Control

In this step you add a database control to your web service. The database control provides your web service access to a database containing bankruptcy information about credit applicants. Controls, i.e., CTRL files, act as interfaces between your web service and other data resources, such as databases, other web services, Java Message Services, etc.

The tasks in this step are:

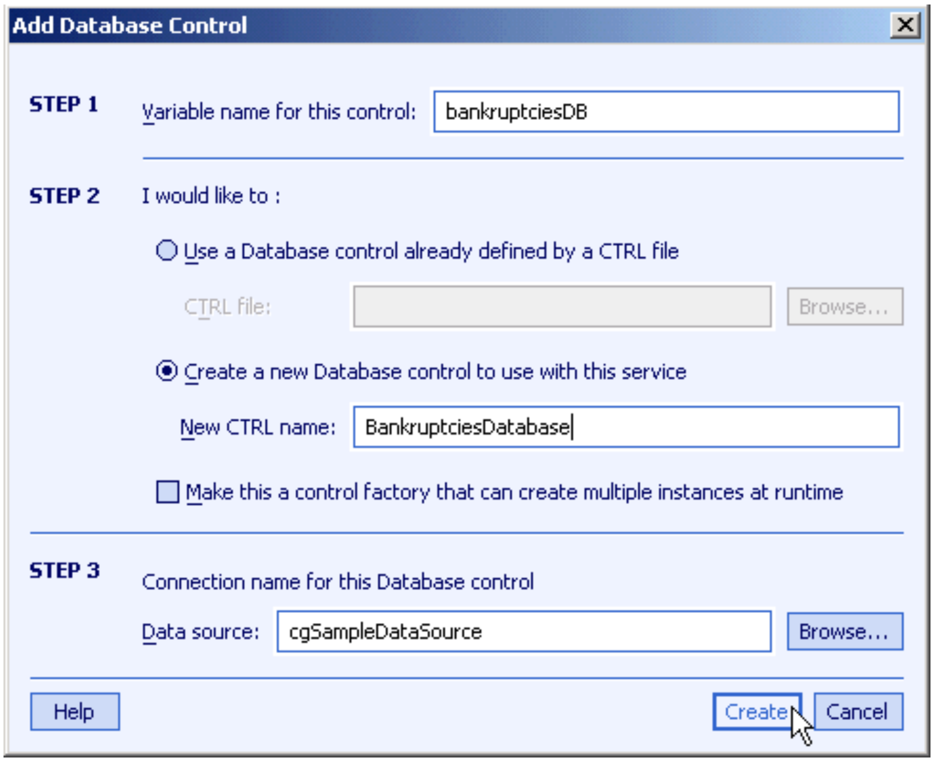
- [To create a database CTRL file]

- To edit the web service code to incorporate the CTRL file
- To test the web service using the debugger

To Create a Database CTRL file

In this task you will create a database CTRL file and then add a method to this CTRL file. The method you add, called checkForBankruptcies, will query a database using a SQL query.

1. If you are not in Design View, click the Design View tab.
2. From the Add Control drop−down list, select Add a Database Control. The Add Database Control dialog appears.
3. Enter values as shown in the following illustration:


Add Database Control

STEP 1 Variable name for this control: bankruptciesDB

STEP 2 I would like to :

○ Use a Database control already defined by a CTRL file

CTRL file:

◉ Create a new Database control to use with this service

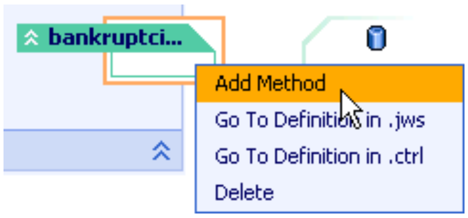
New CTRL name: BankruptciesDatabase

☐ Make this a control factory that can create multiple instances at runtime

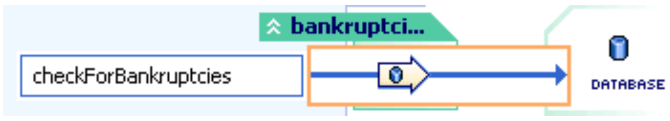
STEP 3 Connection name for this Database control

Data source: cgSampleDataSource

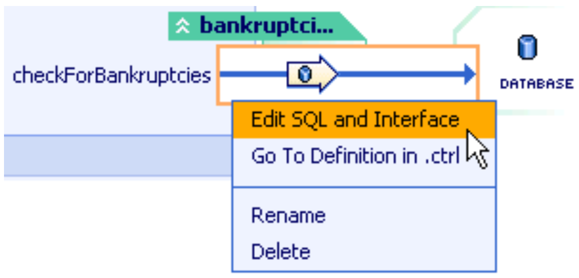
4. Click Create.
5. Right−click the newly created database control and select Add Method.



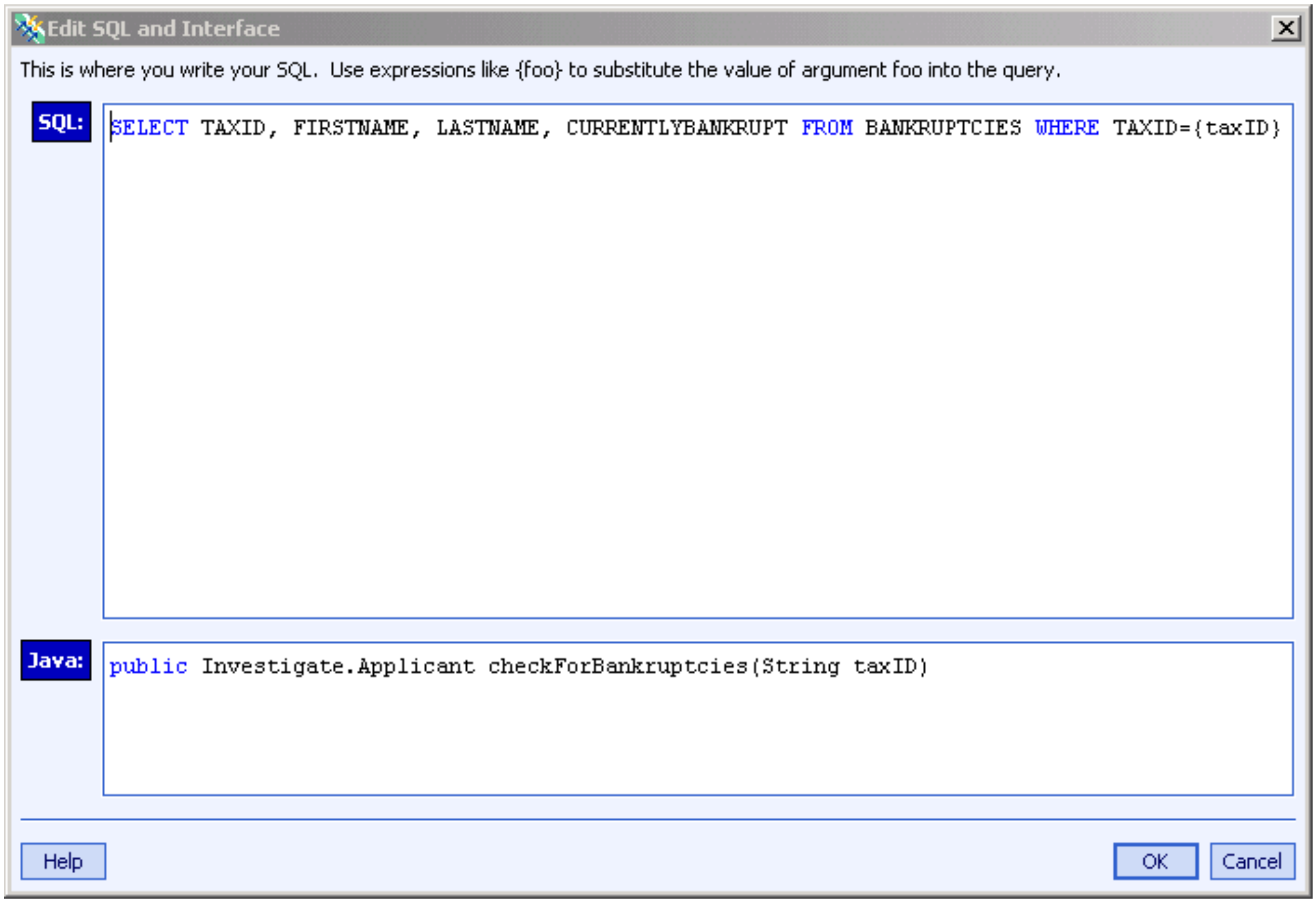
6. In the field that appears, type checkForBankruptcies and press Enter.



7. In Design View, right−click the arrow associated with the checkForBankruptcies method and select Edit SQL and Interface, as shown here. The EditSQL and Interface dialog appears.



8. Enter values as shown in the illustration below and click OK.
   Use the selectable text to cut and paste in the EditSQL and Interface dialog.
   For the SQL widow:
   SELECT TAXID, FIRSTNAME, LASTNAME, CURRENTLYBANKRUPT FROM BANKRUPTCIES WHERE TAXID={taxID}
   For the Java window:
   public Investigate.Applicant checkForBankruptcies(String taxID)



This is where you write your SQL. Use expressions like {foo} to substitute the value of argument foo into the query.

```
SELECT TAXID, FIRSTNAME, LASTNAME, CURRENTLYBANKRUPT FROM BANKRUPTCIES WHERE TAXID={taxID}
```

```
public Investigate.Applicant checkForBankruptcies(String taxID)
```

To Edit the Web Service Code to Incorporate the CTRL File

Next you must modify the web service's code to take advantage of the database control that has been added. You will edit the method requestCreditReportAsynch to invoke the control method checkForBankruptcies. Any information found in the database is stored in the member variable m_currentApplicant.

1. Click the Source View tab.
2. Edit the requestCreditReportAsynch method to look like the following:

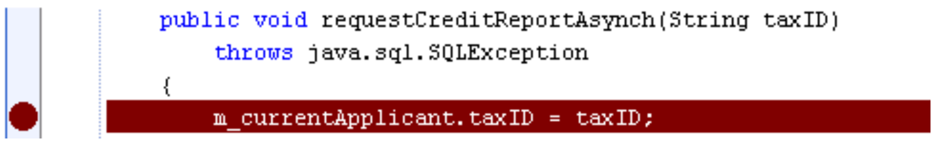
```
public void requestCreditReportAsynch(String taxID)
    throws java.sql.SQLException
{
    m_currentApplicant.taxID = taxID;
    m_currentApplicant = bankruptciesDB.checkForBankruptcies(taxID);
    callback.onCreditReportDone(m_currentApplicant, null);
}
```

To Test the Web Service Using the Debugger

Next you will test your web service using the debugger. The debugger allows you set breakpoints in your code and track how your code is running line−by−line. Setting a breakpoint in your code will cause the Java Virtual Machine to halt execution of your code immediately before the breakpoint, allowing you step through your code beginning at that point.

1. If you are not in Source View, double−click Investigate to view the source code for Investigate.jws.
2. Place the cursor on the first line of code executed within the method requestCreditReportAsynch.
3. Click the Toggle Breakpoint button on the toolbar, as shown here:

The breakpoint appears on the first line of code, as shown here:

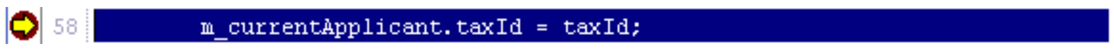


```
public void requestCreditReportAsynch(String taxID)
    throws java.sql.SQLException
{
    m_currentApplicant.taxID = taxID;
```

4. Press the Start and Debug button on the toolbar, shown here:

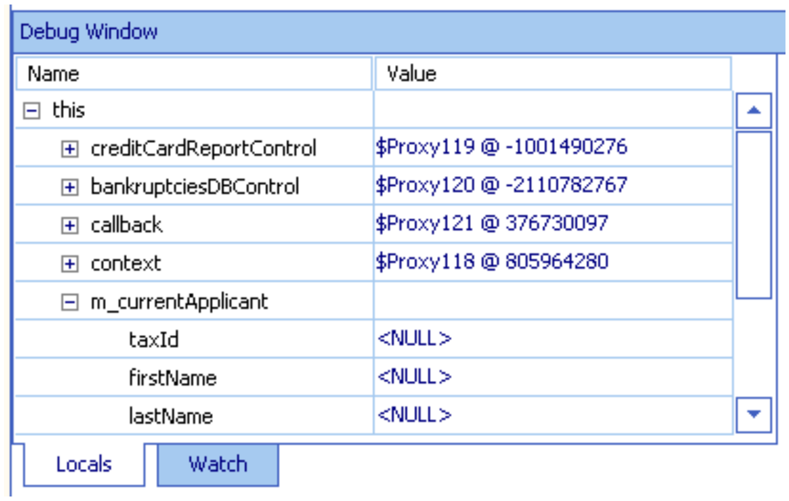
The Test View page appears.

5. In the taxID box, type the nine−digit number 222222222 and click requestCreditReportAsynch. Note: The database you just added to your web service contains data on 6 individuals. The taxIDs of these individuals are 123456789, 111111111, 222222222, 333333333, 444444444, and 555555555. Use these six taxIDs to test your web service throughout the tutorial.
6. The web service does not run through its routine of method calls and callbacks. To verify this, refresh the browser and note that the service has halted at the requestCreditReportAsynch.
7. Return to Workshop and note that the execution of code has halted at the breakpoint shown here:



```
        m_currentApplicant.taxId = taxId;
```

8. On the Locals tab of the Debug Window pane, expand the entries for this and m_currentApplicant, as shown here:

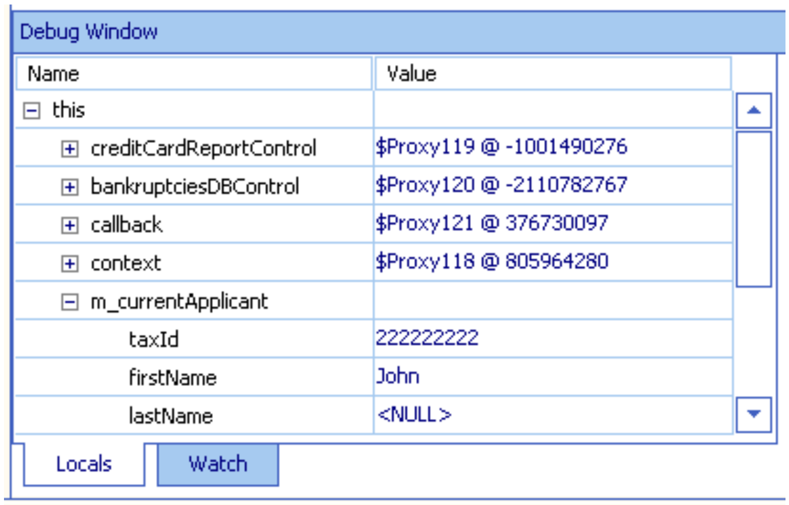


Name	Value
⊟ this	
⊞ creditCardReportControl	$Proxy119 @ -1001490276
⊞ bankruptciesDBControl	$Proxy120 @ -2110782767
⊞ callback	$Proxy121 @ 376730097
⊞ context	$Proxy118 @ 805964280
⊟ m_currentApplicant	
taxId	<NULL>
firstName	<NULL>
lastName	<NULL>

Note: The Debug Window pane gives you information about the current values of the variables within your web service, as well as current position in the call stack.

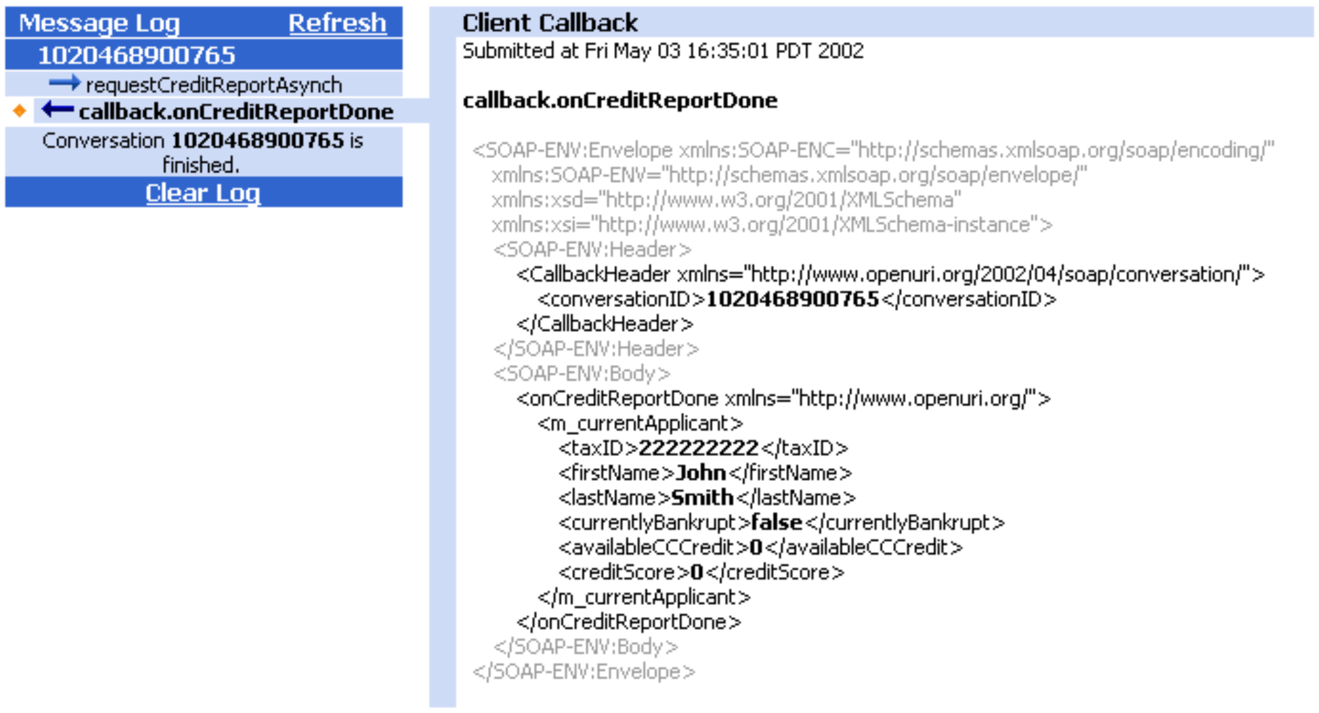
9. Click the Step Into button on the toolbar, shown here:

Each time you press the Step Into button, a new line of code within the requestCreditReportAsynch method is executed. The properties of the m_currentApplicant are filled with values as they are retrieved from the database, as shown here:



Name	Value
⊟ this	
⊞ creditCardReportControl	$Proxy119 @ -1001490276
⊞ bankruptciesDBControl	$Proxy120 @ -2110782767
⊞ callback	$Proxy121 @ 376730097
⊞ context	$Proxy118 @ 805964280
⊟ m_currentApplicant	
taxId	222222222
firstName	John
lastName	<NULL>

10. Continue clicking the Step Into button until the web service finishes executing.
11. Return to the browser window that displays the test page and refresh the browser.
12. Under Message Log, click callback.onCreditRepotDone. The response sent back to the client application appears, as shown here:


Message Log
Refresh
1020468900765
requestCreditReportAsynch
callback.onCreditReportDone
Conversation 1020468900765 is finished.
Clear Log

Client Callback
Submitted at Fri May 03 16:35:01 PDT 2002

callback.onCreditReportDone

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <CallbackHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
      <conversationID>1020468900765</conversationID>
    </CallbackHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <onCreditReportDone xmlns="http://www.openuri.org/">
      <m_currentApplicant>
        <taxID>222222222</taxID>
        <firstName>John</firstName>
        <lastName>Smith</lastName>
        <currentlyBankrupt>false</currentlyBankrupt>
        <availableCCCredit>0</availableCCCredit>
        <creditScore>0</creditScore>
      </m_currentApplicant>
    </onCreditReportDone>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

You have completed adding a database control and testing your web service with the debugger. In the next step of the tutorial you will add a service control to your web service.

Related Topics

Debugging Web Services

Database Control: Using a Database from Your Web Service

Click one of the following arrows to navigate through the tutorial.

⇦ ⇨

Step 4: Add a Service Control

Below we will add a service control to your web service. In other words, you will add a CTRL file which allows your web service to invoke another external web service. This other web service provides credit card data on a credit applicant.

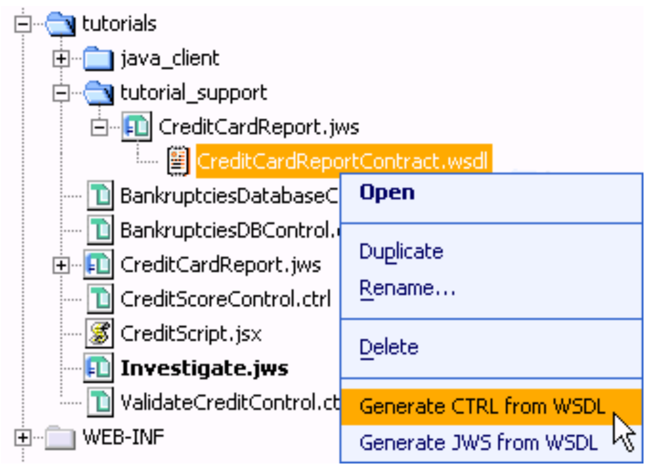
The tasks in this step are:

- To generate a CTRL file from a WSDL file
- To add the CTRL file to your web service
- To add code to invoke an external web service
- To add code to handle callbacks from the external web service
- To test the web service

To Generate a CTRL File from a WSDL File

By definition, each web service provides a WSDL file that explains how the web service is operated. In this task you will automatically generate a CTRL file based the WSDL file of another web service. The resulting CTRL file is incorporated into your web service to form the interface between your web service and the external web service.

1. In the Project tree, expand the tutorials folder, then expand the tutorial_support folder, finally expand CreditCardReport.jws as shown below.
2. Right−click the CreditCardReportContract.wsdl file and select Generate CTRL from WSDL, as shown here:

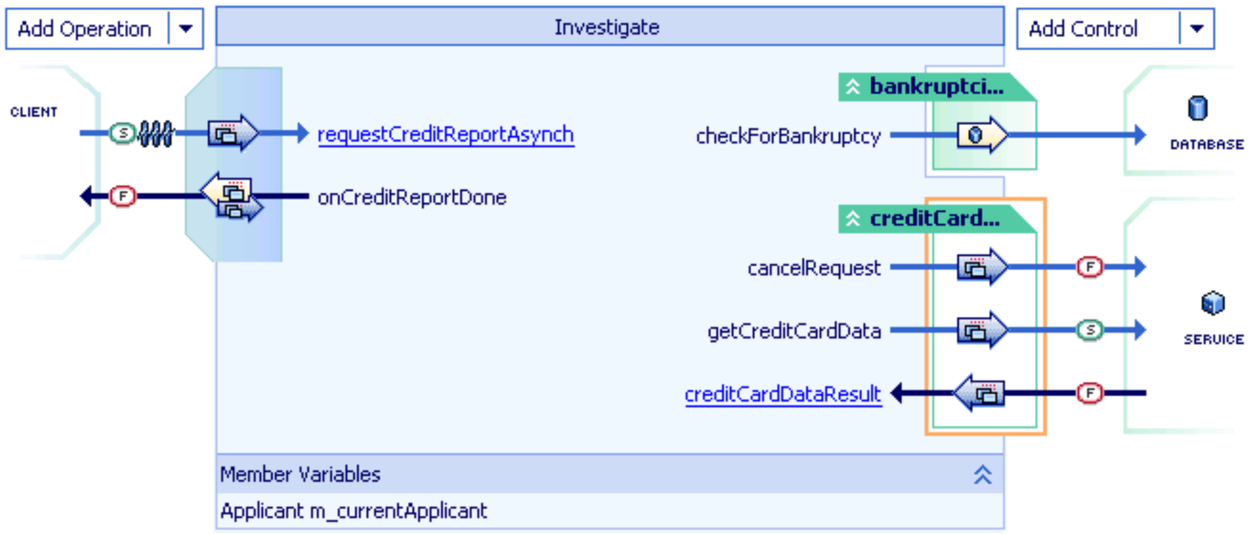


Note that a new CTRL file is generated and placed underneath CreditCardReportContract.wsdl.

To Add the CTRL File to Your Web Service

In this task you will add the new CTRL file to your web service. With the CTRL file added you will be able to invoke the external web service that provides credit card data.

1. If you are not in Design View, click the Design View tab.
2. Drag and drop the newly generated CreditCardReportContract.ctrl from the Project Tree into Design View. Your screen should now look like the one shown here:



A new service control, containing two methods and a callback, is added to your web service.

To Add Code to Invoke the External Web Service

In this task you will add code that invokes the external web service.

1. Click the Source View tab.

2. Edit the requestCreditReportAsynch method to look like the code shown here. Make sure that you delete the line of code shown crossed out below.

```
public void requestCreditReportAsynch(String taxID)
    throws java.sql.SQLException
{
    m_currentApplicant.taxID = taxID;
    m_currentApplicant = bankruptciesDB.checkForBankruptcies(taxID);
    creditCardReportControl.getCreditCardData(taxID);
    callback.onCreditReportDone(m_currentApplicant, null);
}
```

To Add Code to Handle Callbacks from the External Web Service

In this task you will add code to handle the credit card information sent back from the external web service. This code does two things: first, it stores the relevant credit card data in the member variable m_currentApplicant. Second, it sends the applicant profile back to the client using the callback onCreditReportDone.

1. If you are not in Design View, click the Design View tab.
2. Click creditCardDataResult.
3. Edit the callback handler to look like the following:

```
private void creditCardReportControl_creditCardDataResult(CreditCard[] cards) {    for(int i = 0; i <
cards.length; i++)  {        m_currentApplicant.availableCCCredit += cards[i].availableCredit;    }
    callback.onCreditReportDone(m_currentApplicant, null); }
```

4. Press Ctrl+S to save the file.

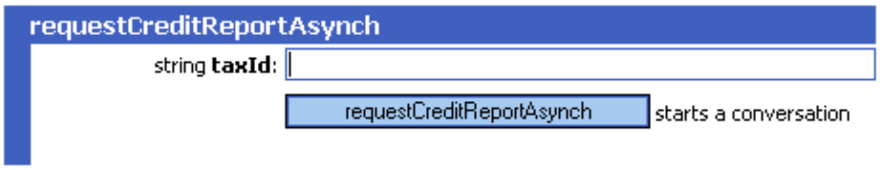
To Test the Web Service

You are now ready to test the new features of your web service.

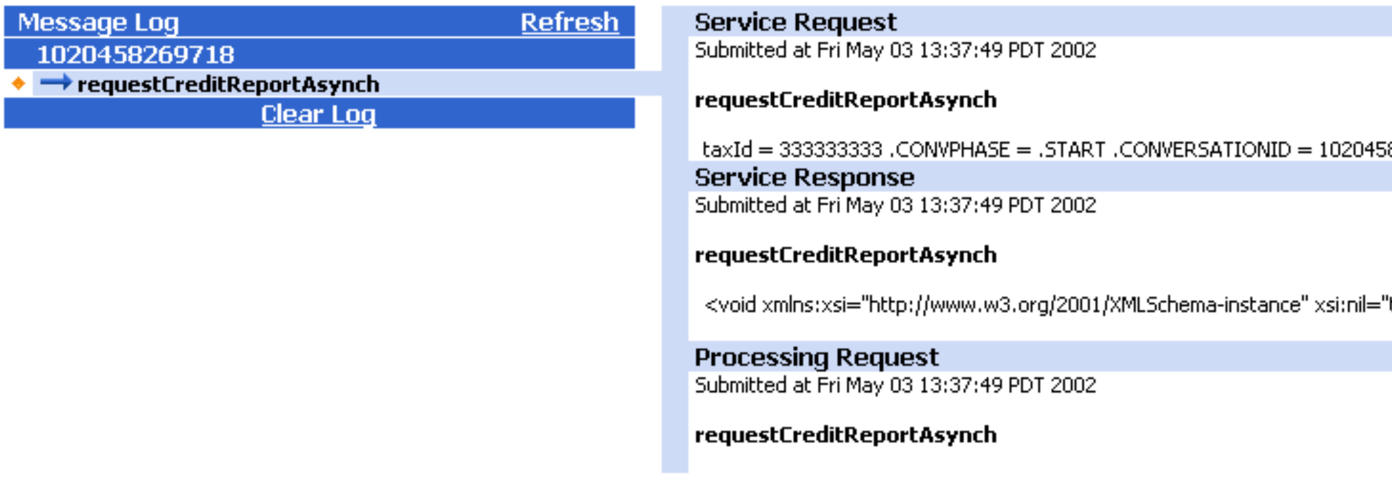
1. Click the Start button, shown here:

WebLogic Workshop tests and compiles the web service. A browser is launched displaying Test View.

2. When Test View launches, type one of the following nine−digit numbers in the taxID box for requestCreditReportAsynch, as shown below:
   123456789, 111111111, 222222222, 333333333, 444444444, 555555555



3. Click requestCreditReportAsynch. The Test View page refreshes to display a summary of the request parameters sent by the client and your service's response, as shown here:



Message Log Refresh
1020458269718
◆ → requestCreditReportAsynch
Clear Log

Service Request
Submitted at Fri May 03 13:37:49 PDT 2002

requestCreditReportAsynch

taxId = 333333333 ,CONVPHASE = ,START ,CONVERSATIONID = 102045

Service Response
Submitted at Fri May 03 13:37:49 PDT 2002

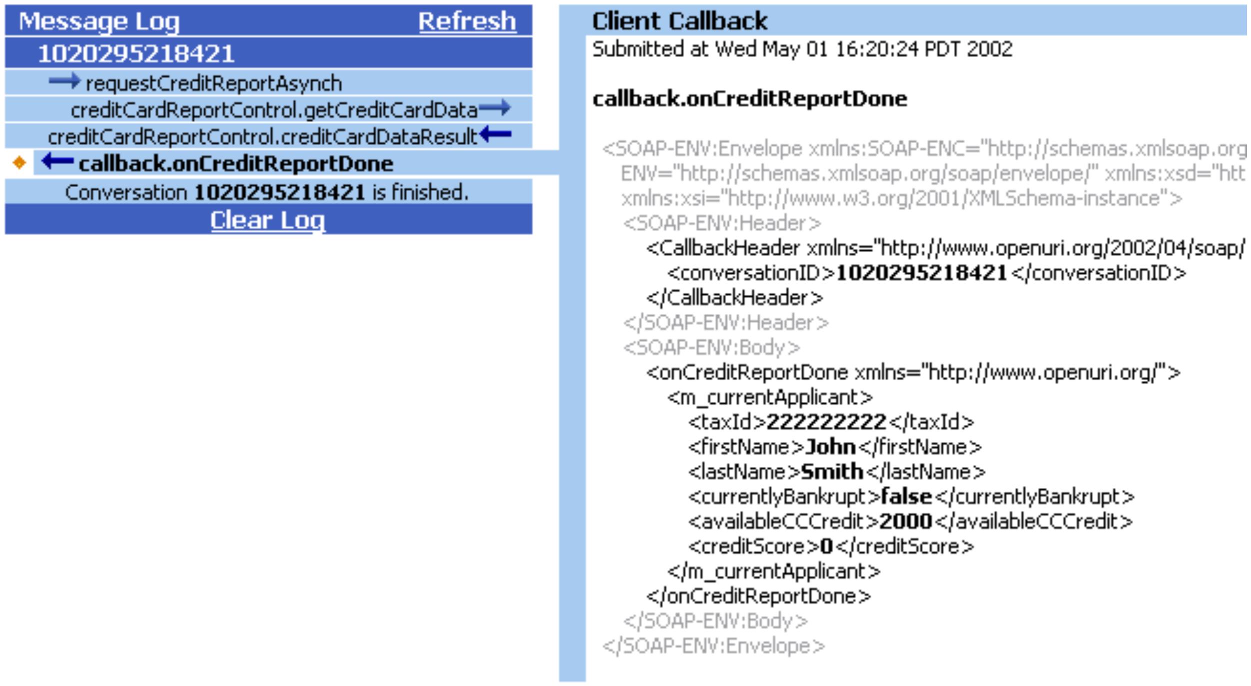
requestCreditReportAsynch

```
<void xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:nil="
```

Processing Request
Submitted at Fri May 03 13:37:49 PDT 2002

requestCreditReportAsynch

4. Refresh the browser. New entries appear in the Message Log, as shown here:



Message Log Refresh
1020295218421
→ requestCreditReportAsynch
creditCardReportControl.getCreditCardData →
creditCardReportControl.creditCardDataResult ←
◆ ← callback.onCreditReportDone
Conversation **1020295218421** is finished.
Clear Log

Client Callback
Submitted at Wed May 01 16:20:24 PDT 2002

callback.onCreditReportDone

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org
  ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="htt
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <CallbackHeader xmlns="http://www.openuri.org/2002/04/soap/
      <conversationID>1020295218421</conversationID>
    </CallbackHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <onCreditReportDone xmlns="http://www.openuri.org/">
      <m_currentApplicant>
        <taxId>222222222</taxId>
        <firstName>John</firstName>
        <lastName>Smith</lastName>
        <currentlyBankrupt>false</currentlyBankrupt>
        <availableCCCredit>2000</availableCCCredit>
        <creditScore>0</creditScore>
      </m_currentApplicant>
    </onCreditReportDone>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

5. Click callback.onCreditReportDone. The callback sent from the web service to the client appears.

Related Topics

Service Control: Using Another Web Service

Click one of the following arrows to navigate through the tutorial.

Step 5: Add a JMS and EJB Control

In this step, you will take the data you have gathered and request help from an in−house application to generate a credit score. You will then pass the credit score to an Enterprise Java Bean (EJB) that will measure the applicant's credit risk.

To generate the credit score, you will use an in−house credit scoring application designed to do just this sort of work. To make this legacy application available to other components, it has been exposed to other parts of the company through a messaging system. That's how your web service will get to it. You will send an XML message to the application with applicant data, and receive an XML message with a credit score.

Messaging systems are often used to create bridges between otherwise incompatible software components. Messaging is also useful when asynchronous communication is needed because the requesting component doesn't have to wait for a response in order to continue working. Here, asynchrony is useful because you have no way of knowing how long the credit scoring application will take to respond to your request. (For more information about messaging, see Overview: Messaging Systems and JMS.)

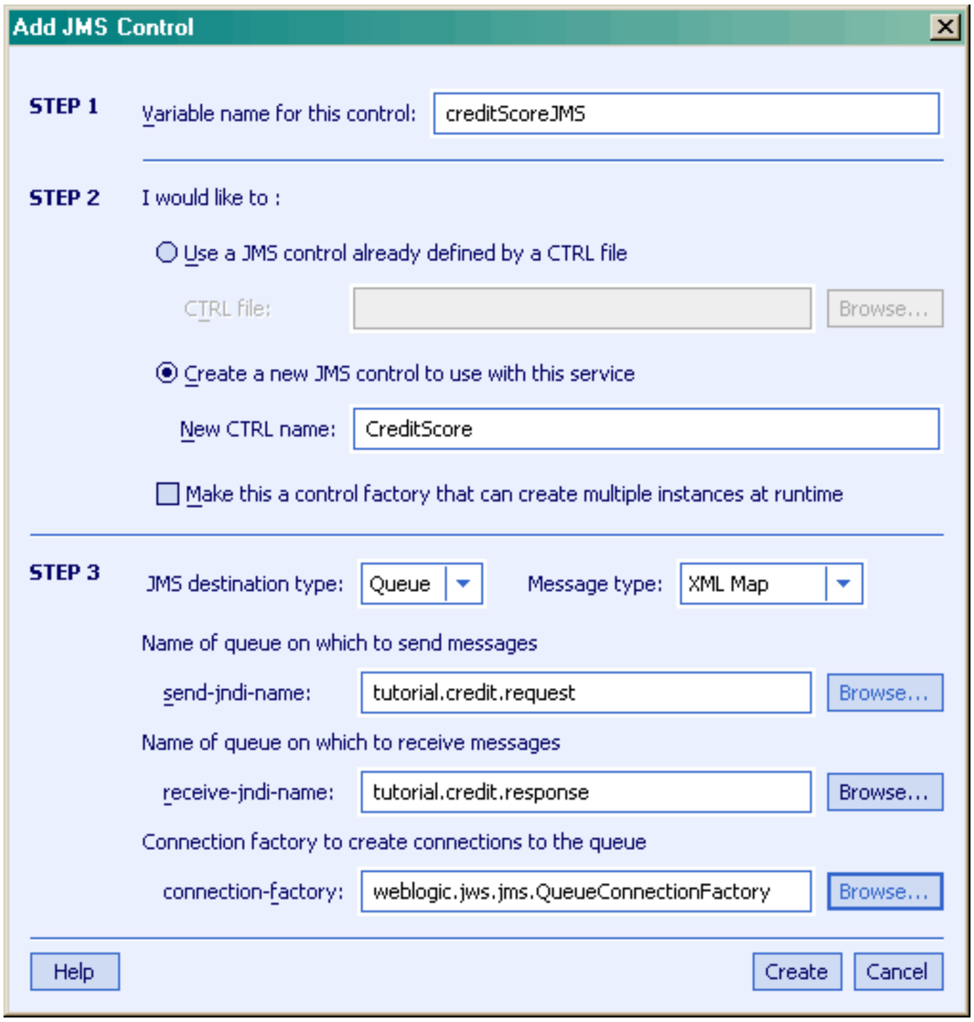
The Java Message Service (JMS), provided with WebLogic Server, supports several ways for applications to use messages. In point−to−point messaging, which you will use here, the message's sender and receiver are both clients of the messaging system (or "JMS provider"). The first client begins the exchange by sending a message to a queue. A queue is a kind of message waypoint to which other clients are listening. When the message arrives at the queue, the receiving client picks it up and reads it. You will use the JMS control to send a message (containing the data you have collected) to a queue that the credit scoring application is listening to. The application will send its response message (containing a credit score) to a queue that your service is listening to.

The tasks in this step are:

- To add a JMS control for requesting a credit score
- To add XML maps for JMS control
- To edit the callback handlers
- To add an EJB control for requesting a credit approval rating
- To add code that requests credit validation
- To launch the Test View

To Add a JMS Control for Requesting a Credit Score

1. If you are not in Design View, click the Design View tab.
2. From the Add Control drop−down list, select Add JMS Control. The Add JMS Control dialog appears.
3. Enter values as shown in the following illustration:


Add JMS Control

STEP 1 Variable name for this control: creditScoreJMS

STEP 2 I would like to :

○ Use a JMS control already defined by a CTRL file

CTRL file: Browse...

● Create a new JMS control to use with this service

New CTRL name: CreditScore

☐ Make this a control factory that can create multiple instances at runtime

STEP 3 JMS destination type: Queue Message type: XML Map

Name of queue on which to send messages

send-jndi-name: tutorial.credit.request Browse...

Name of queue on which to receive messages

receive-jndi-name: tutorial.credit.response Browse...

Connection factory to create connections to the queue

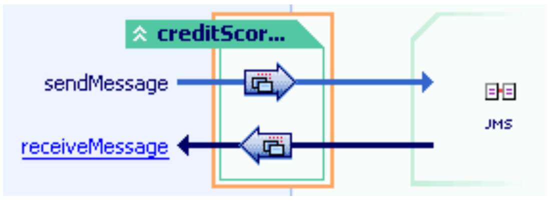
connection-factory: weblogic.jws.jms.QueueConnectionFactory Browse...

Help Create Cancel

Note that the message type is XML Map. XML maps are a powerful tool for turning your Java types into XML. These will be covered in more detail later in this tutorial.

4. Click Create.

Your service design now includes the creditScoreJMS control, as shown in the following illustration.



To Add XML Maps to the JMS Control

1. Double−click the arrow corresponding to the sendMessage method. The Edit Maps and Interface dialog appears.
2. Confirm that the Message XML tab is selected.
3. Edit the contents of the Java field so that it contains the following:

```
public void sendMessage(int availableCredit, boolean bankruptcy)
```

Using this method, you will send data about the applicant to the credit scoring application.

4. Leaving the dialog open, edit the contents of the top box so it contains the following:

```
<score_request>    <credit_remaining>{availableCredit}</credit_remaining>
   <is_bankrupt>{bankruptcy}</is_bankrupt> </score_request>
```

Note: This code represents an XML map. The <score_request>, <credit_remaining>, and <is_bankrupt> elements define the XML expected by the credit scoring application. Notice that the text in curly braces matches parameters of the declaration. This tells WebLogic Server how to insert the values passed to the sendMessage method into the XML. The XML is then sent to the application.

5. Click OK.
6. Double−click the arrow corresponding to the receiveMessage callback handler. The Edit Maps and Interface dialog appears.
7. On the Message XML tab, edit the XML map code as follows:

```
<score_response>    <calculated_score>{score}</calculated_score> </score_response>
```

8. Edit the Java code as follows:

```
public void receiveMessage(int score)
```

This adds an XML map to the callback handler. The map matches the format of the expected XML response. Placing the score parameter in the curly braces tells WebLogic Server to pass the returned data to the callback handler's score parameter.

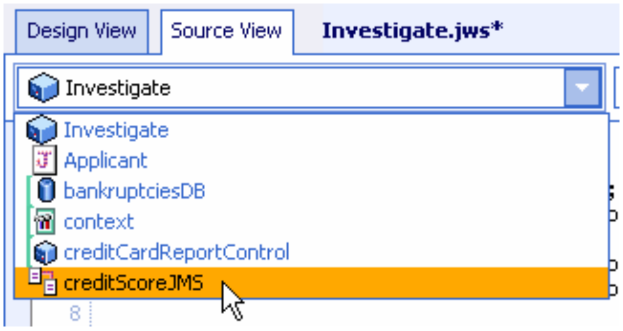
9. Click OK.

To Edit the Callback Handlers

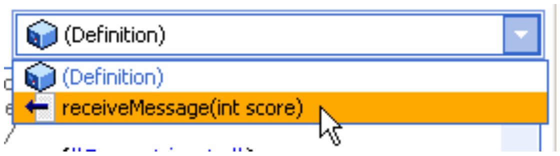
1. In Design View, click the name of the creditCardDataResult callback handler to view its source code.
2. Edit the creditCardDataResult callback handler code so that it appears as follows:

```
private void creditCardReportControl_creditCardDataResult(CreditCard[] cards) {       for(int i = 0; i <
cards.length; i++) {         m_currentApplicant.availableCCCredit += cards[i].availableCredit;    }        /*     *
Use the JMS control to send the available credit and bankruptcy information to the credit     * scoring
application.     */    creditScoreJMS.sendMessage(m_currentApplicant.availableCCCredit,
       m_currentApplicant.currentlyBankrupt); }
```

3. In Source View, immediately beneath the Source View tab, from the class drop−down list, select creditScoreJMS, as shown here:



4. To the right of the class drop−down list, from the member drop−down list, select receiveMessage, as shown here.



The source code for the JMS control's callback handler appears.

5. Edit the receiveMessage callback handler code so that it appears as follows:

private void creditScoreJMS_receiveMessage(int score) {   /* Store the score returned with other data about the applicant. */   m_currentApplicant.creditScore = score; }

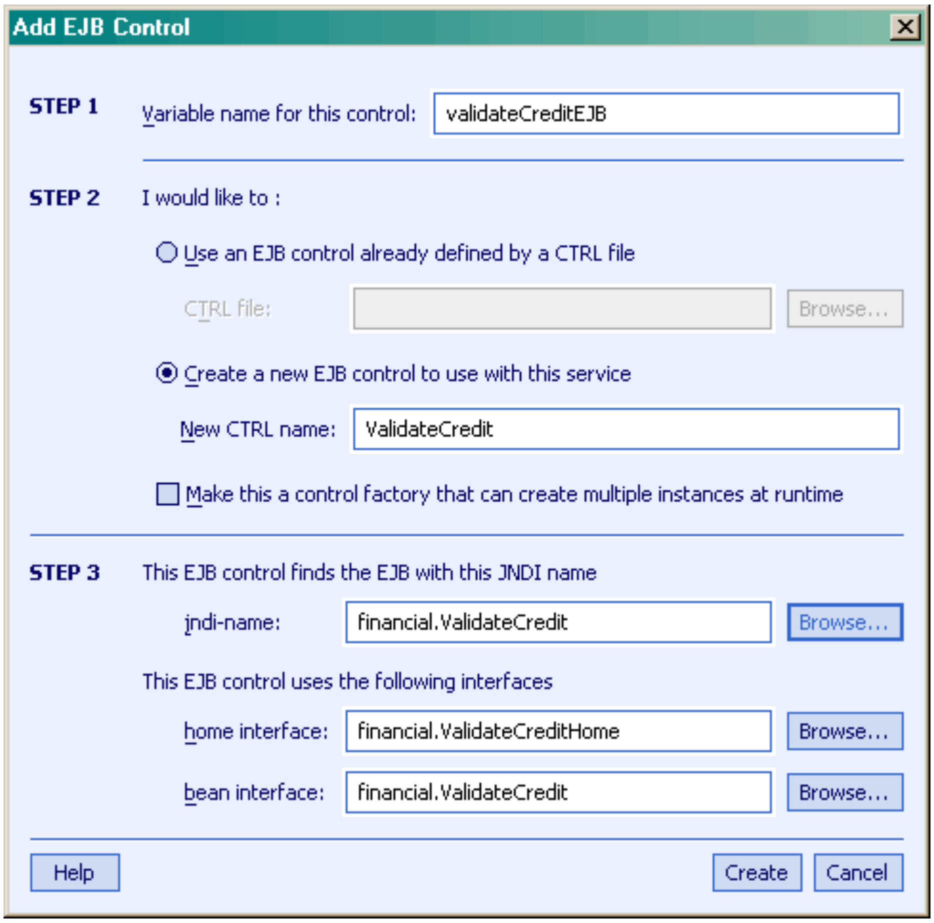
To Add an EJB Control for Requesting a Credit Approval Rating

Now that you have a way to retrieve a credit score, you will use the score to determine whether the applicant deserves credit approval. For this you will use an existing stateless session Enterprise Java Bean (EJB). EJBs are Java software components of enterprise applications. The Java 2 Enterprise Edition (J2EE) Specification defines the types and capabilities of EJBs as well as the environment (or container) in which EJBs are deployed and execute. From a software developer’s point of view, there are two aspects to EJBs: development and deployment of EJBs; and use of EJBs from client software. WebLogic Workshop provides the EJB control as a simplified way to act as a client of an existing EJB from within a web service.

The ValidateCredit EJB you will be accessing is designed to take the credit score and return response about the applicant's credit worthiness. To access the ValidateCredit bean, you will add an EJB control.

Note: In order for you add an EJB control to a web service project, the compiled home and remote interfaces for the EJB must be in the project also. For the sake of the tutorial, the JAR file containing the ValidateCredit bean has already been copied to the WEB−INF\lib folder of the samples project. Having the bean's JAR file in the WEB−INF\lib folder ensures that WebLogic Workshop can find these interfaces. (For more information, see Creating a New EJB Control.)

1. If you are not in Design View, click the Design View tab.
2. From the Add Control drop−down list, select Add EJB Control. The Add EJB Control dialog appears.
3. Enter values as shown in the following illustration. Browse for the jndi−name value required in Step 3 of the dialog. When you select it from the list and click Select, the home interface and bean interface values will be added automatically.



Add EJB Control

STEP 1 Variable name for this control: validateCreditEJB

STEP 2 I would like to :

○ Use an EJB control already defined by a CTRL file

CTRL file: Browse...

● Create a new EJB control to use with this service

New CTRL name: ValidateCredit

☐ Make this a control factory that can create multiple instances at runtime

STEP 3 This EJB control finds the EJB with this JNDI name

jndi-name: financial.ValidateCredit Browse...

This EJB control uses the following interfaces

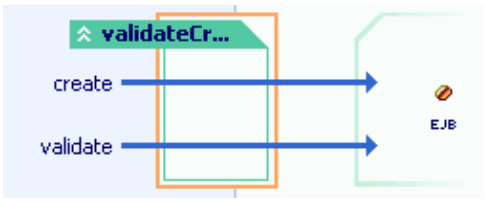
home interface: financial.ValidateCreditHome Browse...

bean interface: financial.ValidateCredit Browse...

Help Create Cancel

4. Click Create.

You can see that WebLogic Workshop has added a validateCreditEJB control to your design, as shown below. The control shows that the EJB exposes two methods: create and validate. The validate method is the one you will use (all EJBs expose a create method that is used by WebLogic Server; you will not need to call this).



Now you need to connect the score received through the creditScore JMS control with the validateCredit EJB control. You will do this by updating the JMS control's callback handler. The code you add will send the credit score to the EJB for validation.

To Add Code that Requests Credit Validation

1. If you are not in Design View, click the Design View tab.
2. Click the name of the receiveMessage callback handler. Its source code appears in Source View.
3. Edit the receiveMessage code so that it appears as follows:

private void creditScoreJMS_receiveMessage(int score)   throws java.rmi.RemoteException {
   m_currentApplicant.creditScore = score;   /*   * Pass the credit score to the EJB's validate method. Store the value returned   * with other applicant data.   */   m_currentApplicant.approvalLevel = validateCreditEJB.validate(m_currentApplicant.creditScore);   /*   * Send the Applicant object, now

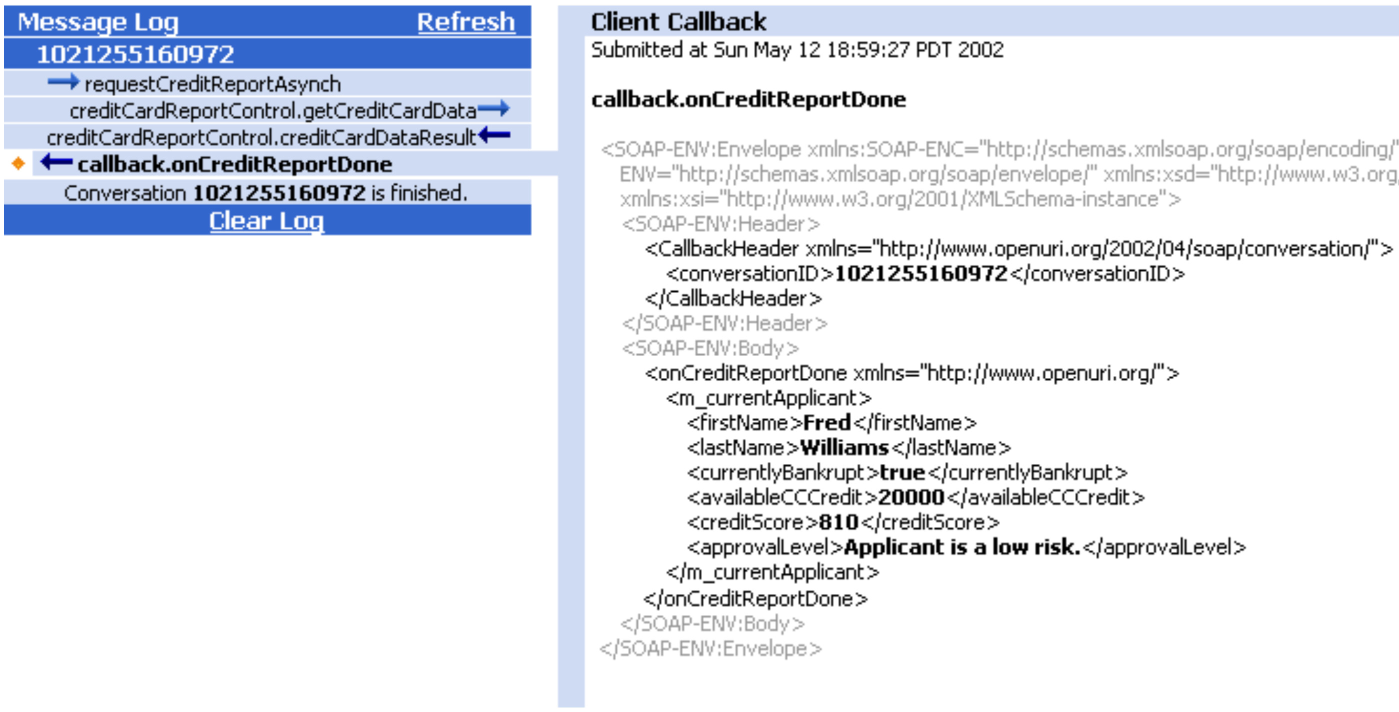
complete with all the data the client requested,   * to the client using the callback.   */
    callback.onCreditReportDone(m_currentApplicant, "Credit score received."); }

You've added a lot of functionality in these last steps. Now it's time to try it out and see if it all works.

To Launch Test View

1. Click the Start button. As before, this launches your browser to display the Test View.

2. Click the Refresh button to reload Test View until callback.onCreditReportDone appears in the Message Log. Your screen should resemble the following:



```
Message Log                                   Refresh
 1021255160972
   → requestCreditReportAsynch
     creditCardReportControl.getCreditCardData →
   creditCardReportControl.creditCardDataResult ←
 ◆ ← callback.onCreditReportDone
    Conversation 1021255160972 is finished.
                  Clear Log
```

```
Client Callback
Submitted at Sun May 12 18:59:27 PDT 2002

callback.onCreditReportDone

 <SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
   ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <SOAP-ENV:Header>
     <CallbackHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
       <conversationID>1021255160972</conversationID>
     </CallbackHeader>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
     <onCreditReportDone xmlns="http://www.openuri.org/">
       <m_currentApplicant>
         <firstName>Fred</firstName>
         <lastName>Williams</lastName>
         <currentlyBankrupt>true</currentlyBankrupt>
         <availableCCCredit>20000</availableCCCredit>
         <creditScore>810</creditScore>
         <approvalLevel>Applicant is a low risk.</approvalLevel>
       </m_currentApplicant>
     </onCreditReportDone>
   </SOAP-ENV:Body>
 </SOAP-ENV:Envelope>
```

Notice that the <creditScore> and <approvalLevel> elements of the response now have values in them. This means that in just the few steps of this topic, you have connected the Investigate web service with a JMS messaging system and an Enterprise Java Bean.

In the next step you will enhance the service's design to return its result in a more generic format, making it more broadly useful.

Related Topics

[Using a Control]()

[JMS Control: Using Java Message Service Queues and Topics from Your Web Service]()

[EJB Control: Using Enterprise Java Beans from a Web Service]()

Click one of the following arrows to navigate through the tutorial.

Step 6: Add Script for Mapping

The Investigate web service returns an Applicant object to its client. That's fine for clients that have their own version of the Applicant class. When the data returns to them as a SOAP message, their software will translate the message into an Applicant object for their code to use. But what about other clients? In this step you will add an XML map to Investigate's callback to translate the Applicant object into plain XML that can be understood by another client. This client does not have the Applicant class, but does expect to receive XML that looks a particular way.You can shape the XML message the client receives with an XML map.

As you saw in the preceding step with the JMS control, XML maps look like XML documents. WebLogic Server uses XML maps you create to translate Java to XML, or XML to Java. Maps do this by showing where Java values should be inserted as XML element and attribute values (for incoming XML messages, it works the other way around). The direction of substitution depends on whether the XML message is sent or received by your web service.

You will apply an XML map to translate the data in the Applicant object into a particular XML shape. There's a twist in this case, though: The XML shape expected by the client includes an additional element. This element provides a place for the national percentile ranking of the applicant's credit score. But it's a little tedious to go back and retrofit your Java code for this client only. Instead, you can calculate the percentile through the map.

To do this, you will write a function in ECMAScript (also known as JavaScript), then refer to the function from the XML map. It will send the percentile back wrapped in XML that can be inserted into the XML map. This will be easier than you might expect, however. WebLogic Workshop includes support for an extended version of ECMAScript. These extensions turn XML nodes and node lists into native script objects. These objects also expose functions specifically designed for use with XML.

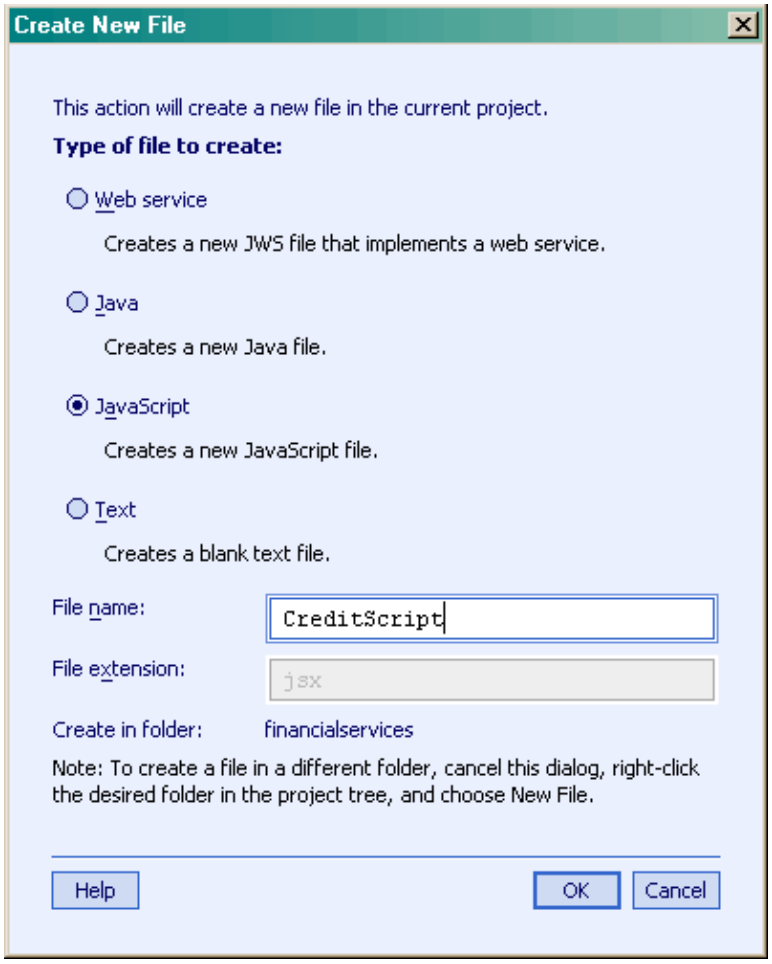
The tasks in this step are:

- To add a script file for use with mapping
- To write script that maps Java types to XML
- To launch Test View

To Add a Script File for Use with Mapping

1. If you are not in Design View, click the Design View tab.
2. Right−click the financialservices folder, then select New File. The Create New File dialog opens.

3. Enter values as shown in the following illustration:


Create New File

This action will create a new file in the current project.

Type of file to create:

( ) Web service

Creates a new JWS file that implements a web service.

( ) Java

Creates a new Java file.

(●) JavaScript

Creates a new JavaScript file.

( ) Text

Creates a blank text file.

File name: CreditScript

File extension: jsx

Create in folder: financialservices

Note: To create a file in a different folder, cancel this dialog, right-click the desired folder in the project tree, and choose New File.

4. Click OK. This adds a JSX file called CreditScript.jsx. As you will see, files with a JSX extension have special meaning. WebLogic Workshop displays the new file in Source View.

Note: The new file includes comments and example code to get you started. As you can see, JSX files support mapping through functions that take a particular form. A function either translates from XML to Java, or Java to XML.

Because you are mapping results to the client, you only need a function that translates Java to XML. You might find that it simplifies your work to delete the other function declaration, along with documentation comments.

5. Edit the file so that only the following code remains:

// import mypackage.MyOuterClass.MyClass; /* Rename the "toXML" function given you by WebLogic Workshop, but keep the toXML suffix. This is required in order for WebLogic Server to know that this function is for translating from Java to XML. A "fromXML" function would be for translation in the other direction. */ function calcScoreInfoToXML(obj) { }

To Write Script that Maps Java Types to XML

Next, you will write script to help generate the XML shape expected by the client. The following XML gives an example of the intended shape. Your script will generate the portion in the <score_info> element; the bold portion will come from an XML map:

```
<applicant id="111111111">
```

```
   <name_last>Walton</name_last>
   <name_first>Bill</name_first>
   <bankrupt>true</bankrupt>
   <balance_remaining>1000</balance_remaining>
   <risk_estimate>Ha! Who are they trying to kid?</risk_estimate>
   <score_info>
      <credit_score>460</credit_score>
      <us_percentile>19.0</us_percentile>
   </score_info>
</applicant>
<notes>Credit score received.</notes>
```

1. At the top of the file, replace the example import statement with the following:

import financialservices.Investigate.Applicant;

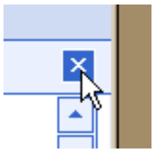
This imports the Applicant class so that you can access it in your script.

Note: The import directive here is not from Java; it is a part of the extended ECMAScript.

2. Edit the function's code so that it appears as follows:

import financialservices.Investigate.Applicant; /* The applicantJava parameter represents the Applicant object * passed from the onCreditReportDone callback through its XML map. */ function calcScoreInfoToXML(applicantJava) {   print(typeof applicantJava.creditScore);   /* Confirm that the applicantJava object passed into the function isn't null.   * If it is, trying to assign values from it will cause a script error.   * If it's null, your script will simply return two empty nodes to add to the   * message created by the XML map.   *   * In the scoreInfoNode variable assignment below, notice too that the two   * new nodes are enclosed in empty elements. These are known as "anonymous"   * elements. They are an ECMAScript extension that enables you to create lists   * of elements such as the two−member list below. Without them, the two elements   * alone would be malformed XML because it lacks a root element.   *   * Simply initializing the variable with a value that begins with < automatically   * makes the scoreInfoNode variable an XML object.   */   if(applicantJava == null) {   var scoreInfoNode = <>   <credit_score/>   <us_percentile/>   </>;   }   /* If the applicantJava value isn't null, use {} to insert a value from it   * into the <credit_score> element. Also, use a function defined in the script   * to calculate the credit score's national percentile ranking.   */   else {   var scoreInfoNode = <>   <credit_score>{applicantJava.creditScore}</credit_score>   <us_percentile>{calcPercentile(applicantJava.creditScore)}</us_percentile>   </>;   }   /* Return the completed XML. */   return scoreInfoNode; } /* A function to figure out the applicant's credit score percentile. */ function calcPercentile (score) {   var percentile = 0;   var scoreLowEnd = 300;   var number = 3162.5;   percentile = Math.ceil((Math.pow(score, 2) − Math.pow(scoreLowEnd, 2))/(2 * number));   return percentile; }

3. Click the close button in the upper right corner of Workshop to return to Design View for the Investigate web service, as shown here:



Note: If you don't see the Investigate service after closing the CreditScript.jsx file, double−click Investigate.jws in the Project Tree.

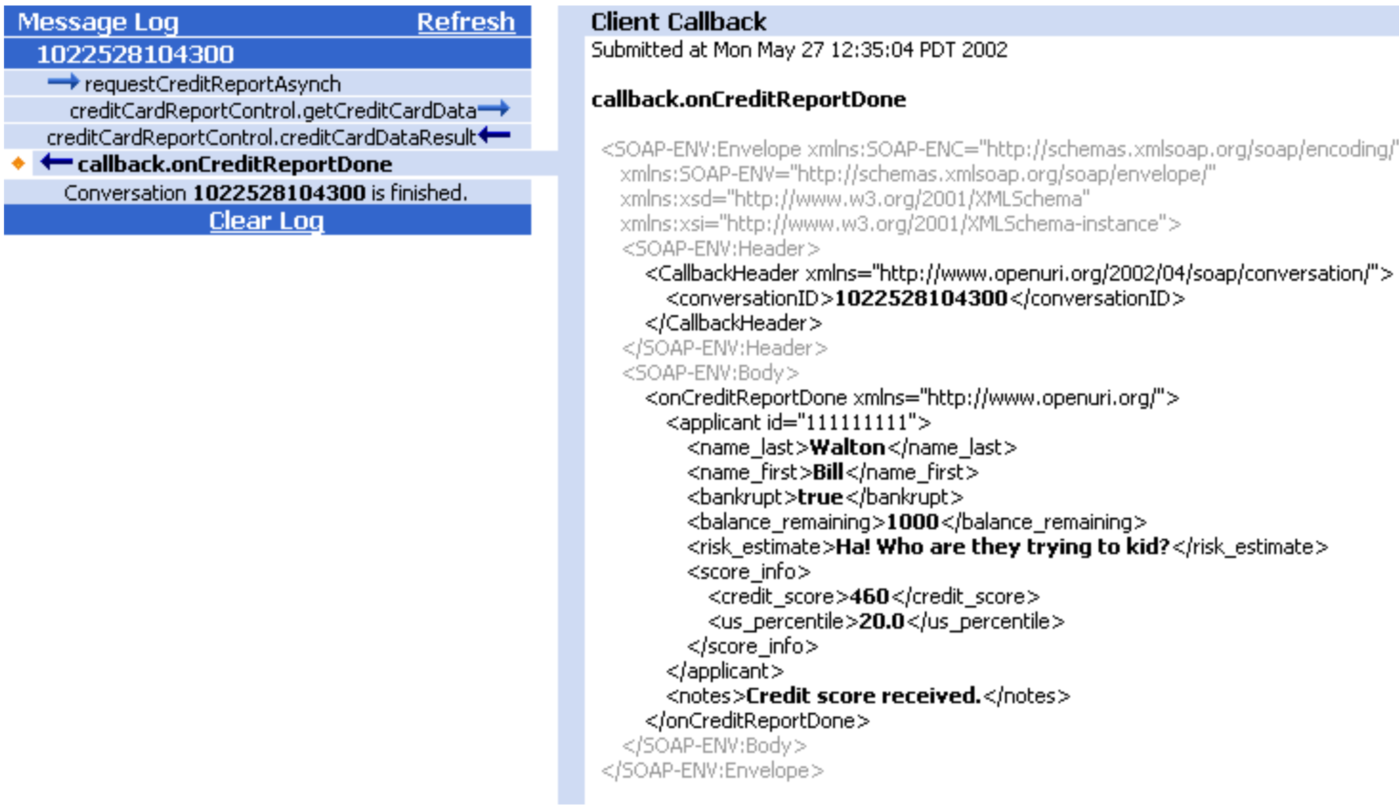
4. Double−click the arrow corresponding to the onCreditReportDone callback. The Edit Maps and Interface dialog appears.
5. On the Parameter XML tab, edit the top pane so that the XML map appears as follows:

```
<onCreditReportDone xmlns="http://www.openuri.org/">    <applicant id="{m_currentApplicant.taxID}">
       <name_last>{m_currentApplicant.lastName}</name_last>
       <name_first>{m_currentApplicant.firstName}</name_first>
       <bankrupt>{m_currentApplicant.currentlyBankrupt}</bankrupt>
       <balance_remaining>{m_currentApplicant.availableCCCredit}</balance_remaining>
       <risk_estimate>{m_currentApplicant.approvalLevel}</risk_estimate>        <score_info>
           {financialservices.CreditScript.calcScoreInfo(m_currentApplicant)}        </score_info>    </applicant>
   <notes>{responseMessage}</notes> </onCreditReportDone>
```

Throughout this XML map, the text between {} tells WebLogic Server to insert values from the callback's parameter, m_currentApplicant. Take a look also at the map's <score_info> element. The code in {} there tells WebLogic Server to use the script you have written to translate the callback's m_currentApplicant parameter.

To Launch Test View

1. Click the Start button. As before, this launches your browser to display the Test View.
2. Enter one of the tax ID numbers in the taxID box (for example, enter 111111111), then click requestCreditReportAsynch.
3. Click the Refresh button to reload Test View until callback.onCreditReportDone appears in the Message Log.
4. Click the callback.onCreditReportDone message. Your screen should resemble the following:


Message Log
Refresh
1022528104300
requestCreditReportAsynch
creditCardReportControl.getCreditCardData
creditCardReportControl.creditCardDataResult
callback.onCreditReportDone
Conversation 1022528104300 is finished.
Clear Log
Client Callback

Submitted at Mon May 27 12:35:04 PDT 2002

callback.onCreditReportDone

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <CallbackHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
      <conversationID>1022528104300</conversationID>
    </CallbackHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <onCreditReportDone xmlns="http://www.openuri.org/">
      <applicant id="111111111">
        <name_last>Walton</name_last>
        <name_first>Bill</name_first>
        <bankrupt>true</bankrupt>
        <balance_remaining>1000</balance_remaining>
        <risk_estimate>Ha! Who are they trying to kid?</risk_estimate>
        <score_info>
          <credit_score>460</credit_score>
          <us_percentile>20.0</us_percentile>
        </score_info>
      </applicant>
      <notes>Credit score received.</notes>
    </onCreditReportDone>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Compare this image with the onCreditReportDone results in the preceding step of this tutorial. You will see that this one is quite different from that one due to your XML map. Through the map, you have shaped the outgoing XML message so that a particular client can use it.

In the next step you will enhance the service's design to return its result in a more generic format, making it more broadly useful.

Related Topics

Why Use XML Maps?

Getting Started with XML Maps

Getting Started with Script for Mapping

Click one of the following arrows to navigate through the tutorial.

Step 7: Add Support for Cancellation and Exception Handling

Working through the first steps of this tutorial has given you a web service that works. In ideal conditions, it does what it is supposed to do. But as it now stands, the service isn't quite ready for prime time. Here are a few possible problems:

- The client may want to cancel the credit report before Investigate has sent back a response.

Providing this ability is especially important in asynchronous exchanges, which may be long−lived.

- Investigate's dependency on the credit card web service (which is accessed asynchronously) may make the overall response time to the client quite long.

You have no way of knowing how long it will take the credit card service to respond to your requests. To better manage this, you can use a timer to set a limit on how the resource may take to respond.

- Operation methods (such as requestCreditReportAsynch) may throw an uncaught exception.

In your current design, this would leave your service, and its client, hanging. The active conversation would continue until it expired. The client would not receive a response to its request and might never know why. You can handle such exceptions to ensure a clean recovery.

Through the following procedures, you will enhance the Investigate web service to better handle these possible problems.

This tasks in this step are:

- To add a method so that clients can cancel operations
- To add a TimerControl to limit the time allowed for response
- To handle exceptions thrown from operation methods

To Add a Method so that Clients Can Cancel Operations

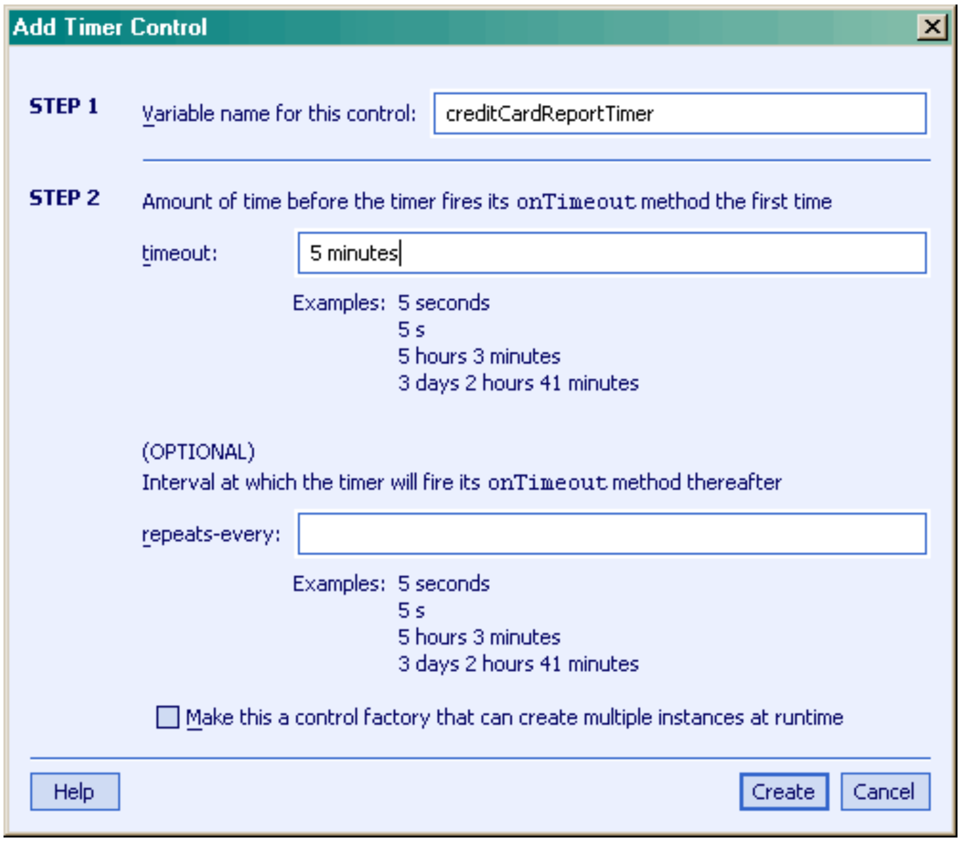
1. If you are not in Design View, click the Design View tab.
2. From the Add Operation drop−down list, select Add Method.
3. In the field that appears, type cancelInvestigation and press Enter. the method appears in Design View.
4. Click cancelInvestigation to view its code in Source View.
5. Edit the cancelInvestigation method code so it appears as follows:

/** * @jws:operation * @jws:conversation phase="finish" */ public void cancelInvestigation() {   /* Cancel the request to the credit card company because it is now unnecessary. */
   creditCardReportControl.cancelRequest();   /* Use the callback to send a message to the client. Note that this also ends   * the conversation because the callback's conversation phase property is set to "finish". */
   callback.onCreditReportDone(null, "Investigation canceled at client's request."); }

To Add a TimerControl to Limit the Time Allowed for Response

It can be difficult to predict how long an asynchronous resource will take to respond to a request. For example, Investigate's call to the credit card service may take hours. Here, you will add a way to limit the amount of time the credit card's web service has to respond. Using a TimerControl, you can specify an amount of time after which the operation should be canceled.

1. Click the Design View tab to return to Design View.
2. From the Add Control drop−down list, select Add Timer Control. The Add Timer Control dialog appears.
3. Enter values as shown in the following illustration:


Add Timer Control

STEP 1 Variable name for this control: creditCardReportTimer

STEP 2 Amount of time before the timer fires its onTimeout method the first time

timeout: 5 minutes

Examples: 5 seconds
5 s
5 hours 3 minutes
3 days 2 hours 41 minutes

(OPTIONAL)
Interval at which the timer will fire its onTimeout method thereafter

repeats-every:

Examples: 5 seconds
5 s
5 hours 3 minutes
3 days 2 hours 41 minutes

☐ Make this a control factory that can create multiple instances at runtime

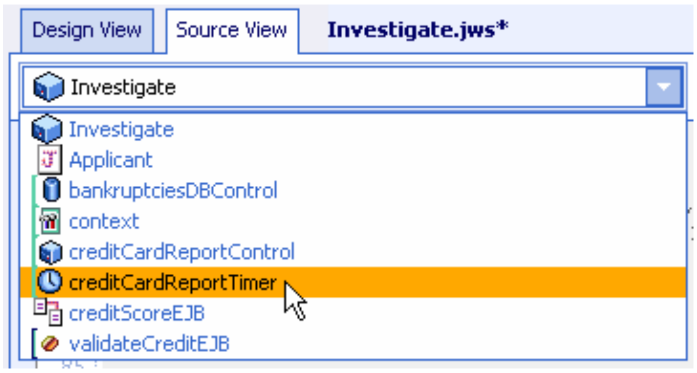
These values specify that the TimerControl will send its onTimeout callback five minutes after the timer starts.

4. Click Create. You are returned to Design View.
5. Click requestCreditReportAsync to view its code in Source View.
6. At the very end of the requestCreditReportAsynch source code, shown below, add the code shown in bold:

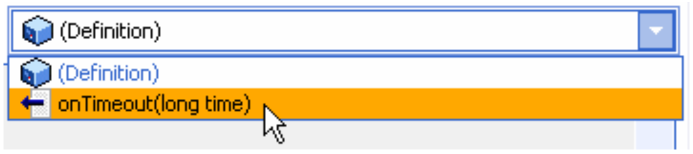
creditCardReportControl.getCreditCardData(taxID); creditCardReportTimer.start();

This start the timer as soon as the credit card service is called. If the service does not respond by the time the timeout duration is reached, then the timer's onTimeout callback handler is invoked.

7. In Source View, immediately beneath the Source View tab, from the class drop−down list, select creditCardReportTimer, as shown here:



8. To the right of the class drop−down list, from the member drop−down list, select onTimeout, as shown here



The source code for the JMS control's callback handler appears.

9. Edit the callback handler source code so that it appears as follows:

private void creditCardReportTimer_onTimeout(long time) {   /* Because the credit card service has not yet returned, cancel the request. */   creditCardReportControl.cancelRequest();   /* Send a response to the client, saying something about what happened.   * Remember that this will also effectively finish the conversation. */   callback.onCreditReportDone(null, "Unable to retrieve credit card information."); }

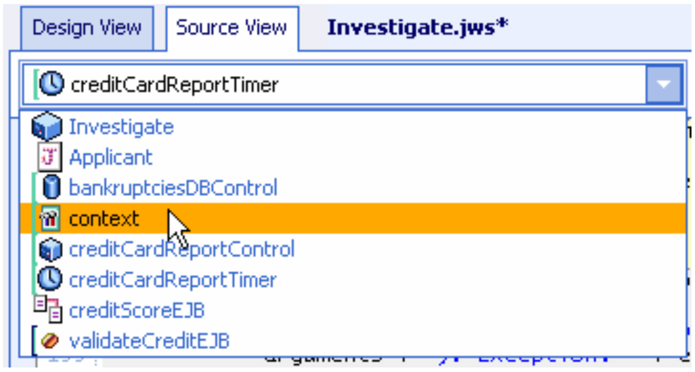
To Handle Exceptions Thrown from Operation Methods

Unhandled exceptions thrown from operations (such as methods exposed to clients) can interrupt your service's work and leave the client hanging. Such exceptions do not automatically end a service's conversation. Instead, the service may simply continue on the system, unnecessarily using resources. As a result, one action your code should take when responding to such exceptions is to finish the conversation.

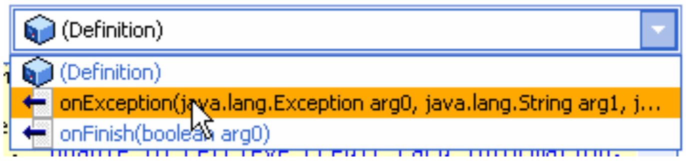
The JwsContext interface that provides the finish method for ending the conversation provides other useful functionality. In this step, you will add code to implement a handler for the JwsContext.onException callback. The callback handler receives the exception object thrown from the operation, the name of the method that threw the exception, and the parameters passed to the method.

WebLogic Workshop provides a way for you to preserve information about exceptions by logging it in a text file. You will add exception handling code to log the exception, then send an appropriate response to the client.

1. In Source View, immediately beneath the Source View tab, from the class drop−down list, select context, as shown here:



2. To the right of the class drop−down list, from the member drop−down list, select onException, as shown here:


(Definition)
(Definition)
onException(java.lang.Exception arg0, java.lang.String arg1, j...
onFinish(boolean arg0)

The source code for the JwsContext onException callback handler appears.

3. Edit the onException callback handler code so that it appears as follows:

public void context_onException(Exception e, String methodName, Object[] arguments) { /* Create a logger variable to use for logging messages. Assigning it the * "Investigate" category name will make it easier to find messages from this * service in the log file. */ Logger logger = context.getLogger("Investigate"); /* Log an error message, giving the name of the method that threw the * exception and a stack trace from the exception object. */ logger.error("Exception in " + methodName + ": " + e); /* Invoke the callback to send the client a response. */ callback.onCreditReportDone(null, "Unable to respond to request at this time. " + "Please contact us at 555−5555."); }

4. At the top of Investigate's source code, add the following line of code as the last of the imports:

import weblogic.jws.util.Logger;

Note: This imports the Logger class that makes logging possible.

By default, for the domain in which you develop and debug WebLogic Workshop web services, the log file is located at the following path of your WebLogic Workshop installation:

BEA_HOME/weblogic700/samples/workshop/jws.log

An entry in the log will resemble the following:

16:18:11 ERROR Investigate : Exception in requestCreditReportAsynch: <exceptionStackTrace> [ServiceException]

Because you're probably anxious to move on to creating an actual client for the Investigate service, you won't test the code you've added here. But if you want to try it out some time, you might do the following:

- Test the cancellation code by starting the service, then cancelling it in Test View. When Test View launches, enter a tax ID for testing. After the test has begun, click the "Continue this conversation" link. When the page refreshes, click the cancelInvestigation button.
- Test the timer by changing its timeout value to 1 second. Unless your computer is very fast, this should be enough time for the credit card service to return before the timeout. After changing the timeout value, start the service and wait one second, then click the Refresh link.
- Test the exception handler by throwing an exception from somewhere in your code.

Related Topics

Timer Control: Using Timers in Your Web Service

JwsContext.onException Callback

How Do I: Handle Errors in a Web Service?

Click one of the following arrows to navigate through the tutorial.

Step 8: Client Application

In this step you will build a Java console client to invoke your web service. Although your web service is currently designed to support a client capable of receiving callbacks, building a client that can receive callbacks is beyond the scope of this tutorial. As an alternative, you will modify your web service to support a polling interface. That is, you will modify your web service so that a client can periodically make requests to, or poll, the web service for the results.

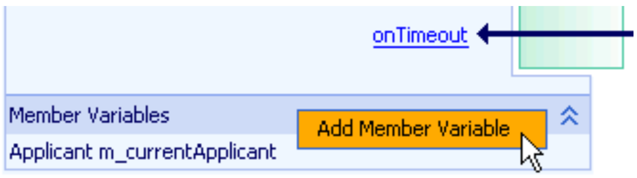
The tasks in this step are:

- To modify your web service to support a polling interface
- To save the web service's proxy classes
- To compile and run the Java client application

To Modify the Web Service to Support a Polling Interface

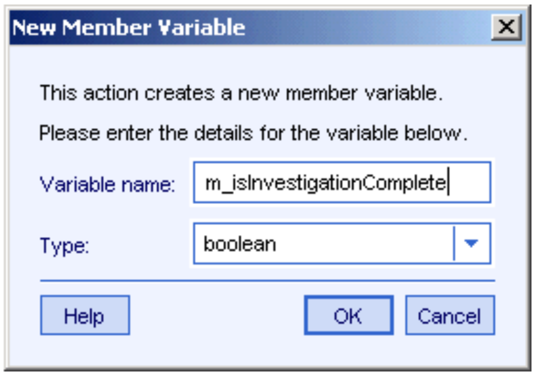
In this section, you will add a new member variable to your web service called m_isInvestigationComplete so that your web service will know when it is time to provide the complete results to the Java client. This member variable records whether the web service has completed its task of building a credit profile of an applicant. When m_isInvestigationComplete is false, the web service returns a null response each time the Java client polls for information. Once m_isInvestigation is true, the web service returns a full profile of the credit applicant to the Java client poll.

1. If you are not in Design View, click the Design View tab.
2. Right−click Member Variables and select Add Member Variable, as shown here:



The New Member Variable dialog appears.

3. In the Variable name field, type m_isInvestigationComplete, as shown below.
4. Confirm that the value in the Type drop−down list is boolean, as shown below.


New Member Variable

This action creates a new member variable.

Please enter the details for the variable below.

Variable name: m_isInvestigationComplete

Type: boolean

5. Click OK.
6. Click receiveJMSMessage. The creditScore_receiveJMSMessage code appears in Source View.
7. Add the code in bold, shown here:

```
private void creditScoreEJB_receiveJMSMessage(Message msg)
    throws JMSException, java.rmi.RemoteException
{
    creditScoreTimer.stop();

    m_currentApplicant.creditScore = ((MapMessage)msg).getInt("credit_score");
    System.out.println(m_currentApplicant.creditScore);
    m_currentApplicant.approvalLevel = validateCreditEJB.validate(m_currentApplicant.credit
    if(context.getCallbackLocation() != "")
        callback.onCreditReportDone(m_currentApplicant, "Credit score received.");
    else
        m_isInvestigationComplete = true;
}
```

8. Add the following method to the web service:

```
/**
 * @jws:operation
 * @jws:conversation phase="continue"
 */
public Applicant checkForResults()
{
    Applicant retval = null;
    if(m_isInvestigationComplete == true){
        retval = m_currentApplicant;
        context.finishConversation();
    }
    return retval;
}
```

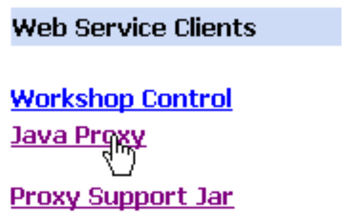
9. Press Ctrl+S to save Investigate.jws.

To Save the Web Service Proxy Classes

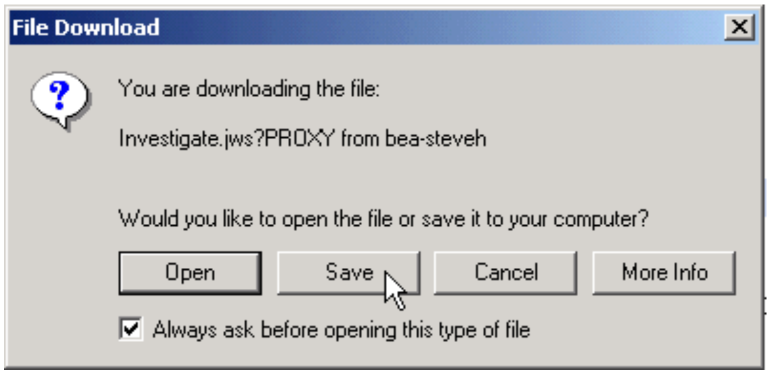
In this task, you will save two JAR files (Java Application Archive files) that will form an interface between your web service and the Java console client. These JAR files contain Java classes which form a proxy of your web service, a proxy through which your client can invoke the actual web service.

1. Press F5 to compile your web service. Your web browser launches Test View.

2. In the browser window showing Test View there are five tabs at the top of the screen: Overview, Consol, Test Form, Test XML, and Warnings. Click the Overview tab.
3. Under Web Service Clients, click Java Proxy, as shown here:



The File Download dialog appears, as shown here:


File Download

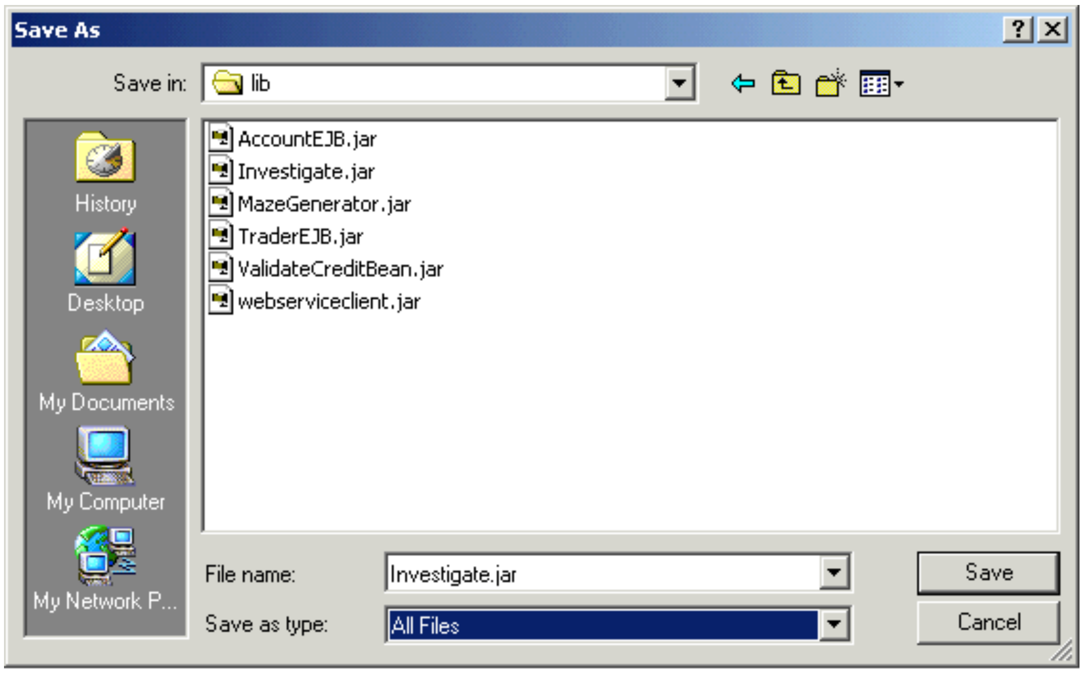
You are downloading the file:

Investigate.jws?PROXY from bea-steveh

Would you like to open the file or save it to your computer?

☑ Always ask before opening this type of file

1. Click Save. The Save As dialog appears.
2. In the Save in drop−down list, navigate to C:\bea\weblogic700\samples\workshop\applications\samples\WEB−INF\lib
3. In the Save as type drop−down list, make sure that All Files is selected, as shown here:


Save As

Save in: lib

AccountEJB.jar
Investigate.jar
MazeGenerator.jar
TraderEJB.jar
ValidateCreditBean.jar
webserviceclient.jar

File name: Investigate.jar

Save as type: All Files

6. Click Save to save Investigate.jar. The Download Complete dialog appears.
7. Click Close.

8. Repeat the entire process for the Proxy Support Jar file.

To Compile and Run the Java Client Application

Now that the JAR files are in place you are ready to compile and run the Java client application.

1. Open a command prompt window.
2. At the command prompt, cd to the following directory:
   C:\bea\weblogic700\samples\workshop\applications\samples\tutorials\java_client
3. To compile the Java console client, type compile and press Enter.
4. To run the Java console client, type run and press Enter. The command prompts you to enter a tax identification number for an applicant.
5. Type one of the following 9 digit numbers:

123456789, 111111111, 222222222, 333333333, 444444444, 555555555

The client now polls the web service every second until the credit report is ready. Once it is ready, the Java console application displays the results.

Note: you may a get warning that your client does not support callbacks. You can ignore this warning because the Java console client does not rely on receiving the web service response as a callback.

Click one of the following arrows to navigate through the tutorial.

Step 9: Deployment and Security

Your web service is one that sends sensitive information across the internet, so it would be a good idea to insure that your service is a secure one. In this step of the tutorial you will modify your web service so that is it exposed through HTTPS (Secure HTTP) instead of HTTP.

Note that the web services you build with Workshop compiled as EAR files, and then are ultimately deployed to WebLogic Server as web applications. When finally deployed on WebLogic Server, your web service is contained within a web application and exposed through that web application. It is the containing web application that is exposed through an HTTPS−enabled port of WebLogic Server, so a web service is secure because it is contained in a web application that is exposed through a HTTPS−enabled port.

The tasks in this step are:

- To configure the web service to be exposed on HTTPS
- To package the web application as an EAR file
- To deploy the web application on a production server

To Configure the Web Service to Be Exposed on HTTPS

You configure the exposure protocol of your web service by editting the weblogic−jws−config.xml file that resides in the samples project's WEB−INF directory.

1. In the Project pane, open the WEB−INF directory.
2. Open the file weblogic−jws−config.xml.
3. Edit weblogic−jws−config.xml to look like the following. Make sure to remove the comment tags (<!−− and −−>) to activate the XML elements:

```
<config> <protocol>http</protocol> <hostname>localhost</hostname> <http−port>7001</http−port>
<https−port>7002</https−port> <jws>  <class−name>financialservices.Investigate</class−name>
```

```
<protocol>https</protocol> </jws> ... </config>
```

To Package the Web Application as an EAR File

1. Stop WebLogic Server and close WebLogic Workshop
2. Using Windows Explorer, create a new directory C:\EARS
3. In a new command window, cd to the directory in which Investigate.jws resides. If you installed WebLogic Server on the C drive, the directory is:

```
C:\bea\weblogic700\samples\workshop\applications\samples\financialservices
```

4. Edit the PATH environmental variable by pressing Start−−>Settings−−>Control Panel. Double click System. Click the Advanced Tab. Click the Environmental Variables... button. Under System Variables select the Path variable. Click the Edit... button. Append the following to the Variable Value:
   C:\bea\weblogic700\server\bin
   Make sure that a semi−colon separates the value you append and the values already present.

5. Run the following command.

```
jwsCompile −ear C:\EARS\MyFirstWebService.ear −app MyFirstWebService Investigate.jws
```

Note that you may get a warning about the namespace of your web service. For the sake of the tutorial, you can ignore these warnings. If you are planning to deploy a web service in a real production scenario, you would probably want to change the namespace to something more unique than "openuri".

To Deploy the Web Application on a Production Server

1. In a new command window, start WebLogic Server in production mode using the following:

```
C:\bea\weblogic700\samples\workshop\startWeblogic.cmd production nodebug
```

2. Edit the PATH environmental variable by pressing Start−−>Settings−−>Control Panel. Double click System. Click the Advanced Tab. Click the Environmental Variables... button. Under System Variables select the Path variable. Click the Edit... button. Append the following to the Variable Value:
   C:\bea\jdk131_03\bin
   Make sure that a semi−colon separates the value you append and the values already present.
3. In a new command window, run the following:

```
java −cp C:\bea\weblogic700\server\lib\weblogic.jar weblogic.Deployer −password installadministrator
−source C:\EARS\MyFirstWebService.ear −targets cgServer −name MyFirstWebService −activate
```

3. Open an internet browser, and enter the following address:

```
http://localhost:7001/console
```

4. Logon to WebLogic Server using the password "installadministrator".
5. Navigate to workshop/Deployments/Applications/MyFirstWebService, using the navigation pane on the left−hand side of the screen.
6. Note the two files contained in MyFirstWebService: MyFirstWebService.war and financialservices.InvestigateEJB.jar
7. Open another internet browser, and enter the following address:

https://localhost:7002/MyFirstWebService/financialservices/Investigate.jws

8. The server will present you with a digital certificate, and a summary of the Investigate web service will appear.
9. Modify the address to look like the following:

https://localhost:7002/MyFirstWebService/financialservices/Investigate.jws?WSDL

A WSDL file will appear. This is the WSDL that clients to your web service will use to learn how to operate your service.

You now have a complete and deployed web service ready to take requests from clients.

Summary: Your First Web Service

This tutorial introduced you to the basics of building web services. Along the way, you became acquainted not just with how to use WebLogic Workshop, but with considerations in the design of web services.

This topic lists ideas this tutorial introduced, along with links to topics for more information. You may also find it useful to look at the following:

- For an overview of WebLogic Workshop, see Building Web Services with WebLogic Workshop.
- For a links to information on tasks you can accomplish in WebLogic Workshop, see the How Do I? topics.
- For a list of the samples provided, see Samples.

Concepts and Tasks Introduced in This Tutorial

- You interact with web service source files through a WebLogic Workshop project. You use projects to group files associated with a web service or related web services.

For more information about projects, see WebLogic Workshop Projects.

- As you develop web services, you test and debug code on a running instance of WebLogic Server.

For information on WebLogic Server, see BEA WebLogic Server 7.0 Release Documentation. To learn how to start WebLogic Server from within WebLogic Workshop, see How Do I: Start or Stop WebLogic Server?

- You design a web service as you might make a drawing of it and its relationships with clients and other resources. WebLogic Workshop provides a Design View you can use to generate code to start with as you create your design.

For reference information on Design View, see Design View.

- The source file for a web service is a JWS file, which is automatically recognized as a web service when deployed with WebLogic Server.

For an introduction to JWS files, see JWS Files.

- You expose the functionality of a web service by creating methods within a JWS file. You can add the method in Design View, setting properties to specify its characteristics, including powerful server features.
- To test a service, you use the Test View, a dynamically generated HTML page through which you can invoke service methods with specific parameter values.

For reference information on Test View, see Test View.

- You can easily access databases from your web service with the DatabaseControl.

For an introduction and detailed information about the DatabaseControl, see Database Control: Using a Database from Your Web Service.

- You can use asynchronous communications to handle latency issues inherent on the Internet, and sometimes with web services in general. Through asynchrony, you avoid situations in which client software is blocked from proceeding until it receives the results of its requests.

For an introduction to asynchrony and the WebLogic Workshop tools that support it, see Using Asynchrony to Enable Long−Running Operations.

- Callbacks provide a useful way to return results to clients where asynchronous communication is needed. Callbacks require an agreement that the client will implement a means to handle the callback a service makes.
- For more details, see Using Callbacks to Notify Clients of Events.
- You can use conversations to maintain consistent state even between disparate asynchronous exchanges. Conversations make it possible to correlate these exchanges with the original request and the client that made it.

For more information, see Maintaining State with Conversations and Using Asynchrony to Enable Long−Running Operations.

- You can access other web services using the ServiceControl. By generating a ServiceControl from the WSDL of another web service. When you do, you have a single component to represent the other service in your service's design.

For an introduction and details about the ServiceControl, see Service Control: Using Another Web Service.

- You can access components that are available via the Java Message Service (JMS) by using the JMSControl. Through this control, you can send and receive messages of various types, including XML.

For more on the JMSControl, see JMS Control: Using Java Message Service Queues and Topics from Your Web Service.

- You can access Enterprise Java Beans (EJBs) through the EJBControl. The EJBControl simplifies your use of an EJB by providing a single component representing the EJB's interface in your web service design.

For more detail on the EJBControl, see EJB Control: Using Enterprise Java Beans from a Web Service.

- When you need to handle or control the specific shape of XML messages your service exchanges with other components, you can use XML maps. XML maps customize WebLogic Server's translation of XML to Java and vice versa.

For more on XML maps, see Handling and Shaping XML Messages with XML Maps.

- For calculations outside your Java code, and for more powerful control over the shape of XML messages, you can incorporate ECMAScript into mapping. To use ECMAScript, you write a script function in a JSX file, then refer to the function in your XML map. At run time, WebLogic Server calls the script function when translating between XML and Java.

For an introduction and details about using ECMAScript for mapping, see Using Script Functions From XML Maps.

- With the TimerControl, you can add timer functionality to your service. In this way, you can limit the time specific operations are allowed to execute, cause something to happen and regular intervals, and so on.

For more about the TimerControl, see Timer Control: Using Timers in Your Web Service.

- You can use the onException callback exposed by the JwsContext interface to handle exceptions thrown from your web service's operations. When prompted by this callback, your code can perform any necessary clean−up and send a message to the client.

For reference information about the onException callback, see JwsContext.onException Callback.

- By supporting polling, your web service can offer an alternative to clients that aren't capable of receiving callbacks. With polling, a client can periodically ask your web service if the response to its request is ready yet.

For information about polling, see Using Polling as an Alternative to Callbacks.

- As you build your web service, WebLogic Workshop generates classes that can be used by client software. Through these proxy classes, a client can invoke your service's methods. You can download the proxy classes from Test View.

For more information, see How Do I: Use the Java Proxy for a Web Service?

- You can make your web service secure by ensuring that it is exposed via the https protocol rather than http. To do this, you edit the configuration file associated with your web service's project.

For more detail, see Using HTTPS to Secure a Workshop Web Service.

- Using the JwsCompile command, you can package your web service for deployment to a production server.

For a topic on web service deployment, see Deploying Web Services. For reference information on JwsCompile, see JwsCompile Command.

The topics in this section provide the detailed information you need to build a sophisticated web service with WebLogic Workshop.

WebLogic Workshop Projects

Explore the structure and content of a WebLogic Workshop project.

Controls: Using Resources from a Web Service

Learn how to use controls to access an enterprise application, database, Enterprise JavaBean, messaging service, or another web service.

Using Asynchrony to Enable Long−Running Operations

Build robust, asynchronous web services to coordinate long−running operations.

Maintaining State with Conversations

Track and maintain application state with conversations.

Handling and Shaping XML Messages with XML Maps

Manipulate incoming and outgoing XML messages to build loosely−coupled web services.

Debugging Web Services

Debug your web services with the WebLogic Workshop integrated debugger.

Documenting Web Services

Document your web services with Javadoc comments.

Security

Learn about security in WebLogic Workshop and how it interacts with WebLogic Server security.

Protocols and Message Formats

Learn about supported protocols and message formats.

Implicit Transactions in WebLogic Workshop

Learn about the implicit transactions that wrap operations in web services created with WebLogic Workshop.

Deployment and Clustering

Deploy web services in a production environment.

WebLogic Workshop Projects

All WebLogic Workshop web services are developed and run within a project. A project provides the directory structure in which web service files, including supporting files, reside.

Contents of a Project

A project may contain multiple web services. The web services in the project may be organized in several hierarchical directories (folders).

There are four types of files you typically work with. Select the links below to learn more about the various file types:

- JWS Files: Java Web Services: Java Web Service files that contain the source code to a web service.
- CTRL Files: Control files that contain interfaces to WebLogic Workshop controls, including controls that represent other web services, databases and other resources.
- WSDL Files: Web Service Descriptions: Web Service Description Language files that contain XML descriptions of your web service or of other services your web services use.
- XMLMAP Files: XML map files that contain maps that may be shared by more than one method and/or web service.
- JSX files: ECMAScript (also known as JavaScript) files that use the ECMAScript XML extensions defined by WebLogic Workshop to perform mapping between XML and Java. JSX files are referred to by XML maps in JWS files. See Handling XML with ECMAScript Extensions.
- JAVA files: You may include additional JAVA files in your project, and web services may access classes defined in JAVA files.

Note: Files in the project may be served in source form by WebLogic Server. To avoid exposing your JAVA source files to clients, place them a subdirectory of your project's WEB−INF directory (but not in the classes or lib subdirectories).

In addition to the files you use to implement web service code, projects also contain files and directories that are required by WebLogic Server to support your web services. These directories will be most useful to advanced users.

You may notice the following project directory in a directory:

- WEB−INF: This directory contains supporting code for your web services. This supporting code may include Java classes that you have added so that you may use them from your code. Code that is automatically generated by WebLogic Workshop to implement your web services is also placed here. You may add class files to the classes folder or jar files to the lib folder, but you should not modify the other contents of this directory.

If you already have some experience using WebLogic Server, you may recognize that a WebLogic Workshop project is very similar to a WebLogic Server Web Application (webapp).

Location of WebLogic Workshop Projects

A WebLogic Workshop project must be a valid WebLogic Server webapp, which must in turn reside in a WebLogic Server domain. The default location for WebLogic Workshop projects is <install>/weblogic700/samples/workshop/applications.

Note: <install> refers to your WebLogic Server installation directory. If you accepted the defaults during installation, the value of <install> is:

- c:\bea on systems running Microsoft Windows
- /home/<user>/bea on systems running Linux
- $HOME/bea on systems running Solaris

WebLogic Workshop is installed with two projects: 'samples' and 'DefaultWebApp'.

The 'samples' Project

WebLogic Workshop is installed with a project named 'samples'. The samples project contains many example web services that demonstrate specific features or capabilities of WebLogic Workshop web services.

You may create additional web services in the samples project. When you begin to develop web services that you intend to deploy and publish, create one or more new projects to hold the new services.

For descriptions of the web services in the samples project, see Samples.

Project Name

Web services are accessed via URLs. For example, the Hello World sample web service, which is located in the samples project is accessed by the following URL:

http://localhost:7001/samples/HelloWorld.jws

Notice that the project name is part of the URL for the web service. This is true for all WebLogic Workshop web services. Since you will be publishing your web services by publishing their URLs, you should choose project names that will make sense in the resulting URLs.

Note: In the example URL above, localhost would be replaced by an actual hostname or IP address in the CTRL or WSDL files used to publish the web service's location to others.

Browsing a WebLogic Workshop Project

WebLogic Workshop provides a facility for browsing WebLogic Workshop projects. To access this facility, browse to the URL of any WebLogic Workshop project with /jwsdir appended to the URL. For example:

http://localhost:7001/samples/jwsdir

The directory shows only JWS files by default. Select Show all files to see other file types and subdirectories.

In addition to allowing a user to explore the structure of a project and access web service Test View pages, this view also displays the "Clean All" button. This button may be used to clear the state of web services deployed in WebLogic Server. This can be useful when errors in a web service under development lead to erroneous server behavior.

Related Topics

Web Service Development Cycle

Controls: Using Resources from a Web Service

Controls provide a common model to interacting with resources from within a web service. If you access a resource such as a relational database through a control, your interaction with the resource is greatly simplified because the underlying control implementation takes care of most of the details for you.

All controls expose Java interfaces that may be invoked directly from web service code. So you use all controls the same way: you create an instance of the specific control you want to use and then invoke its methods and implement handlers for its callbacks.

All controls except Timer controls are implemented in CTRL files. To learn more about CTRL files, see CTRL Files.

Control Types

WebLogic Workshop provides the following six types of controls to help you interact with resources:

Application View Control

The Application View control allows your service to access an enterprise application using an Application View.

For more information on the Application View control, see Application View Control: Accessing an Enterprise Application from a Web Service.

Timer Control

The Timer control notifies your service when a specified period of time has elapsed or when a specified absolute time has been reached.

For more information on the Timer control, see Timer Control: Using Timers in Your Web Service.

Service Control

The Service control provides an interface to another web service, allowing a calling service to access the methods and handle the callbacks of the service represented by the Service control.

For more information on the Service control, see Service Control: Using Another Web Service.

Database Control

The Database control provides simplified access to a relational database, allowing a web service to call Java methods and operate on Java objects that are appropriate to the operations being performed.

For more information on the Database control, see Database Control: Using a Database from Your Web Service.

EJB (Enterprise Java Bean) Control

The EJB control provides access to an existing Enterprise Java Bean (EJB). Many repetitive details that must be addressed when using EJBs are hidden and automated by an EJB control.

For more information about the EJB control, see EJB Control: Using an Enterprise Java Bean from Your Web Service.

JMS (Java Message Service) Control

The JMS control proves access to an existing Java Message Service (JMS) queue or topic. Many repetitive details that must be addressed when using JMS queues and topics are hidden and automated by the JMS control.

For more information about the JMS control, see JMS Control: Using Java Message Service Queues and Topics from Your Web Service.

Related Topics

CTRL Files: Implementing Controls

Using a Control

This topic describes how to use a control from within your web service. It explains how to:

- locate an existing control
- use an existing control
- import and declare a control
- invoke a control method
- handle control callbacks
- handle control−generated exceptions

This topic assumes you have access to a control that you either implemented yourself or that was implemented by another developer. You have access to a control if you have access to its CTRL file in your project. The sections below describe how to modify the control's CTRL file and your web service's JWS file in order to access the control.

To learn about controls, see the Controls: Using Resources from a Web Service.

Using an Existing Control

You can bring an existing control into your web service in several ways: you can add it, you can copy it, or you can reference it. Each method has specific implications. The sections that follow describe each of these methods in greater detail so you can choose which method will work best for a given circumstance.

Wherever you decide to place the control's CTRL file, it must be in your WebLogic Workshop project. Java imposes rules on references between packages that prevent you from using a control if it is not located in the correct place. If you place the CTRL file in the root folder of your project, web services that are not in the root folder (i.e., the Java default package) will not be able to use the control.

To learn about WebLogic Workshop project organization, see WebLogic Workshop Projects.

To use an existing control from your web service, locate the control's CTRL file in the Project Tree and drag the CTRL file onto the Design View for your web service.

Referencing an Existing Control

If you have an existing CTRL file in your project, you can add a reference to that control in any web service by dragging the CTRL file from the Project Tree to the Design View of the web service from which you would like to use the control. You may also use the Add Control menu in the upper right corner of Design View. Any change you make to the control in Design View will effect the referenced CTRL file, which means it will also effect any other web services that are referencing the same CTRL file.

Copying the Control CTRL File

If the CTRL file for the control you wish to use is not in your project, you must copy it to your project. The destination to which you copy the control depends on your expected usage. If the control will be used only by a single web service in your project, you may choose to copy the CTRL file to the same folder as your web service. If multiple web services in different folders in your project may use the control, you may wish to copy the control's CTRL file to a common folder. Remember that the CTRL file must be in your WebLogic Workshop project.

Wherever you decide to place the control's CTRL file, it must be in your WebLogic Workshop project.

Note: In Java terminology, a class that is defined in the default package may not be referenced from a class that is in a named (non−default) package. This means that if you place a CTRL file in the root folder of your project, you cannot reference the control from any web service that is not in the root folder. To learn about Java packages, see Introduction to Java.

Be aware that if you copy a CTRL file to a new location, you are creating a new control. If you do not change the definition of the copy, you now have two identical but separate controls. If the original copy of the control is subsequently changed, the second copy will remain unchanged. This means that any bug fixes or feature enhancements made to the original control will not be reflected in the copy.

If you want multiple web services to use the same control and you want changes in the control to affect the behavior of all web services that use the control, then you want to reference the original control instead of copying the control.

You may reference any control in your project by creating an appropriate import statement in your web service's JWS file. To learn how to import a control, see the Importing and Declaring the Control section below.

Modifying the Control's Package

If you drag a CTRL file from one location to another within your WebLogic Workshop project, the package statement is automatically modified for you and you don't need to read the remainder of this section. However, if you decide to copy a control's CTRL file in some other way than dragging it in the Project Tree, you must change the package statement in the CTRL file.

In Java the location of a source file in the directory hierarchy determines (and must agree with) the Java package to which the classes in the file belong. If a Java file is located in the <project>\controls\financial directory, the Java package statement in the file must be the following:

```
package controls.financial;
```

After copying a control's CTRL file to a new location, you must change the package statement in the file to reflect the new location.

For more information on Java packages, see Introduction to Java.

Importing and Declaring the Control

If you specify that you want your web service to use a control by dragging that control's CTRL file onto the Design View of your web service, the necessary code changes are made to your web service's JWS file automatically and you do not need to read the rest of this section. However, if you declare a control instance manually in your web service's JWS file, you need to do three things: import the control's class, declare the control instance, and annotate the control instance declaration correctly so that WebLogic Workshop can connect the control. These procedures are described in this section.

To use the control from within a web service, import the control's main interface in the web service's JWS file using a Java import statement. The import typically resembles the following form:

```
import controls.financial.CurrencyExchangeControl;
```

For more information on Java import statements, see Introduction to Java.

Once you have imported the control, you must declare an instance of the control before you can use it in your web service. To declare the control you must provide an appropriate Javadoc comment that identifies the declaration as a control, and you must declare the member variable that will represent the control in your web service as shown in the following example:

```
/**
 * @jws:control
 */
private CurrentExchangeControl currency;
```

The @jws:control tag indicates to the WebLogic Workshop user interface that the object declared in the associated code should be treated as a control in WebLogic Workshop and that WebLogic Workshop should connect the control to its supporting code.

The declaration itself must mention an interface that extends (indirectly) the weblogic.jws.control.Control interface.

It is customary to declare control instances to be private within a web service's JWS file because it typically doesn't make sense to use a control instance from outside of a JWS method.

Invoking a Control Method

Once a control has been declared as described in the preceding section, you can access the methods of the control using the standard Java dot notation. For example, if the control named CurrencyExchangeControl defines the method shown here:

```
String [] getAllAvailableCurrencyNames();
```

then you can access the method from within your web service JWS file using the following code:

```
String [] currencyNames;
currencyNames = currency.getAllAvailableCurrencyNames();
```

Handling Control Callbacks

Some control types allow the specification of callbacks. Callbacks provide a way for a control or a web service to asynchronously notify a client that an event has occurred.

For a discussion of asynchrony, see Using Asynchrony to Enable Long−Running Operations.

A callback is a method signature that is defined by the control (or web service) where the method implementation must be provided by the client. The client enables reception of a callback by implementing a callback handler.

Callback Definition

A callback definition in a control may look like the following:

```
void onReportStatus(String status);
```

This declaration would be made in the source code for the service or control that defines the callback.

It is common for callbacks to have names that begin with "on" because a callback represents an event and the callback handler in the client will be called on occurrence of the event.

Callback Handler Definition

The client is responsible for implementing a handler for any callback it wishes to receive. To implement a callback handler for the example callback in the previous section, the JWS file for a web service would include the following:

```
void exampleControl_onReportStatus(String status)
{
    <take appropriate action given status>
}
```

In WebLogic Workshop, callback handler names are determined by the name of the control instance and the name of the callback. In the example above, the control instance from which we wish to receive the callback is exampleControl and the callback defined by the control is named onReportStatus. The full name of the callback handler, exampleControl_onReportStatus, is the control instance name followed by an underscore and the name of the callback.

Handling Control Method Exceptions

The designer of a control may choose whether or not to explicitly declare that exceptions are thrown by the control's methods. If a control method is declared to throw exceptions, you must enclose your invocations of that method in a try−catch block.

Even if the designer of the control choses not to declare exceptions, the support code that implements the control may still throw exceptions. Such exceptions are always thrown as weblogic.jws.control.ControlException.

You should strongly consider handling all exceptions that may be thrown by the controls you use in your web service. If you do not handle an exception thrown by a control, the web service method will fail and the exception will be passed on to the client of your web service. In most cases, the exception is useless to the client and the client does not have the necessary information to diagnose or remedy the problem.

To learn more about exceptions and try−catch blocks, see Introduction to Java.

Related Topics

Web Service Development Cycle

Application View Control: Accessing an Enterprise Application from a Web Service

Note: The Application View control uses Application Views defined using the Application Integration (AI) component of WebLogic Integration. The Application View control is available in WebLogic Workshop only if you are licensed to use WebLogic Integration's AI component.

Note: In this release, the Application View control is supported only in the sample workshop domain. In future versions, support will be provided for enabling the Application View control in other domains.

The Application View control allows your service to access an enterprise application using an Application View. An Application Views must be created using WebLogic Integration's Application Integration (AI) component before it can be referenced using an Application View control. To learn more about application views and their relationship to enterprise applications, see Overview: Adapters and Application Views.

Like other WebLogic Workshop controls, the Application View control allows WebLogic Workshop web services to interact with enterprise applications using simple Java APIs. They allow a web service developer to access an enterprise application even if they don't know any of the details of the application's implementation.

The Application View control provides a means for a web service developer to invoke Application View services both synchronously and asynchronously, and to subscribe to Application View events. In both the service and event cases, the developer uses simple Java APIs. The developer need not understand XML, or the particular protocol or client API for the enterprise application (hereafter referred to as an Enterprise Information System or EIS).

Topics Included in this Section

Overview: Adapters and Application Views

Describes enterprise application Adapters and WebLogic Integration Application Views.

Creating a New Application View Control

Describes how to create and configure an Application View control.

Updating an Application View Control

Describes how to update an Application View control when the underlying Application View changes.

Using an Application View Control

Describes how to use an existing Application View control from within a web service.

WebLogic Workshop Is Not Intended for Application View Development

The Weblogic Workshop Application View control is designed to make it easy for you to use an existing, deployed Application View from within your web services. WebLogic Workshop is specifically not designed to help you develop and deploy Application Views. Please consult the WebLogic Integration documentation to learn how to create and deploy Application Views.

Related Topics

Controls: Using Resources from a Web Service

Overview: Adapters and Application Views

The Application View control allows WebLogic Workshop web services to access enterprise applications (also called Enterprise Information Systems or EISs). An EIS is typically a large−scale business application such as a Customer Relationship Management (CRM), Enterprise Resource Planning (ERP) or Human Resources (HR) application. Examples of EISs include SAP, PeopleSoft or Siebel.

Adapters

In order to integrate the operations of an enterprise, the data and functions of the various EISs in an organization must be exposed. In the Java 2 Enterprise Edition (J2EE) model, EIS functionality is exposed to Java clients using an adapter (sometimes called a resource adapter or a connector) according to the J2EE Connector Architecture. Adapters for popular EISs are available from the applications' vendors, from BEA Systems and from third−party vendors.

WebLogic Integration (WLI), the enterprise integration offering BEA Systems, includes, as part of its Application Integration (AI) component, the Adapter Development Kit (ADK). A developer can use the ADK to construct adapters, which define services and events.

A service represents a message that requests a specific action in the EIS. For example, an adapter might define a service named AddCustomer that accepts a message defining a customer and then invokes the EIS to create the appropriate customer record.

An event issues messages when events of interest occur in the EIS. For example, an adapter might define an event that sends messages to interested parties whenever any customer record is updated in the EIS.

Application Views

In addition to defining and implementing adapters, the AI component of WebLogic Integration enables a developer to create Application Views. An Application View provides a layer of abstraction on top of an adapter; whereas adapters are closely associated with the specific functions available in the EIS, an application view is associated with business processes that must be accomplished by clients. The Application View converts the steps in the business process into operations on the adapter.

An application view exposes services and events that serve the business process. The WebLogic Workshop Application View control is associated with a particular Application View, and makes the services and methods of the Application View available to WebLogic Workshop web services as control methods and callbacks.

Creating a New Application View Control

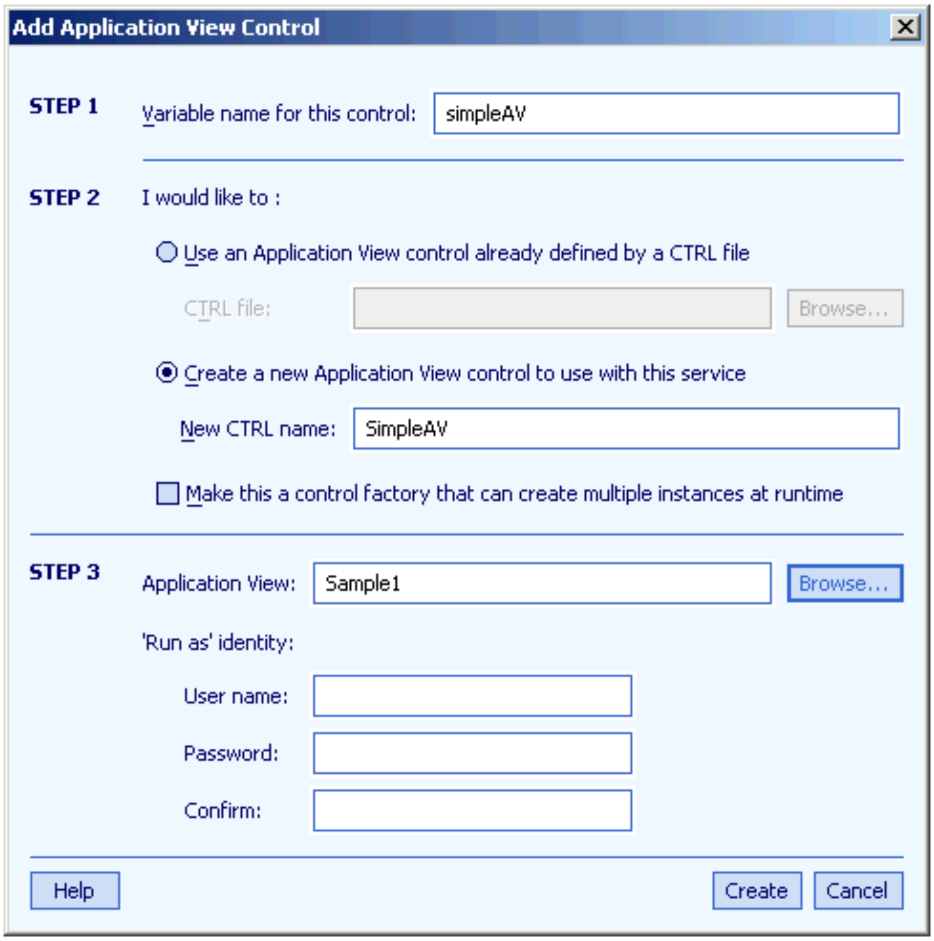
This topic describes how to create a new Application View control.

To learn about Application View controls, see Application View Control: Accessing an Enterprise Application from a Web Service.

To learn about WebLogic Workshop controls, see Controls: Using Resources from a Web Service.

To Create a New Application View Control

1. If you are not in Design View, click the Design View tab.
2. From the Add Control drop−down list in the upper−right corner of Design View, select Add Application View Control. The Add Application View Control dialog opens, as shown here:


Add Application View Control

STEP 1 Variable name for this control: simpleAV

STEP 2 I would like to :

○ Use an Application View control already defined by a CTRL file

CTRL file: Browse...

● Create a new Application View control to use with this service

New CTRL name: SimpleAV

☐ Make this a control factory that can create multiple instances at runtime

STEP 3 Application View: Sample1 Browse...

'Run as' identity:

User name:

Password:

Confirm:

3. In the Variable name for this control field, type the variable name used to access the new Application View control instance from your web service. The name you enter must be a valid Java identifier.
4. In the Step 2 pane, choose the Create a new Application View control to use with this service radio button.
5. In the New CTRL name field, type the name of your new CTRL file. The word "Control" is automatically appended to the name you enter, as is the ".ctrl" filename extension.
6. Decide whether you want to make this a control factory and select or clear the Make this a control factory that can create multiple instances at runtime check box. For more information about control factories, see Control Factories: Managing Collections of Controls.
7. In the Step 3 pane, click Browse... The Application Views Browser dialog opens, displaying Application Views that are deployed in the current domain.
8. Select the deployed Application View you want this Application View control to represent, then select OK.
9. In the Add Application View Control dialog, click Create.

To learn more about the Add Application View Control Dialog, see Add Application View Control Dialog.

Customizing an Application View Control

You can customize an Application View control in several ways. You may modify the properties of the control itself, the properties of the control's methods, or the XML maps on the control's methods. Each of these modifications is described in more detail in the sections that follow.

Control Properties

The Application View control exposes the av−identity property with the name, user−id and password attributes. For a description of the av−identity property and its attributes, see @jws:av−identity Tag.

Method Properties

Each method of an Application View control exposes the av−service property that binds the Application View method to a service of the Application View control. For a description of the av−service property and it's attributes, see @jws:av−service Tag.

XML Maps

Each method of an Application View control may have associated parameter−xml and return−xml maps, and each callback may have an associated parameter−xml map. You can edit these maps by double−clicking on the map icon associated with the method or callback in Design View.

Related Topics

Controls: Using Resources from a Web Service

Using an Application View Control

Updating an Application View Control

To update an Application View control when the target Application View changes, you must regenerate the Application View control.

Rename the old Application View control CTRL file before generating a new Application View control with the same name. If the XML maps in the old Application View control were modified, you can copy and paste them to the new CTRL file using Source View or any text editor.

Related Topics

Creating a New Application View Control

Using an Application View Control

This topic describes how to use an existing Application View control in your web service.

To learn about controls, see the Controls: Using Resources from a Web Service.

To learn about Application View controls, see Application View Control: Accessing an Enterprise Application from a Web Service.

To learn how to create a Application View control, see Creating a New Application View Control.

Using an Existing Application View Control

All controls follow a consistent model. Therefore, most aspects of using an existing Application View control are identical to using any other existing control.

To learn about using an existing control, see Using a Control.

Customizing a Application View Control

There are properties that are specific to the Application View control. If you choose to copy and customize an existing Application View control, the properties you may wish to modify are:

- av−identity
  For more information, see @jws:av−identity Tag.
- av−service
  For more information, see @jws:av−service Tag.

ApplicationViewControl Interface

All Application View controls are subclassed from the ApplicationViewControl interface. The interface defines methods that may be called on Application View control instances from a web service.

To learn more about this interface, see ApplicationViewControl Interface.

Related Topics

Creating a New Application View Control

Timer Control: Using Timers in Your Web Service

A Timer control notifies your web service when a specified period of time has elapsed or when a specified absolute time has been reached. Each Timer control is customized with a particular behavior.

Unlike most controls, a Timer control is declared directly in a JWS file.

All Timer controls are instances of the weblogic.jws.control.TimerControl base class.

To learn about WebLogic Workshop controls, see Controls: Using Resources from a Web Service.

Topics Included in this Section

Creating a New Timer Control

Explains how to create a new Timer control.

Using a Timer Control

Explains how to use a Timer Control you've already created.

How Do I: Configure a Timer Control?

Describes the processes of setting defaults for your Timer control.

Specifying Time on a Timer Control

Explains how to specify time when setting the attributes of a Timer control.

Timer Control Samples

SimpleTimer.jws Sample

AdvancedTimer.jws Sample

HelloWorldAsync.jws Sample

Related Topics

Controls: Using Resources from a Web Service

CTRL Files: Implementing Controls

Creating a New Timer Control

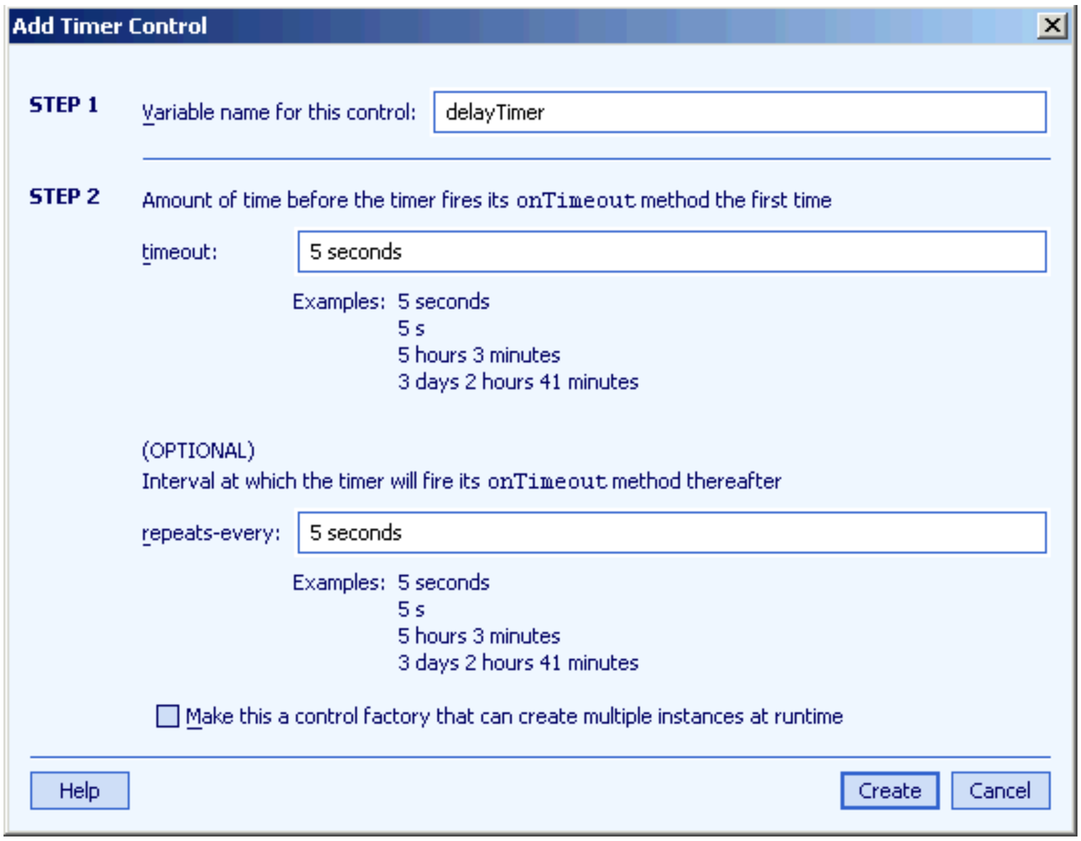
This topic describes how to create a new Timer control. Timer controls are declared locally in a web service's JWS file, so you do not need to create a CTRL file in order to define a Timer control.

To learn about Timer controls, see Timer Control: Using Timers in Your Web Service.

To learn about WebLogic Workshop controls, see Controls: Using Resources from a Web Service.

Creating a New Timer Control

1. If you are not in Design View, click the Design View tab.
2. From the Add Control drop−down list in the upper−right corner of Design View, select Add Timer Control. The Add Timer Control dialog opens, as shown here:



STEP 1 Variable name for this control: delayTimer

STEP 2 Amount of time before the timer fires its onTimeout method the first time

timeout: 5 seconds

Examples: 5 seconds
5 s
5 hours 3 minutes
3 days 2 hours 41 minutes

(OPTIONAL)
Interval at which the timer will fire its onTimeout method thereafter

repeats-every: 5 seconds

Examples: 5 seconds
5 s
5 hours 3 minutes
3 days 2 hours 41 minutes

☐ Make this a control factory that can create multiple instances at runtime

3. In the Variable name for this control field, type the name of the new control. The name you enter must be a valid Java identifier.
4. In the timeout field, specify the amount of time you want to elapse before the timer fires the first time.
5. In the repeats−every field, specify the interval between firings after the Timer fires the first time.
6. Choose whether or not you want to make this a control factory by selecting or clearing the Make this a control factory that can create multiple instances at runtime checkbox. For more information about control factories, see Control Factories: Managing Collections of Controls.
7. Click Create.

As with all controls, when you add a control to your web service, WebLogic Workshop adds the declaration of an instance of the control class to your JWS file. In the case of Timer controls, the control is always of class weblogic.jws.control.TimerControl.

For more information about the Add Timer Control dialog, see Add Timer Control Dialog.

Timer Control Declarations

In the JWS file, the declaration appears as shown here:

```
import weblogic.jws.control.TimerControl;
...
/**
 * @jws:control
 * @jws:timer timeout="5 seconds" repeats−every="5 seconds"
 */
TimerControl delayTimer;
```

The actual attributes that are present on the @jws:timer tag depend on the values you entered in the Add Timer Control Dialog. For a description of the fields in the Add Timer Control dialog, see Add Timer Control Dialog.

The @jws:control tag informs WebLogic Workshop that the associated declaration is a control. Without this tag, the control will not be properly connected to supporting code and will not function.

For more information on the @jws:control tag, see @jws:control Tag.

The @jws:timer tag controls the behavior of the Timer control. All of the attributes of the @jws:timer tag are optional and have default values.

For more information on the @jws:timer tag, see @jws:timer Tag.

The Timer control, named myTimer in the example above, is declared as an instance of TimerControl. Timer controls are unusual in that you can instantiate the control class directly. Most other controls requires that you declare a subclass of the base class (in a CTRL file) and then instantiate the new class.

Related Topics

Add Timer Control Dialog

Using a Timer Control

Specifying Time on a Timer Control

Using a Timer Control

This topic describes how to configure and use an existing Timer control.

To learn how to create a Timer control, see Creating a New Timer Control.

To learn about Timer controls, see Timer Control: Using Timers in Your Web Service.

To learn about WebLogic Workshop controls, see Controls: Using Resources from a Web Service.

Using a Timer Control

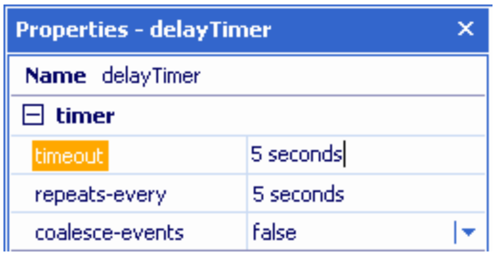
There are three aspects to using a Timer control:

- Configure default Timer control behavior using the timer property
- Control the behavior of the Timer control using the methods of the TimerControl interface to start, stop or configure it.
- Respond to timer events by handling callbacks from the Timer control.

Each of these aspects are described in the following sections.

Setting Default Timer Control Behavior

You can set the initial behavior of a Timer control using the control's timeout and repeats−every properties. For example, if you select a Timer control instance named delayTimer in Design View, the Properties Pane displays the following properties:



Name	delayTimer
⊟ timer	
timeout	5 seconds
repeats-every	5 seconds
coalesce-events	false

In the JWS file, the property values are expressed as attribute values associated with the @jws:timer tag. The @jws:timer tag has the following attributes:

- timeout, which specifies the time until the Timer control fires the first time, once started
- repeats−every, which specifies how often the Timer control should fire after the first time
- coalesce−events, which specifies how the Timer control should behave if delivery of its events is delayed

Note: The properties of a Timer control set the Timer control's initial behavior. The weblogic.jws.control.TimerControl class exposes methods that may be used to change the Timer control's behavior after it is created. See Using Methods of the TimerControl Interface below.

To learn more about specifying default Timer control behavior with attributes of the @jws:timer tag, see @jws:timer Tag.

Using Methods of the TimerControl Interface

Once you have declares and configured a Timer control, you can invoke its control's methods from within your web service's methods and implement handlers for the Timer control's callbacks.

Brief descriptions of each method are provided below. For complete information on each method, see TimerControl Interface.

Basic Timer Control Methods

The methods of the TimerControl interface that are most commonly called are:

- start: starts timer operation. The Timer control will fire after the period specified by the timeout attribute has passed.
- restart: resets the Timer control such that the next firing will occur after the period specified by the timeout attribute has passed.
- stop: stops the Timer control from firing again until start or restart has been called.

Timer Control Configuration Methods

In addition to the methods listed above, the TimerControl interface also defines the following methods that may be used to configure a Timer control. Each method sets or gets the value of the attribute named in the method name.

- setTimeoutAt
- getTimeoutAt
- setTimeout
- getTimeout
- setRepeatsEvery
- getRepeatsEvery
- setCoalesceEvents
- getCoalesceEvents

Handling Timer Control Callbacks

The Timer control defines one callback:

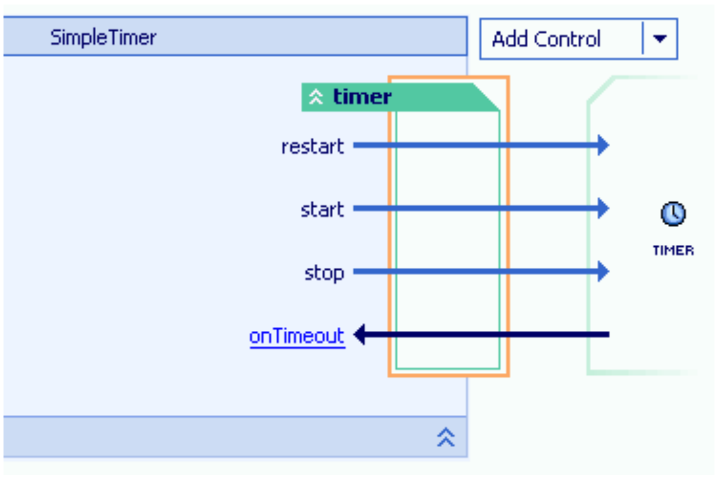
- onTimeout

The purpose of a Timer control is to notify a web service when a specified time period has elapsed or an absolute time has been reached. The Timer control delivers this notification using its onTimeout callback. You receive the callback by implementing a callback handler named timerName_onTimeout, where timerName is the name of the Timer control instance.

The callback has a single parameter that is the time at which the callback was scheduled. Note that this is not the same as the time at which the callback handler executes. A delay may occur between Timer control expiration and callback handler invocation due to system load.

Creating a Timer Control Callback Handler Automatically

WebLogic Workshop will create the skeleton of a Timer control's callback handler for you. To create a skeleton callback handler for a Timer control's onTimeout callback, click the onTimeout link associated with the Timer control in Design View, as shown here:



If a callback handler does not exist for the selected callback, WebLogic Workshop creates one, switches to Source View, and places the cursor in the callback handler for the selected callback.

In the example above, the callback handler will be named delayTimer_onTimeout.

Related Topics

[Creating a New Timer Control]

[Add Timer Control Dialog]

[Specifying Time on a Timer Control]

[TimerControl Interface]

How Do I: Configure a Timer Control?

You configure a Timer control in the following ways:

- By setting the attributes of the @jws:timer tag associated with the Timer control's declaration in your JWS file.
- By calling methods of the TimerControl interface on the instance of TimerControl you wish to configure.

Both of these techniques are described in Using a Timer Control.

Samples

The following sample web services demonstrate use of a Timer control:

SimpleTimer.jws Sample

AdvancedTimer.jws Sample

Related Topics

Controls: Using Resources from a Web Service

Timer Control: Using Timers in Your Web Service

Specifying Time on a Timer Control

TimerControl Interface

Specifying Time on a Timer Control

This topic describes how to specify relative and absolute time values for a Timer control.

To learn more about controls, see Controls: Using Resources from a Web Service.

To learn about the Timer control, see Timer Control: Using Timers in Your Web Service.

To learn about configuring and using a Timer control, see Using a Timer Control.

Specifying Relative Time

You need to specify relative time periods whenever you are:

- setting the values of the timeout or repeats−every attributes of the @jws:timer tag.
- calling the setTimeout or setRepeatsEvery methods of the TimerControl interface.

When expressed as a text string, the format of relative time specifications is:

- Integers followed by time units (which are listed below), optionally truncated, case insensitive, and optionally separated by spaces.
- the string "p" (case insensitive) is allowed at the beginning; if it is present, then single−letter abbreviations and no spaces must be used and parts must appear in the order y m d h m s.

For example, the following code smaple is a valid duration specification that exercises all the time units, spelled out fully:

```
/**
 * @jws:control
 * @jws:timer timeout="99 years 11 months 13 days 23 hours 43 minutes 51 seconds"
 */
Timer almostCentury;
```

This example creates a Timer control whose default initial firing will occur in almost 100 years.

Units may be truncated, for example: valid truncations of "months" are "month", "mont", "mon", "mo", and "m". If both months and minutes are specified, use long enough abbreviations to be unambiguous.

The following Timer control declaration is equivalent to the previous example, but uses the fully truncated form:

```
/*
 * @control
 * @timer timeout="P99Y11M23H43M51S"
 */
Timer almostCentury;
```

Durations are computed according to Gregorian calendar rules, so if today is the 17th of the month, 3 months from now is also the 17th of the month. If the target month is shorter and doesn't have a corresponding day (for example, no February 31), then the closest day in the same month is used (for example, February 29 on a leap year).

Specifying Absolute Time

You can configure a Timer control to fire at an absolute time by calling the setTimeoutAt methods of the TimerControl interface.

The setTimeoutAt method configures the timer to fire an event as soon as possible on or after the supplied absolute time. If you supply an absolute time in the past, the timer will fire as soon as possible.

If setTimeoutAt is called within a transaction, its effect (any work performed in the callback handler) is rolled back if the transaction is rolled back, and its effect is committed only when the transaction is committed.

If setTimeoutAt is called while the timer is already running, it will have no effect until the timer is stopped and restarted.

The setTimeoutAt method takes as its argument a java.util.Date object. Please see the documentation for the java.util.Date class to learn how to manipulate Date objects. Other Java classes that are useful when dealing with Date are java.util.GregorianCalendar and java.text.SimpleDateFormat.

Related Topics

Timer Control: Using Timers in Your Web Service

Database Control: Using a Database from Your Web Service

A Database control provides simplified access to a relational database, allowing a web service to call Java methods and operate on Java objects that are appropriate to the operations being performed. The Database control automatically performs the translation from Java objects to database queries and vice versa.

Each Database control is customized to access a particular database and perform particular operations on that database.

A Database control can operate on any database for which an appropriate JDBC (Java Database Connectivity) driver is available and for which a data source is configured in WebLogic Server.

As with all controls (except Timer controls), a Database control is defined in a CTRL file.

All Database controls extend the weblogic.jws.control.DatabaseControl base class.

To learn about WebLogic Workshop controls, see Controls: Using Resources from a Web Service.

Using an Existing Database Control

To learn how to use an existing Database control, see Using an Existing Database Control.

Creating a New Database Control

To learn how to create a new Database control, see Creating a New Database Control.

Database Control Samples

CustomerDBClient.jws Sample

LuckyNumberDBClient.jws Sample

Related Topics

Controls: Using Resources from a Web Service

WebLogic Workshop Projects

CTRL File: Implementing Controls

Creating a New Database Control

A Database control provides a Java interface to database operations, making those operations very simple to access from the Java code in a web service. A specific instance of the Database control must be created that includes the particular operations desired. This customized Database control is defined in a CTRL file.

This topic discusses the mechanics of creating a Database control.

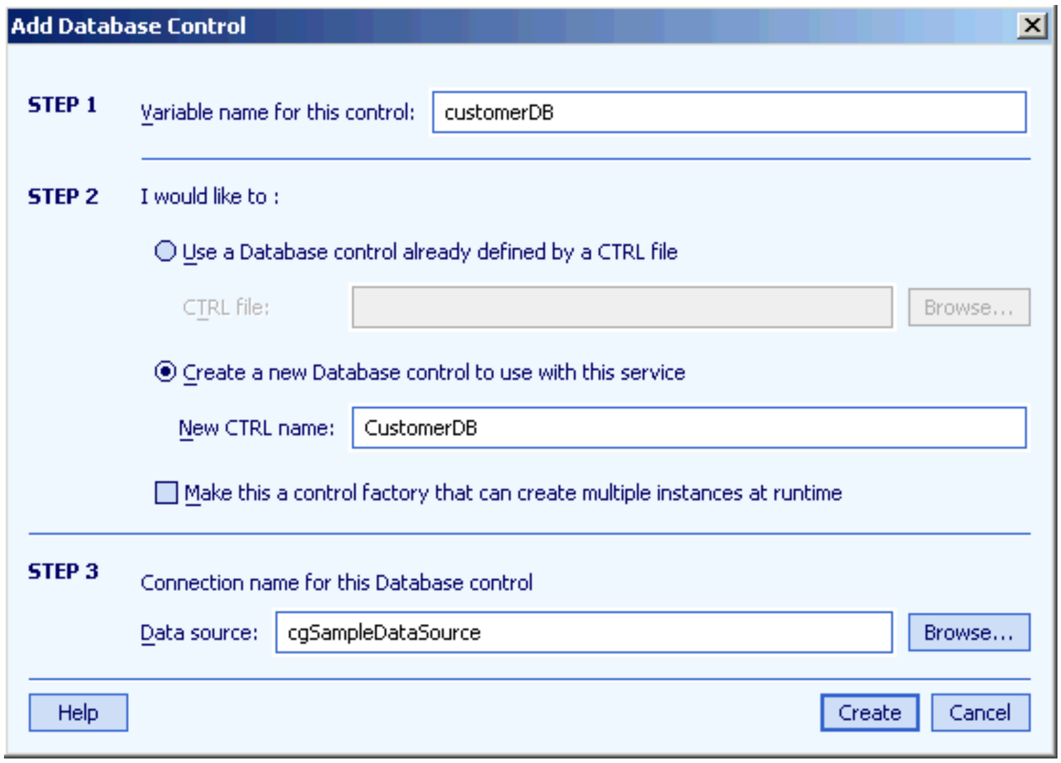
To learn about controls, see Controls: Using Resources from a Web Service.

To learn about the Database control, see Database Control: Using a Database from Your Web Service.

To learn about the issues you should consider when designing a Database control, see Database Control Design Issues.

Creating a Database Control

You can create a new Database control and add it to your web service by using the Add Database Control Dialog. The Add Database Control Dialog can be accessed via the Service menu on the Menu Bar; via the right−button context menu on the service in the Design View; or via the Add Control option menu in the upper right corner of Design View.


Add Database Control

STEP 1 Variable name for this control: customerDB

STEP 2 I would like to :

○ Use a Database control already defined by a CTRL file

CTRL file:

● Create a new Database control to use with this service

New CTRL name: CustomerDB

☐ Make this a control factory that can create multiple instances at runtime

STEP 3 Connection name for this Database control

Data source: cgSampleDataSource

For an explanation of the steps in the Add Database Control dialog, see Add Database Control Dialog.

Alternatively, you may create a Database control CTRL file manually. For example, you may copy an existing Database control CTRL file and modify the copy.

To learn how to add a method to a Database control, see Adding a Method to a Database Control.

The Template Database Control CTRL File

If you create a new Database control in WebLogic Workshop, the new CTRL file will look like this:

```
import weblogic.jws.*;
import weblogic.jws.control.*;
import java.sql.SQLException;
/**
 * Defines a new database control.
 *
 * The @jws:connection tag indicates which WebLogic data source will be used
 * by this database control. Please change this to suit your needs. You can
 * see a list of available data sources by going to the WebLogic console
 * in a browser (typically http://localhost:7001/console) and clicking
 * Services, JDBC, Data Sources.
 *
 * @jws:connection data-source-jndi-name="cgSampleDataSource"
 */
public interface CustomerDBControl extends DatabaseControl
{
    // Sample database function. Uncomment to use
    // static public class Customer
    // {
    //   public int id;
    //   public String name;
    // }
    //
    // /**
    //  * @jws:sql statement="SELECT ID, NAME FROM CUSTOMERS WHERE ID = {id}"
    //  */
    // Customer findCustomer(int id) throws SQLException, ControlException;
}
```

The name that was specified for this Database control is CustomerDBControl. It is customary for CTRL files to have names that end with "Control". Whenever WebLogic Workshop creates a CTRL file, it appends "Control" to the end of the name you specify. In Java, the main class or interface in a file must have the same name as the file. So the name of the interface in this case is CustomerDBControl.

There are three characteristics of this CTRL file that pertain to Database controls:

- The main interface in the file extends weblogic.jws.control.DatabaseControl.
- The Javadoc comment on the main interface contains a @jws:connection tag to indicate how to connect to the database.
- The file contains methods that are annotated with @jws:sql tags that define the database operations available via this control.

To learn how to add a method to a Database control, see Adding a Method to a Database Control.

Transactions and the Database Control

Database operations occur within the context of an implicit transaction that wraps each web service method invocation. To learn more about WebLogic Workshop's default transaction semantics, see Transactions in WebLogic Workshop.

The Database Connection: The @jws:connection Tag

Before you can perform operations on a database, you must have a connection to the database. The Database control will handle all of the details of managing the database connection, but you must supply the name of a data source that has been configured with the information necessary to access a database.

A default data source called cgSampleDataSource is configured when WebLogic Workshop is installed; it uses the PointBase database.

To learn how to create, configure and register a data source, see How Do I: Connect a Database Control to a Database Such as SQL Server or Oracle.

Once the data source is configured and registered in the JNDI registry, the data source name may be used in the data−source−jndi−name attribute of the @jws:connection tag.

For detailed information on the @jws:connection tag, see @jws:connection Tag.

Related Topics

Controls Overview

Database Control: Using a Database from Your Web Service

How Do I: Connect a Database Control to a Database Such as SQL Server or Oracle

Adding a Method to a Database Control

Using an Existing Database Control

CustomerDBClient.jws Sample

LuckyNumberDBClient.jws Sample

DatabaseControl Interface

@jws:sql Tag

@jws:connection Tag

Adding a Method to a Database Control

This topic discusses the mechanics of adding a method to a Database control.

To learn about controls, see Controls: Using Resources from a Web Service.

To learn about Database controls, see Database Control: Using a Database from Your Web Service.

To lean how to create a Database control, see Creating a New Database Control.

To learn about issues you should consider when designing a Database control, see Database Control Design Issues.

Adding a Method to a Database Control

The Database Control Method Pattern

A method in a Database control always associates a Java method definition with a SQL database operation. The SQL statement that defines the database operation may optionally contain substitutions from the parameter list of the Java method. The return type of the database operation is determined by the return type of the Java method.

The example Database control method below illustrates the pattern used to define Database control methods.

```
/**
 * @jws:sql statement="UPDATE customer SET address = {customerAddress} WHERE custid={customerID
 */
public int changeAddress(int customerID, String customerAddress);
```

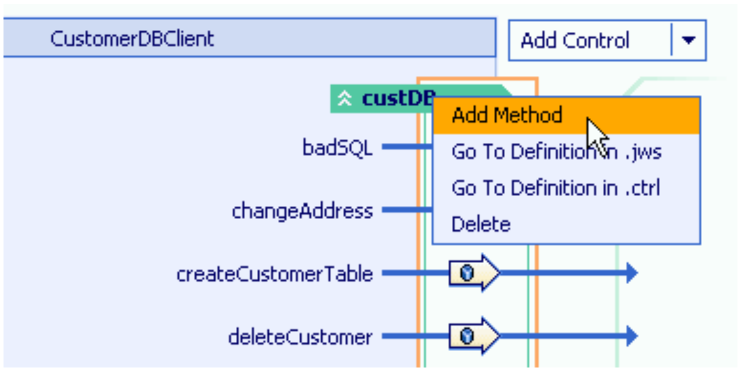
The method signature declares a method that a user of this control may invoke. You should design this method such that its arguments and return value are convenient and useful to developers of web services that will use this control.

The @jws:sql tag describes the SQL statement that is associated with the method. The SQL statement may include substitution parameters that reference the parameter names in the method signature. In the example above, the SQL statement includes the substitutions {customerAddress} and {customerID}. These are a references to the customerID and customerAddress parameters of the findCustomer method. When the method is invoked, the values of any referenced parameters are substituted in the SQL statement before it is executed. Note that parameter substitution is case−sensitive; parameters mentioned in substitutions must exactly match the spelling and case of the parameters to the method.

For detailed information on the @jws:sql tag, see @jws:sql Tag.

Adding a Method in Design View

If you are editing a web service in Design View and you wish to add a method to a Database control your web service is using, you may add the method directly by right−clicking on the Database control instance in Design View, as shown here:



In the example above, the CustomerDBClient web service is using a Database control whose instance name is custDB. Right−clicking on the custDB object will present a menu that contains the Add Method action. Select the Add Method action to add a method to the Database control of which custDB is an instance.

Note: The Database control variable declared in a web service's JWS file is an instance of a Database control. The actual control is defined in a CTRL file. When you add a method to a Database control from Design View, the method is added to the CTRL file. The method will subsequently be available in all web services that are using that Database control.

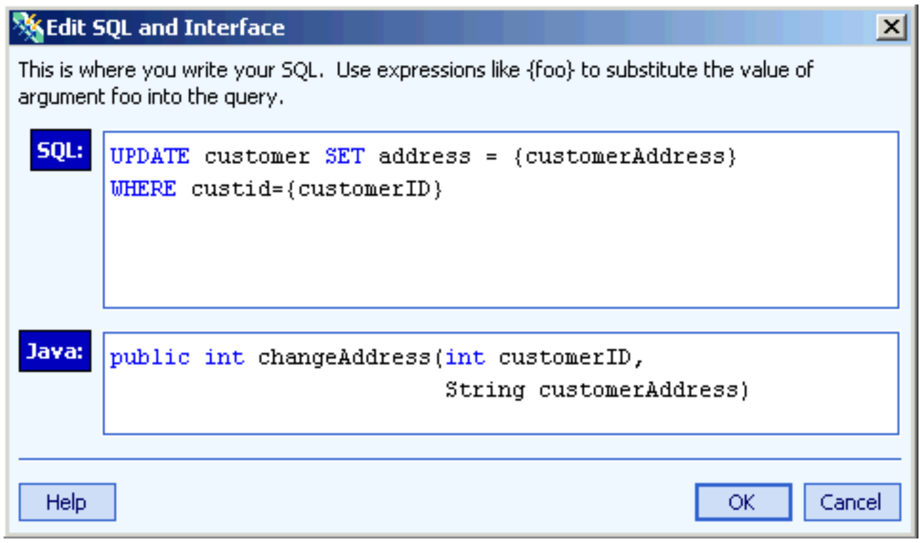
When a method is added, it will initially have no parameters and no associated SQL statement. To edit the parameter list and the associated SQL statement, see The Edit SQL and Interface Dialog section below.

The Edit SQL and Interface Dialog

The Edit SQL and Interface Dialog allows you to customize the parameter list and associated SQL statement for a Database control method. To display the Edit SQL and Interface Dialog, double−click on the map icon associated with the method you wish to customize.

The map icon for a Database control method is shown here:

The Edit SQL and Interface dialog is shown here:



This is where you write your SQL. Use expressions like {foo} to substitute the value of argument foo into the query.

```
UPDATE customer SET address = {customerAddress}
WHERE custid={customerID}
```

```
public int changeAddress(int customerID,
                         String customerAddress)
```

You may edit all aspects of the Database control method in the Edit SQL and Interface Dialog. The Java signature that is available to web services that use this Database control is defined in the lower portion. The SQL statement that is executed whenever the Java method is invoked is defined in the upper portion.

The rules of parameter substitution in Database control method SQL statements are described in Parameter Substitution in @jws:sql Statements.

The various types that may be returned by a Database control method are defined in Return Values of Database Control Methods, below.

Return Values of Database Control Methods

Database control methods may return a single value, a single row, or multiple rows.

Returning a Single Value

Some database operations return a single value. Example of such operations include INSERT and UPDATE operations, which return the number of rows affected by the INSERT or UPDATE. Another example is a SELECT operation that returns a single column of a single row.

To learn how to write Database control methods that return a single value, see Returning a Single Value from a Database Control Method.

Returning a Single Row

Some database operations return multiple columns from a single row. An example of such an operation is a SELECT operation the returns all or a subset of columns from a single row.

To learn how to write Database control methods that return a single row, see Returning a Single Row from a Database Control Method.

Returning Multiple Rows

Some database operations return one or more columns from multiple rows. An example of such an operation is a SELECT operation that returns one or more columns from multiple rows.

To learn how to write Database control methods that return multiple rows, see Returning Multiple Rows from a Database Control Method.

Related Topics

Controls: Using Resources from a Web Service

Database Control: Using a Database from Your Web Service

Creating a New Database Control

Using an Existing Database Control

CustomerDBClient.jws Sample

LuckyNumberDBClient.jws Sample

DatabaseControl Interface

Returning a Single Value from a Database Control Method

This topic discusses methods that you can add to Database control that return a single value from the database, including updates, inserts and single−valued select statements.

To learn about Database controls, see Database Control: Using a Database from Your Web Service.

To learn about creating a Database control, see Creating a New Database Control.

Returning a Single Value from a Database Control Method

Examples of database operations that return a single value are INSERT and UPDATE operations, which return the number of rows affected; or SELECT statements that request only a single column of a single row. In these cases, the return value of the method should be an object or primitive of the appropriate type.

Example 1

In the following example, an UPDATE operation is performed and the number of rows affected is returned:

```
/**
 * @jws:sql statement="UPDATE customer SET address={customerAddr} WHERE custid={customerID}"
 */
public int setCustomerAddress(int customerID, String customerAddr);
```

This example updates the customer table. For each record in the table in which the custid field matches the value of the customerID parameter, the address field is set to the value of the customerAddr parameter.

Example 2

In the following example a single column of a single row is returned (assuming custid is the primary key for the customer table). The field type is VARCHAR, so the return value is declared as String.

```
/**
 * @jws:sql statement="SELECT name FROM customer WHERE custid={customerID}"
 */
public String getCustomerName(int customerID);
```

To learn about the relationship between database types and Java types, see Mapping Database Field Types to Java Types in the Database Control.

Related Topics

Database Control: Using a Database from Your Web Service

Creating a New Database Control

Parameter Substitution in @jws:sql Statements

Returning a Single Row from a Database Control Method

Returning Multiple Rows from a Database Control Method

Returning a Single Row from a Database Control Method

This topic discusses methods that you can add to a Database control that return a single row from the database.

To learn about Database controls, see Database Control: Using a Database from Your Web Service.

To learn about creating a Database control, see Creating a New Database Control.

To learn about parameter substitution in @jws:sql statements, see Parameter Substitution in @jws:sql Statements.

Returning a Single Row from a Database Control Method

If the database operation returns multiple columns, you have a choice of return values for the method. You may return an object that meets specific criteria, or you may return a java.util.HashMap.

Returning an Object

To return an object of a particular class, you must declare the class with specific characteristics:

- The class must contain members with names that match the names of the columns that will be returned by the query. The matching is case−insensitive, since database column names are case−insensitive. The class may contain other members, but must include at least the members corresponding to the columns that will be returned.
- The members must be of an appropriate type to hold a value from the corresponding column in the database. For information on mapping between database types and Java types, see Mapping Database Field Types to Java Types in the Database Control.
- If the class is an inner class (a class declared inside another class), the class must be declared public static.

Since this class exists in a tight relationship with the Database control, the class is typically declared as an inner class of the main interface in the Database control's CTRL file. However, it may be any Java class that meets the above criteria.

The example below illustrates returning an object from a Database control method:

```
public static class Customer
{
    public int custid;
    public String name;
    public Customer() {};
}
/**
 * @sql statement="SELECT custid,name FROM customer WHERE custid={customerID}"
 */
Customer findCustomer(int customerID)
```

Note: Class member variables and the class itself must be public in order for the Database control to substitute them.

If the corresponding database field does not contain a value, the class member will be set to null if it is an object and 0 or false if it is a primitive. This may affect your decisions regarding the types you use in your class. If the database field contained no data, an Integer member would receive the value null, but an int member would receive the value 0. Zero may be a valid value, thus use of int instead of Integer makes it impossible for subsequent code to determine whether a value was present in the database.

If there is no column in the database corresponding to a member of the class, that member will also be set to null or 0, depending on whether the member is an primitive or an object.

If the query returns columns that cannot be matched to the members of the class, an exception is thrown. If you don't know the columns that will be returned or they may change, you should consider returning a HashMap instead of a specific class.

If no rows are returned by the query, the returned value of the Database control method is null.

In this example, the method is declared as returning a single object of type Customer. As such, even if the database operation returns multiple rows, only the first row is returned to the method's caller. To learn how to return multiple rows to the caller, see Returning Multiple Rows from a Database Control Method.

Returning a HashMap

You may choose to return a HashMap if the number of columns or the particular column names returned by the query are unknown or may change.

To return a HashMap, declare the return value of the method as java.util.HashMap.

```
/**
 * @jws:sql statement="SELECT * FROM customer WHERE custid={custID}"
 */
public java.util.HashMap findCustomerHash(int custID);
```

The HashMap returned will contain an entry for each column in the result. The key for each entry is the corresponding column name. Keys are case−insensitive when accessed via the HashMap's methods (the capitalization of the key names returned by HashMap.keySet() depends on the database driver in use). The value is an object of the JDBC default type for the database column.

For information on mapping between database types, Java types and JDBC types, see Mapping Database Field Types to Java Types in the Database Control.

In this example, the method is declared as returning a single object of type java.util.HashMap. As such, even if the database operation returns multiple rows, only the first row is returned to the method's caller.

To learn how return multiple rows to the caller, see Returning Multiple Rows from a Database Control Method

To access the name field of the returned record in the calling web service (JWS file), you could use the following code:

```
/**
 * @jws:control
 */
private CustomerDBControl custDB;



/**
 * @jws:operation
 */
public String getCustomerName(int custID)
{
    java.util.HashMap hash;
    String name;
    hash = custDB.findCustomerHash(custID);
    if( hash != null )
    {
        name = (String)hash.get("NAME");
    }
    else
    {
        name = new String("Customer not found");
    }
    return name;
}
```

If no rows are returned by the query, the returned value of the Database control method is null.

Related Topics

[Parameter Substitution in @jws:sql Statements]()

[Mapping Database Field Types to Java Types in the Database Control]()

Returning Multiple Rows from a Database Control Method

This topic discusses methods that you can add to a Database control that will return multiple rows in from the database.

To learn about Database controls, see [Database Control: Using a Database from Your Web Service]().

To learn about creating a Database control, see [Creating a New Database Control]().

Returning Multiple Rows from a Database Control Method

Some database operations return one or more columns from multiple rows. An example of such an operation is a SELECT operation that returns one or more columns from multiple rows.

There are several ways to return multiple rows from a database operation. The techniques for returning multiple rows are analogous to those described above for returning a single row. See [Returning a Single Row from a Database Control Method]().

The first choice is whether to return an array of objects or a java.util.Iterator.

An array of objects may be more convenient for the users of your control since they won't have to understand how to use Iterators. However, when an array is returned only one database operations is performed and the entire result set must be stored in memory. For large result sets, this is problematic. You can limit the size of the returned array, but then you cannot provide a way for your user to get the remainder of the result set.

To learn how to return an array of objects, see [Returning an Array of Objects]().

While Iterators require more sophistication on the part of users of your control, they are more efficient at handling large result sets. An Iterator is accessed one element (row) at a time (via the Iterator's next() method), and it will transparently make repeated requests from the database until all records have been processed. An Iterator does not present the risk of running out of memory that an array presents.

To learn about returning a java.util.Iterator, see [Returning an Iterator]().

Finally, you may choose to return a java.sql.ResultSet from a Database control method. This grants complete access to the results of the database operation to clients of your control, but it requires knowledge of the java.sql package.

To learn about returning a java.sql.ResultSet, see Returning a Result Set.

Returning an Array of Objects

When you want to return an array of objects, you declare the method's return type to be an array of the object you want to return. That type may be either a type you define, or it may be java.util.Hashmap.

Examples of these techniques are given in the following sections.

Returning an Array of User−defined Objects

The following example demonstrates how to return an array of objects whose type you have declared. In this case, an array of Customer objects is returned.

```
public static class Customer
{
    public int custid;
    public String name;
}



/**
 * @sql statement="SELECT custid,name FROM customer WHERE custage<19"
 *      array−max−length=100
 */
Customer [] findAllMinorCustomers()
```

This example will return all rows in which the custage field contains a value less than 19.

When returning an array of objects, the class declared as the return type of the method must meet the criteria described in Returning an Object.

If no rows are returned by the query, the returned value of the Database control method is a zero−length array.

When an array is returned by a Database control method, the @jws:sql tag may have the array−max−length attribute. This attribute can protect you from very large result sets that may be returned by very general queries. If array−max−length is present, no more than that many rows will be returned by the method.

The default value of array−max−length is 1024.

The array−max−length attribute may have the special value "all" (in quotes), indicating that all rows satisfying the query should be returned. Note that if the query is too general, this can result in use of all available memory. To avoid excessive memory usage, return an Iterator as described below in Returning an Iterator.

Returning an Array of HashMaps

Returning an array of HashMaps is analogous to returning an array of user−defined objects, which is described in the preceding section.

The following example demonstrates returning an array of HashMaps.

```
public static class Customer
{
    public int custid;
    public String name;
    public Customer() {};
}



/**
 * @sql statement="SELECT custid,name FROM customer WHERE custage<19"
 *          array-max-length=100
 */
java.util.HashMap [] findAllMinorCustomersHash()
```

The array of HashMaps returned contains an element for each row returned, and each element of the array contains an entry for each column in the result. The key for each entry is the corresponding column name. Keys are case−insensitive when accessed via the HashMaps methods (the capitalization of the key names returned by HashMap.keySet() depends on the database driver in use). The value is an object of the JDBC default type for the database column.

If no rows are returned by the query, the returned value of the Database control method is a zero−length array.

For information on mapping between Java types and JDBC types (database types), see Mapping Database Field Types to Java Types in the Database Control.

To access the name field of the returned records in the calling web service (JWS file), you could use the following code:

```
/**
 * @jws:control
 */
private CustomerDBControl custDB;





int custID = <some customer ID>
java.util.HashMap [] hashArr;
String name;



hashArr = custDB.findAllMinorCustomersHash();
for(i=0; i<hashArr.length; i++)
{
    name = (String)hashArr[i].get("NAME");
    // say hello to the all of the minors



    System.out.println("Hello, " + name + "!");
}
```

Returning an Iterator

When you want to return an Iterator, you declare the method's return type to be java.util.Iterator. You then add the iterator−element−type attribute to the @jws:sql tag to indicate the underlying type that the Iterator will contain. The specified type may be either a type you define, or it may be java.util.Hashmap. Examples of these techniques are given in the following sections.

Note: The Iterator that is returned is only guaranteed to be valid for the life of the JWS method call to which it is returned. You should not store an Iterator returned from a Database control method as a static member of your web service's class, nor should you attempt to reuse the Iterator in subsequent method calls if it is persisted by other means.

Returning an Iterator with a User−defined Object

To return an Iterator that encapsulates a user−defined type, provide the class name as the value of the iterator−element−type attribute of the @jws:sql tag.

```
public static class Customer
{
    public int custid;
    public String name;
    public Customer() {};
}



/**
 * @sql statement="SELECT custid,name FROM customer"
 *      iterator−element−type="Customer"
 */
java.util.Iterator getAllCustomersIterator()
```

The class specified in the iterator−element−type attribute must meet the criteria described in Returning an Object.

To access the returned records in the calling web service (JWS file), you could use the following code:

```
CustomerDBControl.Customer cust;



java.util.Iterator iter = null;



iter = custDB.getAllCustomersIterator();



while (iter.hasNext())
{
    cust = (CustomerDBControl.Customer)iter.next();
    // say hello to every customer



    System.out.println(“hello, “ + cust.name + “!”);
}
```

Returning an Iterator with HashMap

To return an Iterator that encapsulates a HashMap, provide java.util.HashMap as the value of the iterator−element−type attribute of the @jws:sql tag.

```
public static class Customer
{
    public int custid;
    public String name;
    public Customer() {};
}



/**
 * @sql statement="SELECT custid,name FROM customer"
 *      iterator-element-type="java.util.HashMap"
 */
java.util.Iterator getAllCustomersIterator()
```

To access the returned records in the calling web service (JWS file), you can use the following code:

```
java.util.HashMap custHash;
java.util.Iterator iter = null;



int customerID;



String customerName;



iter = custDB.getAllCustomersIterator();



while (iter.hasNext())
{
    custHash = (java.util.HashMap)iter.next();
    customerID = (int)custHash.get("CUSTID");



    customerName = (String)custHash.get("NAME");
}
```

The HashMap contains an entry for each database column that is returned by the query. The key for each entry is the corresponding column name, in all uppercase. The value is an object of the JDBC default type for the database column.

For information on mapping between Java types and JDBC types (database types), see Mapping Database Field Types to Java Types in the Database Control.

Returning a ResultSet

You may gain complete access to the java.sql.ResultSet returned by a query. This is an advanced use. The Database control is designed to allow you to obtain data from a database in a variety of ways without having to understand the classes in the java.sql package.

If you want to return a ResultSet, you declare the method's return type to be java.sql.ResultSet. A client of your control then accesses the ResultSet directly to process the results of the database operation.

The following example demonstrates returning a ResultSet.

```
/**
 * @jws:sql statement="SELECT * FROM customer"
 */
public java.sql.ResultSet findAllCustomersResultSet();
```

To access the returned ResultSet in the calling web service (JWS file), you could use the following code:

```
java.sql.ResultSet resultSet;
String thisCustomerName;
resultSet = custDB.findAllCustomersResultSet();
while (resultSet.next())
{
    thisCustomerName = new String(resultSet.getString("name"));
}
```

This example assumes the rows returned from the database operation include a column named name.

Related Topics

Parameter Substitution in @jws:sql Statements

Parameter Substitution in @jws:sql Statements

This topic discusses parameter substitution in the @jws:sql tags associated with methods of a Database control.

To learn about Database controls, see Database Control: Using a Database from Your Web Service.

To learn about creating a Database control, see Creating a New Database Control.

Substitution Criteria

Substitution is subject to the following criteria:

- Substitution matching is case sensitive. The method parameter CustCity will not match the substitution pattern {custCity}.
- The type of the method parameter must be compatible with the type of the associated database field in the statement. If you attempt to substitute a Java String where the database expects a NUMBER, the statement will fail. For information on mapping between database types and Java types, see Mapping Database Field Types to Java Types in the Database Control.
- Substitution will not occur if the substitution pattern contains spaces, as in {custCity } or { custCity}. The Java Database Connectivity API, JDBC, allows access to built−in database functions via escapes of the form {fn user()}. If spaces occur in a {} item, the Database control will treat the item as a JDBC escape and will pass it on without substitution. For more information on JDBC escapes, please consult the documentation for your JDBC driver.
- When substituting date or time values, use the classes in the java.sql package. For example, attempting to substitute java.util.Date in a SQL Date field will not work. Use java.sql.Date instead.

Substituting Simple Parameters

If the parameters of a Database control method are primitives or simple types, you may refer to them directly in a {} substitution in the string value of the @jws:sql tag's statement parameter.

The following example illustrates simple parameter substitution:

```
/**
 * @jws:sql statement="SELECT name FROM customer WHERE city={custCity} AND state={custState}"
 */
public String [] getCustomersInCity( String custCity, String custState );
```

The value of the custCity method parameter is substituted in the query in place of the {custCity} item, and the value of the custState method parameter is substituted in the query in place of the {custState} item.

Substitutions are subject to the criteria described in Substitution Criteria.

Treatment of Curly Braces Within Literals

Curly braces "{}" within literals (strings within quotes) are ignored. This means statements like the following will not work as you might expect:

```
/**
 * @jws:sql statement::
 *     SELECT name
 *     FROM employees
 *     WHERE name LIKE '%{partialName}%'
 * ::
 */
public String[] partialNameSearch(String partialName);
```

Since the curly braces are ignored inside the literal string, the expected substitution of the partialName Java String into the SELECT statement does not occur. To avoid this problem, pre−format the match string in the JWS file before invoking the Database control method, as shown below:

```
String partialNameToMatch = "'%" + matchString + "%'"
String [] names = myDBControl.partialNameSeach(partialNameToMatch);
```

Substituting Indirect Parameters

Assume the following class is declared and is accessible to the Database control.

```
public static class Customer
{
    public String firstName;
    public String lastName;
    public String streetAddress;
    public String city;
    private String state;
    public String zipCode;



    public String getState() {return state};
}
```

You can then refer to the members of a Customer parameter in the {} substitutions, as shown in the following example:

```
/**
 * @jws:sql statement="SELECT name FROM customer WHERE city={cust.city} AND state={cust.state}"
 */
public String [] getCustomersInCity( Customer cust );
```

Note: Class member variables and accessor (getXxx) methods must be public in order for the Database control to substitute them.

The dot notation is used to access the members of the parameter object.

Given the substitution pattern {myClass.myMember}, the precedence for resolving dot notations in substitutions is:

- If class myClass exposes public getMyMember() and setMyMember() methods, getMyMember() will be called and the return value will be substituted. For boolean variables, substitute isMyMember() for getMyMemnber().
- Else if class myClass exposes a public field named myMember, myClass.myMember will be substituted.
- Lastly if class myClass implements java.util.Map, myClass.get("myMember") will be called and the return value will be substituted.
- Any combination of these may exist, for example {A.B.C} where B is a public member of A and B has a public getC() method.

If none of these conditions exist, the Database control method will throw a weblogic.jws.control.ControlException.

Substitutions are subject to the criteria described in Substitution Criteria.

Related Topics

Mapping Database Field Types to Java Types in the Database Control

Returning a Single Value from a Database Control Method

Returning a Single Row from a Database Control Method

Returning Multiple Rows from a Database Control Method

.

Declaring Exceptions in Database Control Methods

The database operations that are associated with methods in a Database control may result in error conditions. You have a choice whether to declare that your Database control method throws exceptions.

You may optionally declare that a method in a Database control throws the java.sql.SQLException, as shown here:

```
/**
 * @jws:sql statement="DROP TABLE customer"
 */
public void dropCustomerTable() throws SQLException;
```

However, if you do so, you are forcing the user of your Database control to explicitly handle the exception by surrounding calls to this method in a try−catch block. For example, to call the dropCustomerTable method declared above on an control named custDB, your users will have to write code as follows:

```
try {
    custDB.dropCustomerTable()
}
catch (java.sql.SQLException sqle)
```

```
{
    // handle error here
}
```

If you do not declare that your methods throw exceptions, exceptions that are generated are still thrown. But they are wrapped as weblogic.jws.control.ControlException objects before being thrown. ControlException is a subclass of java.lang.RuntimeException. The advantage of RuntimeException is that it does not require that calls to methods that throw it to be wrapped in try−catch blocks. ControlException can still be caught by users of your Database control, but making it optional may increase convenience for users of your control.

If a ControlException (containing a SQLException) is thrown by your Database control method and is not handled in the calling JWS method, the calling JWS method will return the exception to its client. In most cases, this is not appropriate; the client code will most likely not be able to take action on the exception since the exception is related to the internal workings of the web service. It would be better design to handle the exception in the web service and return an appropriate error to the client if necessary.

To learn about handling exceptions in controls, see Handling Exceptions in Controls.

Related Topics

Database Control: Using a Database from Your Web Service

Creating a New Database Control

Database Control Design Issues

This topic describes design issues you should consider when creating new Database controls.

To learn about Database controls, see Database Control: Using a Database from Your Web Service.

Partition the Interface

When designing a new Database control, you may with to consider creation multiple Database controls for the same database. Interactions with a database typically fall into two or three categories:

- Administration operations: these typically include CREATE and DROP operations on database tables.
- Privileged operations: these typically include INSERT and possibly UPDATE operations. Many applications restrict which users or applications may modify the contents of the database.
- Retrieval operations: typically SELECT operations. Operations that do not modify the contents of the database are typically available to a wider class of users.

If users in your application can be partitioned into these groups, you may wish to create separate Database controls for each class of user. In this way, you can expose only the less privileged operations to web service developers whose services do not require administrative access.

Note that database administration activities are typically carried out as part of application deployment. In a production environment, there is not typically a requirement for a web service interface to administrative activities such as table creation. The Database control samples provided with WebLogic Workshop include table creation methods for convenience as samples, not as examples of good design.

Be Aware of Potentially Large Result Sets

When you design database operations in your Database control, be aware of the potential size of the result sets that might be generated by the operations. With large databases, it is easy to accidentally execute queries that return result sets that are larger than the available memory on the machine.

Here are some ways to protect against out−of−memory errors due to large result sets:

- Limit the number of columns returned from a query to only those required. Avoid "SELECT *" statements unless that are explicitly necessary.
- Perform filtering in the database. If you are only interested in a subset of the records that might be returned from a query, refine the query so that the database system will do the filtering. Database systems are optimized for this type of operation, and are also designed to perform filtering and sorting operations in a memory−efficient manner.
- Use the array−max−length attribute to limit array size. If your Database control method returns an array, you should explicitly provide the array−max−length attribute on the @jws:sql tag for the method. Set the value such that it is higher than the number of rows you expect to be returned, but not so large that you might run out of memory should that many rows be returned. Note that there is no way to subsequently return the "deleted" rows should the array size limit be reached by a query. There is also no way to set array−max−length using an API call.
  The default value of array−max−length is 1024.
- Use an Iterator. A Database control can return a java.util.Iterator instead of an array. The Iterator wraps a java.sql.ResultSet object that accesses the database efficiently. As the web service that obtained the Iterator (via a call to a Database control method) steps through the Iterator, the Iterator and ResultSet transparently make occasional requests to the database to obtain more data. Using this technique, you may enable processing of result sets that are larger than the available memory.

To learn more about writing Database control methods that return multiple rows, see Returning Multiple Rows from a Database Control Method.

Design for User Convenience

When designing a Database control, as with designing any control or web service, think carefully about the information needs of the users of your control. Choose operations and data structures that are convenient for the web service developer.

One way to accomplish this is to provide Java classes that represent the records or partial rows your Database control methods operate on. Consistent use of these classes in the Database control's interface will simplify use of the Database control.

Consciously Choose Between Objects and Primitives

As is noted in the topics describing how to return data from Database control methods, the value returned when a database field is NULL depends on the Java type to which the database field is converted. Java primitives such as int and float are not capable of reflecting a null value; they must always have a valid value. Thus, if a database field with no value is converted to an int, the value will be 0. If it is important to you or the clients of your Database control to differentiate between zero and null, you should use the Java wrapper classes for basic types. For example, use java.lang.Integer instead of int, and java.lang.Float instead of float.

Design for Reuse

Wherever possible, try to avoid building rigidity into your Database controls.

For example, avoid hard−coding value into queries. Parameterize your database operations as much as possible while maintaining convenience for users of your control. It is typically desirable to hard−code the table name because operations are typically table specific. It is typically not desirable to hard−code absolute dates or times into queries.

Related Topics

Creating a New Database Control

Using an Existing Database Control

This topic describes how to use an existing Database control in your web service.

To learn about controls, see the Controls Overview.

To learn about Database controls, see Database Control: Using a Database from Your Web Service.

To learn how to create a Database control, see Creating a New Database Control.

Using an Existing Database Control

All controls follow a consistent model. Therefore, most aspects of using an existing Database control are identical to using any other existing control.

To learn about using an existing control, see Using an Existing Control.

Customizing a Database Control

There are Javadoc tags and tag attributes that are specific to the Database control. If you choose to copy and customize an existing Database control, the tags and attributes you may wish to modify are:

- @jws:connection tag
  For more information, see @jws:connection Tag.
- @jws:sql tag
  For more information, see @jws:sql Tag.

DatabaseControl Interface

All Database controls are subclassed from the DatabaseControl interface. The interface defines methods that may be called on Database control instances from a web service.

See DatabaseControl Interface.

Related Topics

CTRL Files: Implementing Controls

Mapping Database Field Types to Java Types in the Database Control

This topic describes the relationships between database field types and Java types in the Database control. To learn about the Database control, see Database Control: Using a Database from Your Web Service.

Type mappings are given for the PointBase database, which is installed with WebLogic Server, and for Oracle 8i. If you are using a different database, please consult the documentation for the JDBC driver of your database software.

Materialization

The process of converting a value retrieved from a database to a concrete Java type is referred to as materializing the value. It is possible for a database element to be materialized into many types. For example, one could imagine an INTEGER database value being materialized into all the following primitive types and classes: int, Integer, BigDecimal, float, Float, double, Double, String and possibly others.

Each database vendor defines a specific set of materializations that are supported by their database systems. If you attempt to perform a materialization that is not allowed by the database system you are accessing, an exception will be thrown.

The type mappings shown in the tables below are the most direct mappings and are shown as examples. Please consult the documentation for your database system to determine all possible database type to Java type mappings in the database system you are using.

Type Mappings for PointBase

The following table lists the relationships between database types and Java types for the PointBase Version 4.1 database.

Java Data Types	JDBC Data Types	PointBase SQL Data Types (Version 4.1)
boolean	BIT	boolean
byte	TINYINT	smallint
short	SMALLINT	smallint
int	INTEGER	integer
long	BIGINT	numeric/decimal
double	FLOAT	float
float	REAL	real
double	DOUBLE	double
java.math.BigDecimal	NUMERIC	numeric
java.math.BigDecimal	DECIMAL	decimal
String	CHAR	char
String	VARCHAR	varchar
String	LONGVARCHAR	clob
java.sql.Date	DATE	date
java.sql.Time	TIME	time
java.sql.Timestamp	TIMESTAMP	timestamp
byte[]	BINARY	blob

byte[]	VARBINARY	blob
byte[]	LONGVARBINARYbyte[]	blob
java.sql.Blob	BLOB	blob
java.sql.Clob	CLOB	clob

Type Mappings for Oracle 8i

The following table lists the relationships between database types and Java types for the Oracle 8i database.

Java Data Types	JDBC Data Types	Oracle SQL Data Types (Version 8i)
boolean	BIT	NUMBER
byte	TINYINT	NUMBER
short	SMALLINT	NUMBER
int	INTEGER	NUMBER
long	BIGINT	NUMBER
double	FLOAT	NUMBER
float	REAL	NUMBER
double	DOUBLE	NUMBER
java.math.BigDecimal	NUMERIC	NUMBER
java.math.BigDecimal	DECIMAL	NUMBER
String	CHAR	CHAR
String	VARCHAR	VARCHAR2
String	LONGVARCHAR	LONG
java.sql.Date	DATE	DATE
java.sql.Time	TIME	DATE
java.sql.Timestamp	TIMESTAMP	DATE
byte[]	BINARY	NUMBER
byte[]	VARBINARY	RAW
byte[]	LONGVARBINARY	LONGRAW
java.sql.Blob	BLOB	BLOB
java.sql.Clob	CLOB	CLOB

Related Topics

Returning a Single Value from a Database Control Method

Returning a Single Row from a Database Control Method

Returning Multiple Rows from a Database Control Method

Parameter Substitution in @jws:sql Statements

Using an Existing Database Control

Service Control: Using Another Web Service

A Service control provides an interface to another web service, allowing your service to invoke the methods and handle the callbacks of the other service. The other web service can be one developed with WebLogic Workshop or any web service for which a WSDL file is available.

As with most controls, a Service control is defined in a CTRL file. A specific Service control CTRL file provides a way to communicate with a specific web service. The name and location of the web service are explicitly stated in the CTRL file.

All Service controls extend the weblogic.jws.control.ServiceControl base class.

To learn about WebLogic Workshop controls, see Controls: Using Resources from a Web Service.

Understanding Public Contracts

Web services define and expose what is called a public contract. The public contract is typically expressed in a WSDL file.

To learn about WSDL files, see WSDL Files.

The public contract describes the operations that the web service can perform and the format of the messages that should be sent to the service to access the operations and receive operation results. The contract is completely under the control of the author of the web service; it cannot be altered by a client of the web service.

The public contract for a web service developed with WebLogic Workshop is completely defined in the JWS file for the web service: it is the collection of all methods marked with the @jws:operation tag plus all members of the Callback interface. When you generate a WSDL file from a JWS file, the public contract is expressed according to the WSDL standard.

Service Controls are Proxies for Web Services

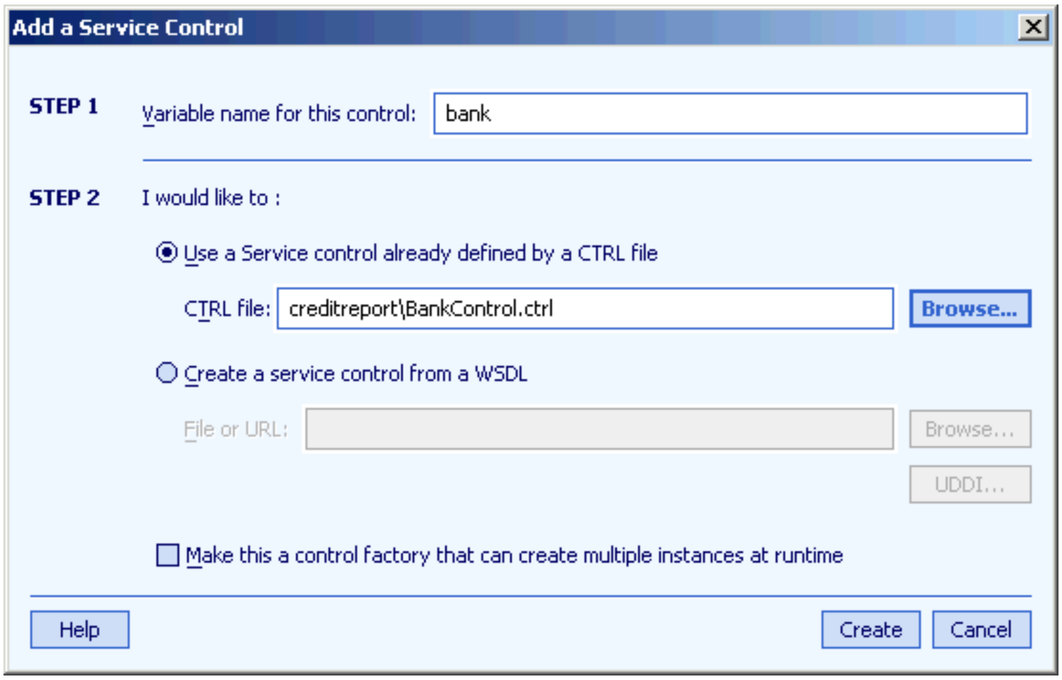
In WebLogic Workshop, a Service control serves as an intermediary, or proxy, for a web service. When web service A wants to invoke an operation of web service B, web service A calls a Java method of the Service control for B. The Service control converts the Java method invocation into an appropriate message to send to the web service B, and it communicates with web service B using a protocol that service B says it can understand. The Service control also converts returned messages from B back into Java method invocations on service A, known as callback handlers.

In this way, the Service control allows service A to use service B merely by implementing application−level Java code. As the author of service A, you do not need to know the details of message formats or protocols.

Using an Existing Service Control

If a Service control CTRL file already exists for another web service that you would like to use from your web service, you only need to add an instance of the Service control to your web service.

Us the Add Service Control Dialog to add a Service control to you web service. The Add Service Control Dialog can be accessed via the Service menu on the Menu Bar; via the right−button context menu on the service in the Design View; or via the Add Control option menu in the upper right corner of Design View.


Add a Service Control

STEP 1 Variable name for this control: bank

STEP 2 I would like to :

(●) Use a Service control already defined by a CTRL file

CTRL file: creditreport\BankControl.ctrl

( ) Create a service control from a WSDL

File or URL:

☐ Make this a control factory that can create multiple instances at runtime

The image above shows creation of a Service control instance named bank, which is an instance of the Service control defined in the BankControl.ctrl CTRL file.

Creating a New Service Control

To learn how to create a new Service control, see Creating a New Service Control.

Customizing Service Controls

There are a limited number of customizations you can make to a Service control's CTRL file to modify the default behavior of the Service control.

To learn about customizing Service controls, see Customizing a Service Control: Overview.

Service Control Samples

Samples that Use Service Controls

AdvancedTimer.jws Sample

Conversation.jws Sample

CreditReport.jws Sample

Investigate.jws Sample

Samples that Customize Service Controls

QuoteClient.jws Sample

Related Topics

WebLogic Workshop Projects

CTRL Files:Implementing Controls

ServiceControl Interface

Creating a New Service Control

A Service control provides an interface to another web service, allowing your service to invoke the methods and handle the callbacks of the other service. The other web service can be one developed with WebLogic Workshop or any web service for which a WSDL file is available. A Service control is defined in a CTRL file.

This topic discusses the mechanics of creating a Service control.

To learn about controls, see Controls: Using Resources from a Web Service.

To learn about the Service control, see Service Control: Using Another Web Service.

Creating a New Service Control

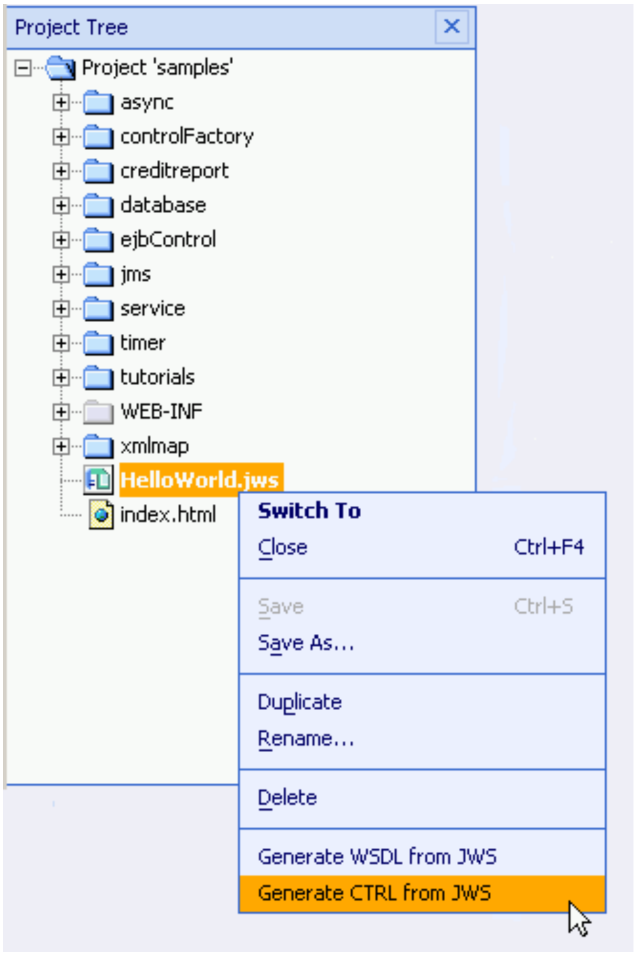
New Service controls are almost always created in order to enable your web service to use an existing web service. The other web service is referred to here as the target web service.

If the target web service was developed with WebLogic Workshop, you may create a Service control directly from the web service's JWS file. No matter how the target web service was developed, you can create a Service control for it if you have a WSDL file for the web service.

To Create a Service Control from a JWS File

If you have a WebLogic Workshop web service (a JWS file) for which you want to create a Service control, so that other WebLogic Workshop web services can use your web service:

1. Ensure that the JWS file for the web service is in your project.
2. Browse to the JWS file in the Project Tree.
3. Right−click on the JWS file in the Project Tree and select the action that generates a CTRL file from a JWS file, as shown here:



The resulting CTRL file is a Service control. It may be used by any other web service in the project, or copied to another project and used by web services there.

Note that WebLogic Workshop will add the word "Control" to the end of the name of generated controls. If you generate a CTRL file from the web service HelloWorld.jws, the CTRL file will be named HelloWorldControl.ctrl.

When you generate a CTRL file from a JWS file in this manner, the CTRL file is automatically linked to its parent JWS file. When you modify the web service, the CTRL file is automatically be regenerated to reflect the change. For example, if you add a method to the web service, an associated CTRL file is automatically regenerated, and the new method is available to every web service that employs that CTRL file as a Service control.

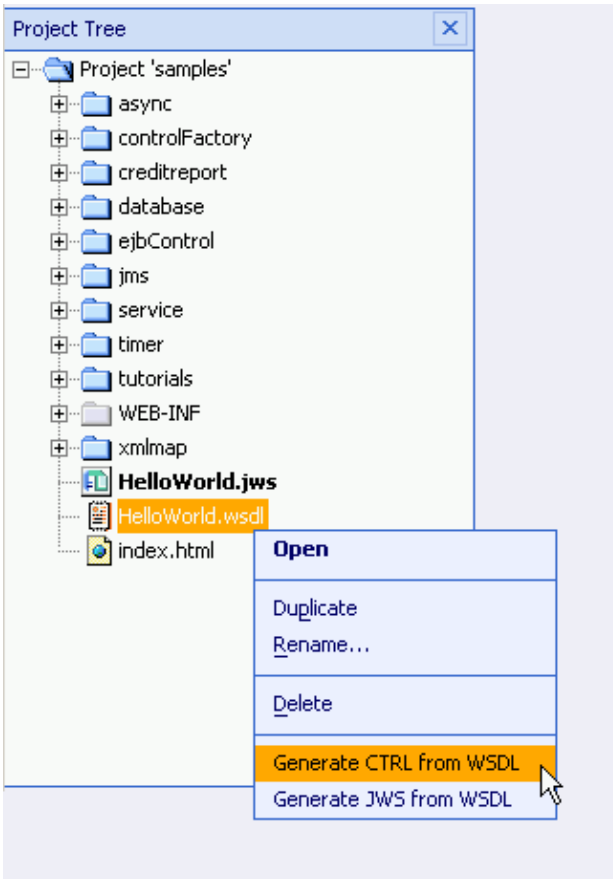
Autogenerated CTRL files in the Project Tree are marked with the icon shown below:

If you modify the source code for an autogenerated CTRL file, when you attempt to save the file WebLogic Workshop warns you that continuing the save operation will turn off autogeneration for this file. If you proceed, the CTRL file will no longer be linked to its parent web service. In general, you probably want to avoid turning off autogeneration for a CTRL file.

Creating a Service Control from a WSDL File

If you have a WSDL file for a web service that you would like to use from your WebLogic Workshop web service:

1. Ensure that the WSDL file for the web service is in your project.
2. Browse to the WSDL file in the Project Tree.
3. Right−click on the WSDL file in the Project Tree and select the action that generates a CTRL file from a WSDL file, as shown here:



The resulting CTRL file is a Service control. It may be used by any web service in the project, or copied to another project and used by web services there.

Note that WebLogic Workshop will add the word "Control" to the end of the name of generated controls. If you generate a CTRL file from the web service HelloWorld.jws, the CTRL file will be named HelloWorldControl.ctrl.

Related Topics

Customizing a Service Control: Overview

QuoteClient.jws Sample

Customizing a Service Control: Overview

This topic describes customizations you can make to a Service control's CTRL file to modify the control's default behavior.

To learn more about controls, see Controls: Using Resources from a Web Service.

To learn more about Service controls, see Service Control: Using Another Web Service.

Customizing a Service Control

There are a limited number of customizations you can make to a Service control's CTRL file. The customizations are described in the sections listed below.

The Service control for a web service cannot violate or modify the public contract of the web service it represents. This restricts the type of changes you can make to a Service control. For example, you can't modify the Service control to use a communication protocol that the target web service doesn't understand.

The topics below, which describe various aspects of customization of Service controls, discuss specific attributes whose values may be modified. These attributes may be set using the Properties pane in Design View. To do so, select the Service control in Design View, then expand the desired properties in the Properties pane.

Managing Shared Service Controls

When modifying Service controls, keep in mind that a Service control may be shared by multiple web services. If all of the web services using a Service control are referring to a single CTRL file, changing the behavior of that CTRL file will affect the behavior of the Service control for all of the referring web services.

Topics Included in This Section

Customizing a Service Control: Declaring Compliance with a WSDL File

Describes how a Service control typically specifies that it complies with the published interface of the web service it represents.

Customizing a Service Control: Specifying the Default Protocol and Message Format

Describes how a Service control specifies the network protocol (e.g. HTTP, SMTP) and message format (e.g. SOAP−RPC, SOAP Literal) that will be used to communicate with the target web service.

Customizing a Service Control: Specifying Conversation Shape

Describes how to specify or modify the conversational characteristics of a Service control's methods and callbacks.

Customizing a Service Control: Buffering Methods and Callbacks

Describes how to add message buffers to a Service control's methods and callbacks.

Customizing a Service Control: Defining Java to XML Translation with XML Maps

Describes how to use XML maps to control the correspondence between the Java signatures of a Service control's methods and callbacks, and the messages sent to and received from the target web service.

Related Topics

Controls: Using Resources from a Web Service

Customizing a Service Control: Declaring Compliance with a WSDL File

This topic describes customizations you can make to a Service control's CTRL file to modify the control's default behavior.

To learn more about controls, see Controls: Using Resources from a Web Service.

To learn more about Service controls, see Service Control: Using Another Web Service.

To learn more about customizing Service controls, see Customizing a Service Control: Overview.

How Service Controls Specify Contract Compliance

A Service control CTRL file typically explicitly declares that it complies with the public contract expressed in a WSDL file. This compliance is declared with the @jws:wsdl tag. The @jws:wsdl tag's file attribute names the WSDL file with which the Service control complies.

The file specified by the @jws:wsdl tag may be placed in line inside the Service control's CTRL file. The WSDL contents are specified as the value of a @jws:define tag. The @jws:wsdl tag's file attribute then refers to #Name instead of an actual filename, where Name is the value of the @jws:define tag's name attribute. The following sample contents of a CTRL file illustrate this arrangement:

```
import weblogic.jws.control.ServiceControl;
/**
 * @jws:location http-url="creditreport/IRS.jws" jms-url="creditreport/IRS.jws"
 * @jws:wsdl file="#IRSWsdl"
 */
public interface IRSControl extends ServiceControl
{
    ...
}
/**
 @jws:define name="IRSWsdl" value::
      <?xml version=1.0 encoding=utf-8?>
      <definitions ...>
        ...remainder of WSDL here...
      </definitions>
  ::
 */
```

If a Service control has an associated @jws:wsdl tag, the methods in the Service control's interface must be capable of producing messages that comply with the operations defined in the WSDL file.

To learn more about the @jws:wsdl tag, see @jws:wsdl Tag.

Related Topics

CTRL Files: Implementing Controls

WSDL Files: Web Service Descriptions

@jws:define Tag

@jws:wsdl Tag

Customizing a Service Control: Specifying the Default Protocol and Message Format

This topic describes customizations you can make to a Service control's CTRL file to modify the control's default behavior.

To learn more about controls, see Controls: Using Resources from a Web Service.

To learn more about Service controls, see Service Control: Using Another Web Service.

To learn more about customizing Service controls, see Customizing a Service Control: Overview.

Changing a Service Control's Default Protocol or Message Format

In some situations, you may wish to modify the Service control to use a different protocol or message format. Of course, you must choose a protocol or message format that the target web service supports. You can determine the set of protocols and message formats the web service supports by examining the service's WSDL file.

Note: this topic is narrowly focused on modifications you may make to Service control CTRL files, which will influence the protocol and message format used to communicate with existing web service. When developing a new web service with WebLogic Workshop, you may specify that your web service supports multiple protocols and message formats. Should you do so, clients will be able to choose which formats they wish to use to communicate with your web service.

Per−Service and Per−Method Settings

Protocols and message formats may be specified on a Service control or on a method of a Service control. The settings on the Service control apply to all methods and callbacks by default. If a method or callback specifies different protocol or message format settings, those settings override the default settings from the Service control.

Protocols

A protocol is a low level network scheme for passing messages between systems. Using an analogy of postal mail, the protocol describes the nature of an envelope and how to print an address on an envelope. The message format defines the possible content and interpretation of the letter inside the envelope.

The standards on which web services are based allow a web service to support a variety of protocols and message formats.

For example, a web service may support communication via HTTP (Hyper Text Transport Protocol, the basic protocol of web browsers and servers) and/or SMTP (Simple Mail Transport Protocol, the protocol used by email systems to pass messages). It may support other protocols as well.

While a web service may advertise that it supports multiple protocols or message formats, when WebLogic Workshop creates a Service control for a web service, one protocol and one message format are chosen as the default.

Setting a Service Control's Default Protocol

You can modify the protocol that a Service control uses to communicate with its target web service using the @jws:protocol tag.

To learn more about specifying protocols, see @jws:protocol Tag.

Message Formats

The standards that make web services possible allow for multiple formats of the messages that pass between web services. The central standard for web services messages is SOAP. SOAP defines two formats for messages that represent web service method invocations. These are referred to as Document Literal and SOAP RPC (Remote Procedure Call).

WebLogic Workshop uses Document Literal as its default message format, but is capable of communicating using SOAP RPC. Other web service development tools may use SOAP RPC as their default message format. When you wish to implement communications between two web services developed using different tools, you need to ensure that both are using the same message format. You may either specify what format your web service expects and dictate that choice to the developer of the other web service, or you may select an alternative format that allows your service to accept the other web service's default format.

Setting a Service Control's Default Message Format

You can modify the message format that a Service control uses to communicate with its target web service using the @jws:location tag.

To learn more about specifying message formats, see @jws:protocol Tag.

Related Topics

@jws:location Tag

Customizing a Service Control: Specifying Conversation Shape

This topic describes customizations you can make to a Service control's CTRL file to modify the control's default behavior.

To learn more about controls, see Controls: Using Resources from a Web Service.

To learn more about Service controls, see Service Control: Using Another Web Service.

To learn more about customizing Service controls, see Customizing a Service Control: Overview.

Changing a Service Control's Conversation Shape

The conversation shape of a Service control (or a web service) is the specific configuration of conversation phases on the various methods or callbacks of the service. If you change the conversation phase of any method or callback, you have changed the conversation shape of the service.

You may change a Service control's conversation shape by adding @jws:conversation tags to individual operations in the Service control's CTRL file.

Modifying the conversational shape of a Service control is not recommended. It is easy to inadvertently introduce incompatibilities with the target web service's public contract.

In particular, WebLogic Workshop web services that use conversations expect special SOAP headers containing conversation parameters to be included in every conversational message. If the target web service is not capable of understanding WebLogic Workshop conversation SOAP headers, including them in messages by adding @jws:conversation tags to the Service control's methods will violate the web service's contract and prevent successful communication with the web service.

To learn more about conversations, see Maintaining State with Conversations.

To learn more about the @jws:conversation tag, see @jws:conversation Tag.

Related Topics

Maintaining State with Conversations

Implementing Conversations

Customizing a Service Control: Buffering Methods and Callbacks

This topic describes customizations you can make to a Service control's CTRL file to modify the control's default behavior

To learn more about controls, see Controls: Using Resources from a Web Service.

To learn more about Service controls, see Service Control: Using Another Web Service.

To learn more about customizing Service controls, see Customizing a Service Control: Overview.

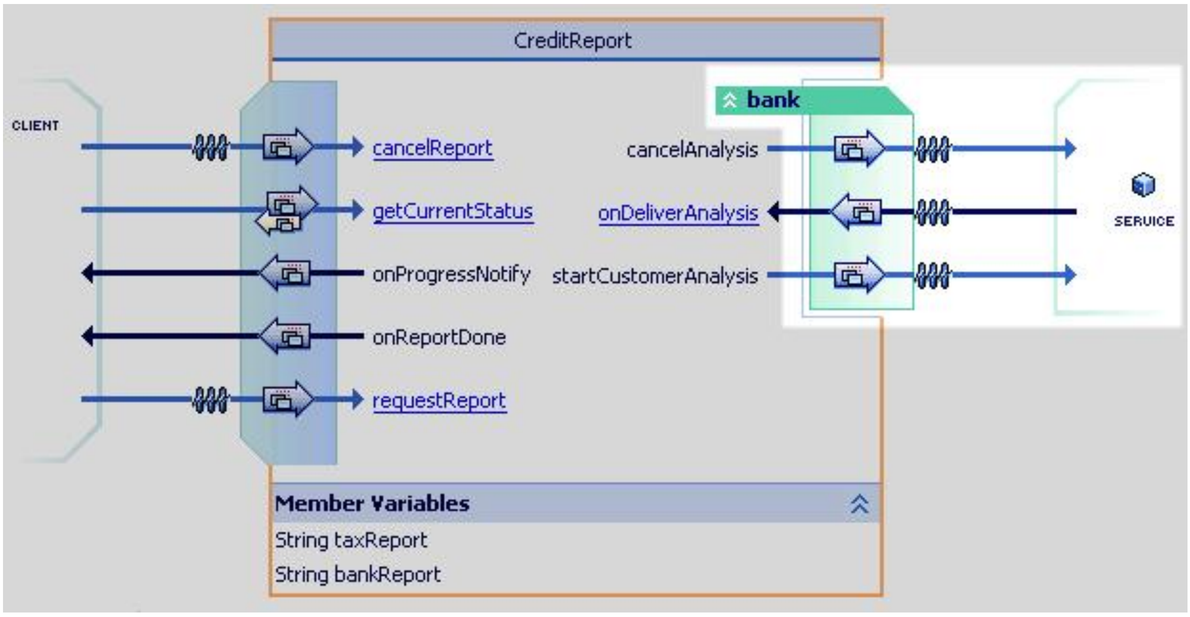
Adding Buffers to Service Control Methods

To learn about message buffers, see Using Asynchronous Methods.

Message buffers may only be added to methods and callbacks that have a void return type.

You may add message buffers to the methods and callback handlers of a Service control. It is important to understand what happens when you add message buffers to Service controls. Remember that a Service control is a proxy for another web service (the target service). In many cases, the target service is located on a remote server. If you add message buffers to a Service control, the buffering always occurs on the local server.

Design View draws message buffers as a "spring" icon on the method or callback, as shown here:



Note that WebLogic Workshop draws message buffers on the "near" end of the wire. In general, you have no control over the execution environment or configuration of remote web services. In other words, you can't change what happens on the "far" end of the wire.

Adding a message buffer to a method makes the method asynchronous, meaning that callers to that method do not wait for a response.

In the case of a Service control, your web service is the client when sending outgoing messages (Service control method invocations) and the other service is the client when sending incoming messages (Service control callbacks) to your service.

Buffer Behavior on Service Control Methods

If you add a message buffer to a Service control's method, outgoing messages (invocations of the method) will be buffered on the local machine. The method invocation will return immediately. This prevents your service from having to wait while the message is sent to the remote server and the (void) response is received. In other words, your service doesn't have to wait for the network roundtrip to occur.

Buffer Behavior on Service Control Callback Handlers

If you add a message buffer to a Service control's callback, incoming messages (invocations of the callback) will be buffered on the local machine. The callback invocation will return to the other web service immediately. This prevents the other service from waiting until your service processes the request. Note that since the buffering occurs on your end of the wire, the calling service still must wait for the network roundtrip even though it will return a void result. But the calling service does not have to wait for your service to process the message.

Adding Message Buffers to a Service Control in WebLogic Workshop

There are two ways to specify that Service control methods and callback handlers should use message buffers:

- In Design View, select the method or callback. In the Tasks pane, select the Task that begins "Buffer this method...".

* In Design View, select the method or callback. In the Properties pane, expand the message−buffer property and set the enabled attribute to true.

These actions will add the appropriate annotations to the Service control's CTRL file. The specific annotations added are described in the next section.

How Message Buffers are Specified in Code

Message buffers are specific in code using the @jws:message−buffer Javadoc tag. In a Service control CTRL file, the @jws:message−buffer tag may be placed on a method or a callback.

To learn about the @jws:message−buffer tag, see @jws:message−buffer Tag.

The following example contains a @jws:message−buffer tag on a Service control method:

```
public interface QuoteServiceControl extends ServiceControl
{
...
   /**
     * @jws:message-buffer enable="true"
     */
    public void getQuote (int customerID, java.lang.String tickerSymbol);
...
}
```

The following example contains a @jws:message−buffer tag on a Service control callback:

```
public interface QuoteServiceControl extends ServiceControl
{
...
    public interface Callback
    {
        /**
         * @jws:message-buffer enable="true"
         */
        public void onQuoteReady (java.lang.String tickerSymbol, double dQuote);
    }
...
}
```

Related Topics

Using Asynchrony to Enable Long−Running Operations

Asynchronous Methods

Customizing a Service Control: Defining Java to XML Translation with XML Maps

This topic describes customizations you can make to a Service control's CTRL file to modify the control's default behavior.

To learn more about controls, see Controls: Using Resources from a Web Service.

To learn more about Service controls, see Service Control: Using Another Web Service.

To learn more about customizing Service controls, see Customizing a Service Control: Overview.

Adding XML Maps to the Methods of a Service Control

You can modify the relationship between the Java interface exposed to clients of a Service control and the XML messages sent to the target web service. The following example is taken from the QuoteServiceControl.ctrl sample Service control used by the QuoteClient.jws Sample.

QuoteServiceControl.ctrl was originally generated from QuoteService.jws. It was then modified: the first parameter of getQuote was removed from the Java signature, meaning callers (web services that use this Service control) are no longer expected to pass it. An XML map was then added to getQuote, with the value of the <customerID> element hard−coded. This leaves the message shape as QuoteService expects it, with two parameters. The calling service passes one parameter, which the Service control combines with the hard−coded parameter to produce the two−parameter message the target service expects.

Here is the code as it was originally generated from QuoteService.jws. Notice that getQuote takes two parameters.

```
import weblogic.jws.control.ServiceControl;
/**
 * @jws:location http-url="QuoteService.jws" jms-url="QuoteService.jws"
 * @jws:wsdl file="#QuoteServiceWsdl"
 */
public interface QuoteServiceControl extends ServiceControl
{
    public interface Callback
    {
        /**
         * @jws:conversation phase="finish"
         */
        public void onQuoteReady (java.lang.String tickerSymbol, double dQuote);
    }
    /**
     * @jws:conversation phase="start"
     */
    public void getQuote (int customerID String tickerSymbol);
}
```

Below is the code after the CTRL file has been modified manually. The customerID parameter has been removed from the method signature and hard−coded into the XML map, which was also manually added. The XML map must comply with the contract of the target web service.

```
import weblogic.jws.control.ServiceControl;
/**
 * @jws:location http-url="QuoteService.jws" jms-url="QuoteService.jws"
 * @jws:wsdl file="#QuoteServiceWsdl"
 */
public interface QuoteServiceControl extends ServiceControl
{
    public interface Callback
    {
        /**
         * @jws:conversation phase="finish"
         */
        public void onQuoteReady (java.lang.String tickerSymbol, double dQuote);
    }
    /**
     * @jws:conversation phase="start"
     * @jws:parameter-xml xml-map::
     *  <getQuote xmlns="http://www.openuri.org/">
     *      <customerID>1234567890</customerID>
     *      <tickerSymbol>{tickerSymbol}</tickerSymbol>
     *   </getQuote>::
     */
    public void getQuote (String tickerSymbol);
}
```

Note that the modifications to the method signature and XML map could also be accomplished in the Edit Maps and Interface Dialog. Access the dialog by double−clicking on the map icon (the fat arrow) for the getQuote method when QuoteClient.jws is being edited in Design View.

Related Topics

Handling and Shaping XML Messages with XML Maps

QuoteClient.jws Sample

EJB Control: Using Enterprise Java Beans from a Web Service

Enterprise Java Beans (EJBs) are Java software components of enterprise applications. The Java 2 Enterprise Edition (J2EE) Specification defines the types and capabilities of EJBs as well as the environment (or container) in which EJBs are deployed and execute. From a software developer's point of view, there are two aspects to EJBs: development and deployment of EJBs; and use of EJBs from client software. WebLogic Workshop provides the EJB control as a simplified way to act as a client of an existing EJB from within a web service.

Note: WebLogic Workshop is not intended for development or deployment of new EJBs; to learn how to develop and deploy EJBs please consult the WebLogic Server documentation or a book on J2EE.

To access the capabilities of an EJB without the EJB control, you would have to perform several preparatory operations. You would look up the EJB in the JNDI registry, obtain the EJBs home interface, obtain an EJB instance, and then finally invoke methods on the EJBs remote interface to do real work.

The EJB control relieves you of all of the preparatory work. Once the EJB control has been created, a web service may use the control to access the EJB's "business methods" directly. The EJB control manages communication with the EJB for you, including all JNDI lookup, interface discovery and EJB instance creation and management.

To learn about WebLogic Workshop controls, see Controls: Using Resources from a Web Service.

Topics Included in this Section

Overview: Enterprise Java Beans (EJBs)

Describes Enterprise Java Beans in general.

Creating a New EJB Control

Describes how to create and configure an EJB control.

Using an EJB Control

Describes how to use an existing EJB control from within a web service.

WebLogic Workshop is Not Intended for EJB Development

WebLogic Workshop is specifically not intended to help you develop and deploy EJBs. In fact, if you store EJB source code in your WebLogic Workshop project you may experience compilation errors because WebLogic Server will attempt to compile source code it finds there. This will conflict with the deployed EJB.

The WebLogic Workshop EJB control is intended to make it easy for you to use an existing, deployed EJB from within a web service.

Related Topics

Controls: Using Resources from a Web Service

Overview: Enterprise Java Beans

As with all controls in WebLogic Workshop, the EJB control is designed to hide most of the details of the resource the control represents. Therefore, you should be able to create an EJB control without knowing most of the information in this topic. This information is provided as background.

If you want to learn more about J2EE and EJBs, please consult the WebLogic Server documentation, java.sun.com or the J2EE programming book of your choice.

What Is an EJB?

An Enterprise Java Bean is a server−side component that encapsulates the business logic of an application. The business logic is the code that fulfills the purpose of the application, as opposed to code that provides infrastructure and plumbing for the application. In an inventory control application, for example, the EJBs might implement the business logic in methods called checkInventoryLevel and orderProduct. By invoking these methods, remote clients can access the inventory services provided by the application.

EJBs always execute within an EJB container, which provides system services to EJBs. These services include transaction management, persistence, pooling, clustering and other infrastructure.

EJB Interfaces

EJB expose two types of interfaces, called the home interface and the business interface (prior to EJB 2.0, the business interface was generally referred to as the bean's remote interface).

Clients obtain an instance of the EJB with which to communicate by using the home interface. The methods in the home interface are limited to those that create or find EJB instances.

Once a client has an EJB instance, it invokes methods of the EJB’s business interface to do real work. The business interface directly accesses the business logic encapsulated in the EJB.

To create an EJB control to represent an EJB, you must know the names of the home and business interfaces. The name for the home interface is typically of the form com.mycompany.MyBeanNameHome and the business interface is typically of the form com.mycompany.MyBeanName.

Types of EJBs

In J2EE 1.3 there are three types of EJBs: Session Beans, Entity Beans and Message−Driven Beans. Each of these types is described briefly in the following sections.

The EJB control provides web services the ability to act as a client for Session and Entity EJBs. Requests can be sent indirectly to Message−Driven bean using the JMS control, thus the EJB control does not support Mesage−Driven Beans.

Session EJBs

A session EJB represents a single client inside the application server. A client of the application accesses the application by invoking the session bean's methods. The session bean shields the client from complexity by executing business tasks inside the server, perhaps by invoking other EJBs.

A session bean is not shared: it may have just one client. Like an interactive session, a session bean is not persistent. When the client terminates, its session bean appears to terminate and is no longer associated with the client.

Entity EJBs

An Entity EJB represents a business object in a persistent storage mechanism. Some examples of business objects are customers, orders, and products. The persistent storage mechanism is a relational database. Typically, each entity bean has an underlying table in a relational database, and each instance of the bean corresponds to a row in that table.

Entity beans differ from session beans in several ways. Entity beans are persistent, allow shared access, have primary keys, and may participate in relationships with other entity beans.

Message−Driven EJBs

A Message−Driven EJB is an enterprise bean that is able to listen for JMS (Java Message Service) messages. The messages may be sent by any JMS−compliant component or application. Message−Driven EJBs provide a mechanism for J2EE applications to participate in relationships with message−based legacy applications.

Related Topics

EJB Control: Using Enterprise Java Beans from a Web Service

Controls: Using Resources from a Web Service

Creating a New EJB Control

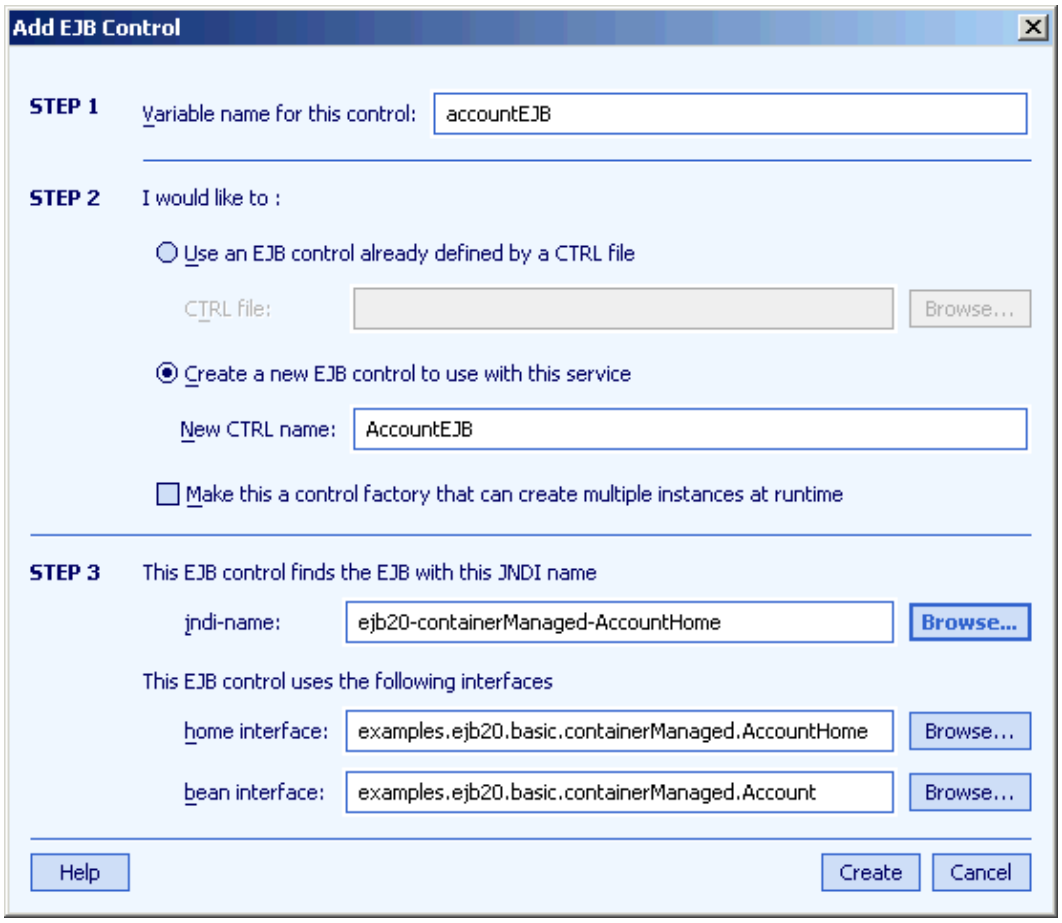
This topic describes how to create a new EJB control.

To learn about EJB controls, see EJB Control: Using Enterprise Java Beans from a Web Service.

To learn about WebLogic Workshop controls, see Controls: Using Resources from a Web Service.

Creating a New EJB Control

You can create a new EJB control and add it to your web service by using the Add EJB Control Dialog. The Add EJB Control Dialog can be accessed via the Service menu on the Menu Bar; via the right−button context menu on the service in the Design View; or via the Add Control option menu in the upper right corner of Design View.


Add EJB Control

STEP 1 Variable name for this control: accountEJB

STEP 2 I would like to :

Use an EJB control already defined by a CTRL file

CTRL file:

[✓ Create a new EJB control to use with this service]

New CTRL name: AccountEJB

☐ Make this a control factory that can create multiple instances at runtime

STEP 3 This EJB control finds the EJB with this JNDI name

jndi-name: ejb20-containerManaged-AccountHome

This EJB control uses the following interfaces

home interface: examples.ejb20.basic.containerManaged.AccountHome

bean interface: examples.ejb20.basic.containerManaged.Account

To learn more about the Add EJB Control Dialog, see Add EJB Control Dialog.

Preparing the Target EJB

Deploying the Target EJB

Before an EJB control will function, the EJB it represents must be deployed in WebLogic Server. To learn how to deploy an EJB, please refer to the WebLogic Server documentation.

Making the EJB's Interfaces Available to WebLogic Workshop

In order for the Add EJB Dialog to discover the EJB's home and remote interfaces, the EJB's compiled interfaces must be copied to your WebLogic Workshop project.

This is typically accomplished by obtaining a copy of the EJB's JAR file and placing the copy in the

WEB−INF/lib directory of your WebLogic Workshop project.

While the complete EJB JAR file will allow WebLogic Workshop to access the EJB's home and remote interfaces, the only classes actually required are the EJB's home and remote interface classes and any other classes used externally by the EJB (for example, as method parameters or method return types). The EJB compiler ejbc is capable of producing a client JAR that will serve this purpose. To learn more about EJB client JAR files, please consult the WebLogic Server documentation topic "Packaging EJBs for the WebLogic Server Container".

Accessing Remote EJBs

You may access remote EJBs using the EJB control. You access remote EJBs by using special syntax in the jndi−name attribute. For example:

jndi://username@password:host:7001/my.resource.jndi.object

Note: Remote EJB access in the EJB control is only supported when the server hosting the EJB control and the server to which the target EJB is deployed are in the same domain. Accessing a remote EJB in a different domain via the EJB control requires advanced transaction configuration. Please consult the WebLogic Server documentation for details.

EJB Control CTRL Files

Some WebLogic Workshop controls may be modified by editing the control's CTRL file. This is not true for the EJB control. The EJB control's CTRL file is created by WebLogic Workshop based on information it obtains directly from the target EJB's code. You should not modify an EJB control's CTRL file.

Related Topics

Controls: Using Resources from a Web Service

Using an EJB Control

Using an Existing EJB Control

This topic describes how to use an existing EJB control in your web service.

To learn about controls, see the Controls Overview.

To learn about EJB controls, see EJB Control: Using Enterprise Java Beans from a Web Service.

To learn how to create a EJB control, see Creating a New EJB Control .

Using an Existing EJB Control

All controls follow a consistent model. Therefore, most aspects of using an existing EJB control are identical to using any other existing control.

To learn about using an existing control, see Using an Existing Control.

EJB Interfaces Required

In order for WebLogic Workshop to provide code completion and otherwise successfully represent the target EJB, the EJB's client interfaces must be available to WebLogic Workshop. You make the EJB's client interfaces available by placing the EJB's client JAR file or the EJB JAR file in the WEB−INF/lib directory of your WebLogic Workshop project.

If you copy an existing EJB control from one WebLogic Workshop project to another, you must also copy the associated JAR file(s) containing the EJB's client interfaces.

Invoking EJB Methods

You invoke methods on the target EJB by invoking the method of the same signature on the EJB control. If the EJB interfaces are available to the WebLogic Workshop visual development environment as described in the preceding section, the Source View will provide code completion for EJB method names and method parameter lists.

To see all methods available for an EJB control, type the name of the EJB control instance followed by a period ("."). For example, "account." where account is the variable name of the control instance.

To see the arguments for a method, type the control instance name, period, the method name, then an opening parenthesis. For example, "account.withdraw(" where account is the variable name of the control instance.

EJB Instance Selection

The EJB control automatically manages references to EJB instances and directs method invocations to the correct instance of the target EJB automatically. The following sections describe what happens when you invoke various EJB methods. What happens depends on whether the target EJB is a Session Bean or an Entity Bean.

Session Beans

The create Method

If the target EJB is a Session Bean, you do not need to invoke a create method of the EJB. The EJB control will automatically create and cache a reference to an appropriate instance of the EJB whenever one of the EJB's business methods is invoked.

Note that the create method for a Session Bean may not create a new instance of the bean; the server may return an existing instance from a pool.

EJB Reference Caching

As described in the preceding paragraph, a reference to a EJB instance is created whenever one of the EJB's business methods is called. This reference is cached in the EJB control and is used for any subsequent calls to the EJB within the current web service method invocation. Remember that the EJB control is being used by a web service. For Session Beans, the lifetime of the reference is the lifetime of the web service method invocation in the reference was created.

Any explicit call to a Session Bean's create method will replace the previously cached reference (if any) with the newly created reference.

Note: The lifetime of the cached EJB reference within the EJB control depends on the type of the target EJB. If the target EJB is a Session Bean, the EJB reference's lifetime is the lifetime of the current web service method invocation.

The remove Method

When you call the remove method on an EJB control that represents a Session Bean, the currently cached instance of the bean is released. The server may destroy the bean at that time, but the actual behavior is up to the server.

Entity Beans

Instances of Entity Beans, unlike Session Beans, are associated with a particular collection of data. Typically this collection of data is a row in a database table. Therefore, the creation and selection of Entity beans works differently from Session beans.

The create Method

To create an instance of an Entity Bean, you may invoke the EJB's create method explicitly. For Entity Beans, the create method creates a new row in the underlying database table (that is, a new persistent entity).

The findByPrimaryKey or findXxx Methods

Entity Bean instances may also be referenced by calling the findByPrimaryKey method or another findXxx method provided by the EJB's designer.

EJB Reference Caching

The EJB control caches a reference to Entity Bean instance being used in the current conversation. In the case of Entity Beans, this is the instance returned by the most recent call to create, findByPrimaryKey or findXxx. When you invoke subsequent methods on the EJB control, the EJB control will invoke that method on the bean instance to which the cached reference refers. If there is no EJB reference currently cached, the EJB control will attempt to invoke findByPrimaryKey with the last successful key used in a create or findByPrimaryKey call. If there is no previous key, the EJB control will throw an exception.

Note: The lifetime of the cached EJB reference within the EJB control depends on the type of the target EJB. If the target EJB is an Entity Bean, the EJB reference's lifetime is the lifetime of the web service conversation. For web service methods that are not conversational (ie. marked with @jws:conversation phase="none"), the lifetime of the conversation (and therefore the EJB reference) is the lifetime of the web service method invocation.

Finder Methods that Return Multiple Records

A findXxx method may return an Enumeration or a Collection object. The EJB control does not cache this object. If you wish to cache the return value of a findXxx method, you should store the object in a member variable of your (conversational) web service.

To learn how to persist state with conversations, see Maintaining State with Conversations.

The remove Method

When you call the remove method on an EJB control that represents an Entity Bean, the record represented by the cached EJB reference is removed from the underlying persistent storage (ie. the row is deleted from the database table).

Related Topics

Overview: Enterprise Java Beans (EJBs)

Creating a New EJB Control

Maintaining State with Conversations

Handling EJB Exceptions

This topic describes how to handle exceptions that might be thrown by the EJB that is the target of an EJB control.

To learn about EJB controls, see EJB Control: Using Enterprise Java Beans from a Web Service.

To learn about WebLogic Workshop controls, see Controls: Using Resources from a Web Service.

EJB Exceptions Are Wrapped

If the target EJB of an EJB control throws an exception, the try−catch block must catch both the actual exception thrown and also weblogic.rmi.extensions.RemoteRuntimeException. The actual thrown exception will never be caught, but the catch block must be there for the compiler.

When a weblogic.rmi.extensions.RemoteRuntimeException is caught, use its getNestedException method to get the actual exception thrown by the EJB.

Note: This is a limitation in the Beta release of WebLogic Workshop. In the future, the EJB control will unwrap the exception for you.

Example

The following example demonstrates the technique described above.

The AccountEJB.jar defines the method withdraw(Double amount) that throws a ProcessingErrorException if the amount of the withdrawal is greater than the current balance.

```
    /**
     * @jws:control
     */
    private AccountEJBControl account;



    /**
     * Returns from the EJB the new account balance of the account id after a withdrawal.
     *
     * @return A String account balance, or the text of any exception that occurs.
     *
     * @jws:operation
     */
    public String withdraw(String accountKey, double withdrawAmount)
    {
        try
        {
            return String.valueOf(account.findByPrimaryKey(accountKey).withdraw(withdrawAmount)
        }
        // catch the reuntime exception
        catch (weblogic.rmi.extensions.RemoteRuntimeException rre)
        {
            // find the real exception
            Throwable t = rre.getNestedException();
            if (t instanceof ProcessingErrorException)
            {
                // perform the exception processing
                return t.getLocalizedMessage();
            }
            else
            {
                // re-throw any other exceptions.
                throw rre;
            }
        }
        catch (FinderException fe)
        {
            return fe.getLocalizedMessage();
        }
        catch (RemoteException re)
        {
            return re.getLocalizedMessage();
        }
        catch (ProcessingErrorException pe)
        {
            // this keeps the compiler happy and will not execute.
            return pe.getLocalizedMessage();
        }
    }
```

Related Topics

Introduction to Java

JMS Control: Using Java Message Service Queues and Topics from Your Web Service

JMS (Java Message Service) is a Java API for communicating with messaging systems. Messaging systems are often packaged as products known as Message−Oriented Middleware (MOMs). WebLogic Server includes built in messaging capabilities via WebLogic JMS, but can also work with third−party MOMs. Messaging systems are often used in enterprise applications to communicate with legacy systems, or for communication between business components running in different environments or on different hosts.

The JMS control enables WebLogic Workshop web services to easily interact with messaging systems that provide a JMS implementation. A specific JMS control is associated with particular facilities of the messaging system. Once a JMS control is defined, web services may use it like any other WebLogic Workshop control.

To learn about WebLogic Workshop controls, see Controls: Using Resources from a Web Service.

Topics Included in This Section

Overview: Messaging Systems and JMS

Describes messaging services in general and the Java Message Service in particular

Messaging Scenarios Supported by the JMS Control

Describes appropriate scenarios in which the JMS control may be used.

Messaging Scenarios Not Supported by the JMS Control

Describes scenarios in which the JMS control may not be used.

Creating a New JMS Control

Describes how to create and configure a JMS control.

Sending and Receiving XML Messages with a JMS Control

Describes how to apply XML maps to the messages sent and received via a JMS control. XML maps provide an easy and automatic way to convert between XML messages and Java objects. XML maps can also be used to encode and extract JMS headers and properties in or from messages.

Manipulating JMS Message Headers and Message Properties in a JMS Control

Describes how to apply XML maps to the headers and properties of messages sent and received by the JMS control.

Using a JMS Control

Describes how to use an existing JMS control from within a web service.

JMS Control Samples

SimpleJMS.jws Sample

CustomJMSClient.jws Sample

Related Topics

Controls: Using Resources from a Web Service

Overview: Messaging Systems and JMS

This topic describes the characteristics of messaging systems that are accessible via JMS (Java Message Service), and therefore via the WebLogic Workshop JMS control.

To learn about WebLogic Workshop controls, see Controls: Using Resources from a Web Service.

To learn about the JMS control, see JMS Control: Using Java Message Service Queues and Topics from your Web Service.

To learn about specific messaging scenarios that are supported by the JMS control, see Messaging Scenarios Supported by the JMS Control.

Messaging Systems

Messaging systems provide communication between software components. A client of a messaging system can send messages to, and receive messages from, any other client. Each client connects to a messaging server that provides facilities for sending and receiving messages. WebLogic JMS, which is a component of WebLogic Server is an example of a messaging server. WebLogic Server also supports third party messaging systems.

Messaging systems provide distributed communication that is asynchronous. A component sends a message to a destination. A message recipient can retrieve messages from a destination. The sender and receiver do not communicate directly. The sender only knows that a destination exists to which it can send messages, and the receiver also knows there is a destination from which it can receive messages. As long as they agree what message format and what destination to use, the messaging system will manage the actual message delivery.

Messaging systems also may provide reliability. The specific level of reliability is typically configurable on a per−destination or per−client basis, but messaging systems are capable of guaranteeing that a message will be delivered, and that it will be delivered to each intended recipient exactly once.

JMS supports two basic styles of message−based communications: point−to−point and publish and subscribe.

JMS Queues for Point−to−Point Messaging

Point−to−point messaging is accomplished with JMS queues. A queue is a specific named resource that is configured in a JMS server.

A JMS client, of which the JMS control is an example, may send messages to a queue or receive messages from a queue. Point−to−point messages have a single consumer. Multiple receivers may listen for messages on the same queue, but once any receiver retrieves a particular message from the queue that message is consumed and is no longer available to other potential consumers.

A message consumer acknowledges receipt of every message it receives.

The messaging system will continue attempting to resend a particular message until a predetermined number of retries have been attempted.

JMS Topics for Publish and Subscribe Messaging

Publish and subscribe messaging is accomplished with JMS topics. A topic is a specific named resource that is configured in a JMS server.

A JMS client, of which the JMS control is an example, may publish messages to a topic or subscribe to a topic. Published messages have multiple potential subscribers. All current subscribers to a topic receive all messages published to that topic after the subscription becomes active.

Connection Factories

Before a JMS client can send or receive messages to a queue or topic, it must obtain a connection to the messaging system. This is accomplished via a connection factory. A connection factory is a resource that is configured by the message server administrator. The names of connection factories are stored in a JNDI directory for lookup by clients wishing to make a connection.

There is a default connection factory pre−configured in WebLogic Workshop, named cgConnectionFactory. This connection factory is used for all JMS controls that do not explicitly override it.

Related Topics

JMS Control: Using Java Message Service Queues and Topics from your Web Service

Messaging Scenarios Supported by the JMS Control

This topic describes specific messaging scenarios that are supported by the JMS control.

To learn more about WebLogic Workshop controls, see Controls: Using Resources from a Web Service.

To learn more about JMS, the Java Message Service, see Overview: Messaging Systems and JMS.

To learn more about the JMS control, see JMS Control: Using Java Message Service Queues and Topics from Your Web Service.

Supported Messaging Scenarios

The JMS specification supports a wide variety of messaging scenarios. Some scenarios that are possible in standalone applications are not possible in the WebLogic Workshop environment due to the nature of web services.

The messaging scenarios in the following sections are supported by the JMS control. For descriptions of messaging scenarios that are not supported by the JMS control, see Messaging Scenarios Not Supported by the JMS Control.

Send Messages to a Queue

A web service, via a JMS control, may send messages to a JMS queue. The web service will not receive a reply. The queue must exist and be registered in the JNDI registry. The administrator who configures the target JMS queue determines the delivery guarantee policies.

Example

To implement this scenario:

1. On the JMS control, specify the name of the target JMS queue as the value of the send−jndi−name attribute of the JMS control's @jws:jms property. To learn how to create a JMS control, see Creating a New JMS Control.
2. From your web service, call the JMS control's default method (sendTextMessage, sendMessage, sendObjectMessage or sendJMSMessage depending on the message type selected when the control was created); or a custom method you have defined for the JMS control.

The following is an example of this scenario. The example assumes you have followed the steps above.

```
public class MyService {

    /**
     *   @jws:control
     */
    private MyQueueControl myQueue;
    /**
     * @jws:operation
     */
    public void sendID(String personID)
    {
        myQueue.sendTextMessage( personID );
    }
}
```

Two−Way Messaging with Queues

A web service, via a JMS control, may send messages to one queue and receive reply messages on another queue. A single JMS control may have both send and receive queues configured, and web services may then send and receive via the same control.

Note: Two−way messaging requires correlation of every received messages with the instance of the web service that sent the original outgoing message. This is typically managed for you by the JMS control with no action necessary on your part. To learn more about message correlation, see the explanation of the send−correlation−property and receive−correlation−property attributes in @jws:jms Tag.

Example

To implement this scenario:

1. On the JMS control, specify the name of the JMS queue to which you want to send messages as the value of the send−jndi−name attribute of the JMS control's @jws:jms tag.
2. Specify the name of the JMS queue from which you want to receive messages as the value of the receive−jndi−name attribute of the JMS control's @jws:jms tag.
3. To send a message from your web service, call the JMS control's default method (sendTextMessage, sendMessage, sendObjectMessage or sendJMSMessage depending on the message type selected when the control was created); or a custom method you have defined for the JMS control.
4. To be notified when messages are received on the receive queue, implement a callback handler for the JMS control’s callback (receiveTextMessage, receiveMessage, receiveObjectMessage or receiveJMSMessage depending on the message type selected when the control was created); or a custom callback you have defined for the JMS control.

The following is an example of this scenario:

```
public class MyService {

    /**
     *   @jws:control
     */
    private MyQueueControl myQueue;

    /**
     * @jws:operation
     */
    public void sendID(String personID) throws Exception
    {
        myQueue.sendTextMessage( personID );
    }
    myQueue_receiveTextMessage(String message)
    {
        // message has arrived from queue, perform desired operations
    }
}
```

Publish to a Topic

A web service, via a JMS control, may publish messages to a JMS topic. The web service will not receive a reply. The topic must exist and be registered in the JNDI registry.

Example

To implement this scenario:

1. On the JMS control, specify the name of the target JMS topic as the value of the send−jndi−name attribute of the JMS control's @jws:jms property.
2. From your web service, call the JMS control's default method (sendTextMessage, sendMessage, sendObjectMessage or sendJMSMessage depending on the message type selected when the control was created); or a custom method you have defined for the JMS control.

The following is an example of this scenario:

```
public class MyService {

    /**
     *   @jws:control
     */
    private MyTopicControl myTopic;

    /**
     * @jws:operation
     */
    public void sendID(String personID) throws Exception
    {
        myTopic.sendTextMessage( personID );
    }
}
```

Subscribe to a Topic

A web service, via a JMS control, may subscribe to messages on a JMS topic. The topic must exist and be registered in the JNDI registry. Only messages sent after the web service has subscribed to the topic will be received.

Example

To implement this scenario:

1. On the JMS control, specify the name of the target JMS topic as the value of the receive−jndi−name attribute of the JMS control's @jws:jms tag.
2. From your web service, call the JMS control's subscribe method .
3. To be notified when messages are received on the receive topic, implement a callback handler for the JMS control’s callback (receiveTextMessage, receiveMessage, receiveObjectMessage or

receiveJMSMessage depending on the message type selected when the control was created); or a custom callback you have defined for the JMS control.
4. To stop being notified when messages are received on the receive topic, call the JMS control's unsubscribe method.

The following is an example of this scenario:

```
public class TopicForwardingService {



    /**
     * @jws:control



     */
    private MyTopicControl myTopic;



    /**
     * @jws:operation
     * @jws:conversation phase=start
     */
    public void registerListener()
    {
        myTopic.subscribe();
    }



    /**
     * @jws:operation
     * @jws:conversation phase="finish"
     */
    public void unregisterListener()
    {
        // This isn't strictly necessary, the JWS will always
        // be unsubscribed on conversation finish.
        myTopic.unsubscribe();
    }



    myTopic_receiveTextMessage(String message)
    {
        // message has arrived from queue, perform desired operations
    }
}
```

Related Topics

Overview: Messaging Systems and JMS

Messaging Scenarios Not Supported by the JMS Control

@jws:jms Tag

@jws:jms−header Tag

@jws:jms−message Tag

@jws:jms−property Tag

Messaging Scenarios Not Supported by the JMS Control

This topic describes specific messaging scenarios that are not supported by the JMS control.

To learn more about WebLogic Workshop controls, see Controls: Using Resources from a Web Service.

To learn more about the JMS control, see JMS Control: Using Java Message Service Queues and Topics from Your Web Service.

Unsupported Scenarios

The JMS specification supports a wide variety of messaging scenarios. Some scenarios that are possible in standalone applications are not possible in the WebLogic Workshop environment due to the nature of web services.

The messaging scenarios in the following section are not supported by the JMS control. For descriptions of messaging scenarios that are supported by the JMS control, see Messaging Scenarios Supported by the JMS Control.

Receive Unsolicited Messages from a Queue

A web service may not, via a JMS control, specify a receive queue and subsequently receive unsolicited messages from that queue.

A web service must be performing work on behalf of a specific client and, in asynchronous situations, as part of a specific conversation. When an unsolicited messages is received from a queue, it is not possible for the JMS control to determine the appropriate conversation or client with which to correlate unsolicited incoming messages.

Note: You may receive unsolicited messages in a web service, but the web service must be a direct JMS client (i.e. not using a JMS control for the queue in question). To learn how to use JMS directly, please consult the WebLogic Server documentation topic "Programming WebLogic JMS".

Related Topics

Overview: Messaging Systems and JMS

Messaging Scenarios Supported by the JMS Control

Creating a New JMS Control

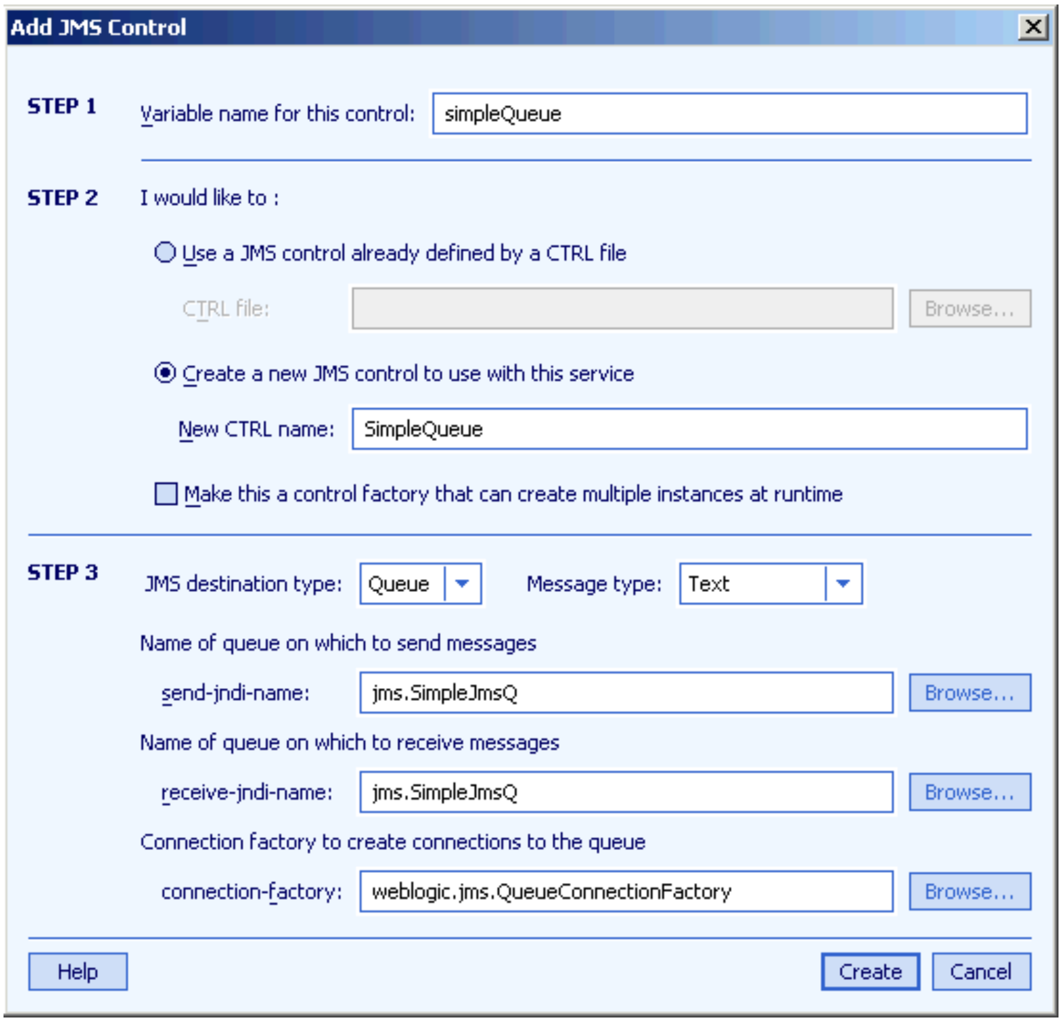
This topic describes how to create a new JMS control.

To learn about JMS controls, see JMS Control: Using Java Message Service Queues and Topics from Your Web Service.

To learn about WebLogic Workshop controls, see Controls: Using Resources from a Web Service.

Creating a New JMS Control

You can create a new JMS control and add it to your web service by using the Add JMS Control Dialog. The Add JMS Control Dialog can be accessed via the Service menu on the Menu Bar; via the right−button context menu on the service in the Design View; or via the Add Control option menu in the upper right corner of Design View.


Add JMS Control

STEP 1 Variable name for this control: simpleQueue

STEP 2 I would like to :

○ Use a JMS control already defined by a CTRL file

CTRL file:

● Create a new JMS control to use with this service

New CTRL name: SimpleQueue

☐ Make this a control factory that can create multiple instances at runtime

STEP 3 JMS destination type: Queue Message type: Text

Name of queue on which to send messages

send-jndi-name: jms.SimpleJmsQ

Name of queue on which to receive messages

receive-jndi-name: jms.SimpleJmsQ

Connection factory to create connections to the queue

connection-factory: weblogic.jms.QueueConnectionFactory

To learn more about the Add JMS Control Dialog, see Add JMS Control Dialog.

Alternatively, you may create a JMS control CTRL file manually. For example, you may copy an existing JMS control CTRL file and modify the copy.

The CTRL File for JMS Control

If you create a new JMS control CTRL file using the Add JMS Control dialog with the settings shown above, the new CTRL file will look like this:

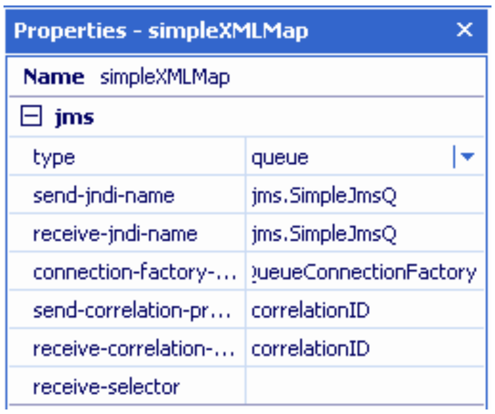
```
import weblogic.jws.control.JMSControl;
import java.io.Serializable;
/**
 *   @jws:jms type="queue" send−jndi−name="jms.SimpleJmsQ" receive−jndi−name="jms.SimpleJmsQ"
 *   connection−factory−jndi−name="weblogic.jws.jms.QueueConnectionFactory"
 */
public interface SimpleQueueControl extends JMSControl
{
    /**
     * this method will send a javax.jms.TextMessage to send-jndi-name
     */
    public void sendTextMessage(String payload);
    /**
     * If your control specifies receive-jndi-name, that is your JWS expects to receive message
     * from this control, you will need to implement callback handlers.
     */
    interface Callback extends JMSControl.Callback
    {
        /**
         * Define only 1 callback method here.
         *
         * This method defines a callback that can handle text messages from receive-jndi-name
         */
         public void receiveTextMessage(String payload);
    }
}
```

The CTRL file contains the declaration of a Java interface with the name specified in the dialog. The interface extends the JMSControl base interface.

The contents of the JMS control's CTRL file depend on the selections made in the Add JMS Control dialog. The example above was generated in response to selection of Text as the Message type drop−down list.

Configuring the Properties of a JMS Control

Most aspects of a JMS control can be configured from the Properties Pane in Design View. If you select a JMS control instance in Design View, the Properties Pane will display the following properties:


Properties - simpleXMLMap

Name simpleXMLMap

jms

type	queue
send-jndi-name	jms.SimpleJmsQ
receive-jndi-name	jms.SimpleJmsQ
connection-factory-...	›ueueConnectionFactory
send-correlation-pr...	correlationID
receive-correlation-...	correlationID
receive-selector	

These properties are encoded in the JMS control's CTRL file as attributes of the @jws:jms Javadoc tag.

For detailed information on the @jws:jms tag and its attributes, see @jws:jms Tag.

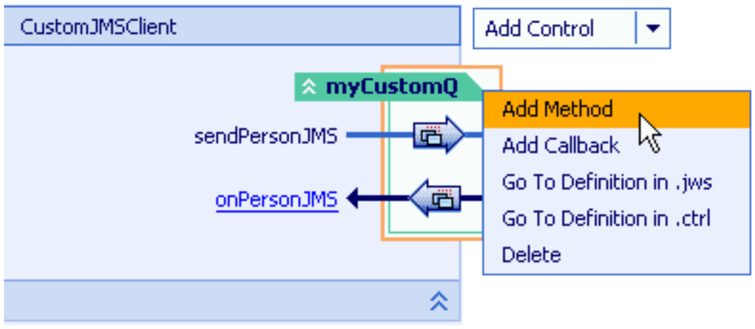
To learn how to create, configure and register JMS queues, topics and connection factories, please consult the WebLogic Server documentation on Programming WebLogic JMS.

Two queues are configured when WebLogic Workshop is installed, in order to support JMS control samples. These are named SimpleJmsQ and CustomJmsCtlQ. The connection factory that provides connections to these queues has the JNDI name weblogic.jws.jms.QueueConnectionFactory. These resources may be used for experimentation.

Note: each JMS control should use unique queues. Multiple JMS controls on the same server may not simultaneously use the same queue.

Adding a Method or Callback in Design View

If you are editing a web service in Design View and you wish to add a method or callback to a JMS control your web service is using, you may add the method directly by right−clicking on the JMS control instance in Design View.



In the example above, the CustomJMSClient web service is using a JMS control whose instance name is myCustomQ. Right−clicking on the myCustomQ object will present a menu that contains the Add Method and Add Callback actions. Select the Add Method action to add a method to the JMS control of which myCustomQ is an instance. Select the Add Callback action to add a callback to the JMS control of which myCustomQ is an instance.

Note: The JMS control variable declared in a web service's JWS file is an instance of a JMS control. The actual control is defined in a CTRL file. When you add a method or callback to a JMS control from Design View, the method or callback is added to the CTRL file. The method or callback will subsequently be available in all web services that are using that JMS control.

Note: You are free to remove the default method and/or callback in a JMS control.

Note: A JMS control may define only one callback.

When a method or callback is added, it will initially have no parameters and no associated XML maps.

Specifying the Format of The Message Body

Within a JMS control, you may define multiple methods and one callback. All methods will send or publish to the queue or topic named by send−jndi−name, if present.

JMS defines several message types that may be sent and or published. The JMS control can send the JMS message types TextMessage, ObjectMessage and Message. The JMS control dynamically determines which type of message to send based on the configuration of the JMS control method that was called.

If the JMS control method takes any XML map (@jws:jms−message, @jws:jms−header or @jws:jms−property), a TextMessage is sent containing the payload, headers and properties specified by the XML maps. To learn how to use XML maps to control the message payload, see Sending and Receiving XML Messages with a JMS Control.

If the JMS control method takes a single String argument and has no XML maps, a TextMessage is sent.

If the JMS control method takes a single argument of a type other than String or javax.jms.Message and has no XML maps, an ObjectMessage is sent.

If the JMS control method takes a single argument of type javax.jms.Message and has no XML maps, that Message object is sent directly.

Specifying Message Headers and Properties

To edit the parameter list and associated XML maps controlling the message headers and message properties, see Manipulating JMS Message Headers and Message Properties in a JMS Control.

Accessing Remote JMS Resources

The JNDI names specified for send−jndi−name, receive−jndi−name and connection−factory may refer to remote JMS resources. The fully specified form of a JMS resource names is:

jms:{provider−host}/{factory−resource}/{dest−resource}?{provider−parameters}

For example:

jms://host:7001/cg.jms.QueueConnectionFactory/jws.MyQueue?URI=/drt/Bank.jws

or:

jms://host:7001/MyProviderConnFactory/MyQueue?SECURITY_PRINCIPAL=foo&SECURITY_CREDENTIALS=b

JMS Control Caveats

Bear in mind the following caveats when you work with JMS controls:

- If you have multiple web services (multiple types, not instances) that reference the same receive−jndi−name for a queue, you must use the receive−selector attribute such that the web services partition all received messages into disjoint sets. If this is not handled properly, messages for a particular conversation may be sent to a control instance that does not participate in that conversation. Note that if you rename a web service that uses a JMS control without undeploying the initial version, the initial version and the new version will be using an identically configured JMS control and will violate this caveat.
- You may have only one callback defined for any JMS control instance (receiveTextMessage, receiveMessage, receiveObjectMessage or receiveJMSMesage, or a developer−defined callback).
- If the underlying JMS control infrastructure receives a message that it cannot deliver to a control instance (e.g. no conversation ID for a control that listens to a queue), it will throw an exception from the JMSControl.onMessage method. This will cause the current transaction to be rolled back. The behavior after that depends on how the administrator set up the JMS destination. Ideally, it should be set up to have a small retry count and an error destination.

Note: If the destination is configured with a large (or no) retry count and no error destination, the JMS control infrastructure will continue attempting to process the message (unsuccessfully) forever.

Related Topics

Overview: Messaging Systems and JMS

@jws:jms−header Tag

@jws:jms−property Tag

@jws:jms−message Tag

JMSControl Interface

Sending and Receiving XML Messages with a JMS Control

This topic describes how to use XML maps to translate between the Java objects that are method and callback parameters in the JMS control's interface and the body of messages sent and received by the control.

To learn about WebLogic Workshop controls, see Controls: Using Resources from a Web Service.

To learn about JMS controls, see JMS Control: Using Java Message Service Queues and Topics from Your Web Service.

To learn how to manipulate message headers and message properties (as opposed to the message body), see Manipulating JMS Message Headers and Message Properties in a JMS Control.

XML Messages and JMS Controls

A JMS control can send and receive one of text messages, XML messages, object messages or javax.jms.Message objects. If you create the JMS control using the Add JMS Control Dialog, the JMS control's CTRL file will contain the appropriate code to send and receive messages of the type selected from the Message type drop−down list in the Add JMS Control dialog.

If the JMS control is configured with the XML Map message type, the JMS control converts between the Java types in the control’s method interface and XML messages using XML maps that you specify.

XML maps are used throughout WebLogic Workshop to convert XML messages to Java objects and vice versa. To learn about general XML map concepts, see How Do XML Maps Work?

Specifying XML Maps for JMS Control Methods and Callbacks

You may specify the XML maps that are used to control the message body, message headers and message properties using the Edit Maps and Interface dialog. To learn how to display and use the Edit Maps and Interface dialog, see The Edit Maps and Interface Dialog.

Example

The example below is the CTRL file generated if you select XML Map from the Message type drop−down list in the Add JMS Control dialog (instructional comments have been removed from the generated code for brevity):

```
import weblogic.jws.control.JMSControl;
import java.io.Serializable;



/**
 *   @jws:jms type="queue" send−jndi−name="jms.SimpleJmsQ" receive−jndi−name="jms.SimpleJmsQ"
 *   connection−factory−jndi−name="cg.jms.QueueConnectionFactory"
 */
public interface SimpleXMLMapControl extends JMSControl
```

```
{
    /**
     * @jws:jms-message xml-map::
     * <YourOuterTag>
     *   <SampleParameter1>{param1}</SampleParameter1>
     *   <SampleParameter2>{param2}</SampleParameter2>
     * </YourOuterTag>::
     */
    public void sendMessage(String param1, String param2);



    interface Callback extends JMSControl.Callback
    {
        /**
         * @jws:jms-message xml-map::
         * <YourOuterTag>
         *   <SampleParameter1>{param1}</SampleParameter1>
         *   <SampleParameter2>{param2}</SampleParameter2>
         * </YourOuterTag>
         * ::
         */
         public void receiveMessage(String param1, String param2);
    }
}
```

The generated code contains a method named sendMessage and a callback named receiveMessage. JMS controls that pass XML messages may define one or more methods of any name and signature. So you may change the name of the generated method and add other methods to the generated interface. You should replace the existing parameter list (param1, param2) with whatever parameter list you require, then refer to the parameters in the XML map.

JMS controls may also define a single callback of any name and signature. You may change the name and signature of the generated callback.

The example above shows XML maps applied to the message body, using the @jws:jms−message tag; the message body may be manipulated with XML maps only if the JMS control was created with the XML Map message type.

To learn how to manipulate the XML message format, see Sending and Receiving XML Messages with a JMS Control.

XML maps may also be used to control the message headers (using the @jws:jms−header tag) and message properties (using the @jms:jws−property tag) on JMS controls of any message type (Text, XML Map, Object or JMS Message). To learn how to manipulate message headers and message properties, see Manipulating JMS Message Headers and Message Properties in a JMS Control.

Example of Customized XML Map JMS Control

The following is an example of a JMS control that defines a sendID method and an onNameReply callback:

```
/**
 * @jws:jms type="queue"
 *          send-jndi-name="OutQueue"
 *          receive-jndi-name="InQueue"
 */
public interface GetPersonControl extends JMSControl
{
    /**
 * @jws::jms-header xml-map::

 *   <header>

 *     <command>GetPerson</command>

 *   </header>

 * ::

 *
 * @jws::jms-property xml-map::

 *   <property>

 *     <requesterID>123</requesterID>

 *   </property>

 * ::

 *

 * @jws:jms-message xml-map::
 *   <person>
 *     <identifier>{personID}</identifier>
 *   </person>
 * ::
 *
 */
public void sendID(String personID);

public static interface Callback extends JMSControl.Callback
{

    /**
     * @jws:jms-message xml-map::
     *   <person>
     *     <firstname>{firstname}</firstname>
     *     <lastname>{lastname}</lastname>
```

```
         *   </person>::
         */
        public void onNameReply(String firstname, String lastname);
    }
}
```

The sendID method sends a message to the queue named OutQueue.

The @jws:jms−header tag sets message headers in the outgoing message. This example includes a JMS message header named command with the hard−coded value GetPerson in the outgoing XML message. The outermost element of the XML map for @jws:jms−header must be <header>.

The @jws:jms−property tag sets message properties in the outgoing message. This example includes a JMS message property named requesterID with the hard−coded value 123 in the outgoing XML message. The outermost element of the XML map for @jws:jms−property must be <property>.

The @jws:jms−message tag formats the message body. The message body will consist of the XML document fragment defined by the XML map, with the indicated substitution.

There is an additional tag, @jws:jms−header, that can format message headers. All three of these tags work the same way.

The onNameReply callback uses an XML map defined with the @jws:jms−message tag to extract data from the incoming XML message and transfer it to the Java parameters of the callback.

To learn more about the @jws:jms−header tag, see @jws:jms−header Tag.

To learn more about the @jws:jms−property tag, see @jws:jms−property Tag.

To learn more about the @jws:jms−message tag, see @jws:jms−message Tag.

Related Topics

Overview: Messaging Systems and JMS

Sending and Receiving XML Messages with a JMS Control

Manipulating JMS Message Headers and Message Properties in a JMS Control

SimpleJMS.jws Sample

CustomJMSClient.jws Sample

@jws:jms Tag

Manipulating JMS Message Headers and Message Properties in a JMS Control

This topic describes how to use XML maps to translate between the Java objects that are method and callback parameters in the JMS control's interface and the JMS message headers and message properties on messages sent and received by the control.

To learn about WebLogic Workshop controls, see Controls: Using Resources from a Web Service.

To learn about JMS controls, see JMS Control: Using Java Message Service Queues and Topics from Your Web Service.

To learn how to manipulate the XML format of the message body, see Sending and Receiving XML Messages with a JMS Control.

XML Messages and JMS Controls

A JMS control can send and receive text messages, XML messages, object messages or javax.jms.Message objects. While XML maps may be used to specify the message body only in JMS controls with message type "XML Map", XML maps may be used to control the message headers and message properties on JMS controls of any message type (Text, XML Map, Object or JMS Message).

XML maps are used throughout WebLogic Workshop to convert XML messages to Java objects and vice versa. To learn about general XML map concepts, see Why Use XML Maps?

Accessing Message Headers with XML Maps

You may access the headers of messages sent and received using the JMS control by using the @jws:jms−header tag.

You may use XML maps on methods to add headers to outgoing messages, optionally substituting the values of parameters to the method in the header. You may also use XML maps on the JMS control's callback to extract headers from incoming messages and assign the extracted values to the callback's parameters.

The example below demonstrates use of the @jws:jms−header tag to set and extract message headers.

Accessing Message Properties with XML Maps

You may access the properties of messages sent and received using the JMS control by using the @jws:jms−property tag.

You may use XML maps on methods to add properties to outgoing messages, optionally substituting the values of parameters to the method in the property. You may also use XML maps on the JMS control's callback to extract properties from incoming messages and assign the extracted values to the callback's parameters.

The example below demonstrates use of the @jws:jms−property tag to set and extract message properties.

The Edit Maps and Interface Dialog

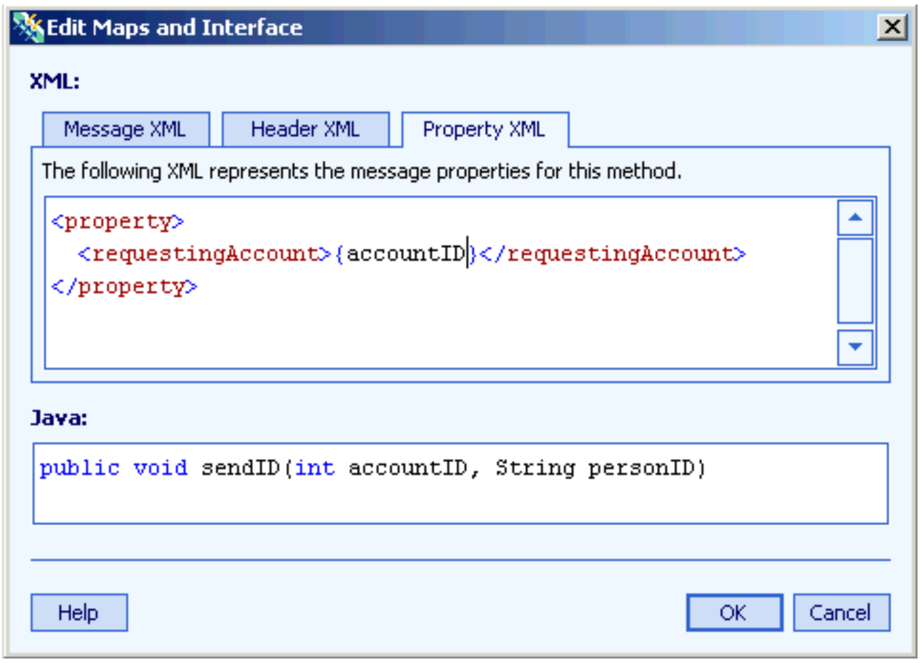
The Edit Maps and Interface Dialog allows you to customize the parameter list and associated XML maps for a JMS control method or callback. To display the Edit Maps and Interface Dialog, double−click on the map icon associated with the method or callback you wish to customize.

The map icon for a JMS control method is shown here:

The map icon for a JMS control callback is shown here:

The Edit Maps and Interface dialog for the JMS control contains three tabbed panes that allow you to edit the XML maps for the message body (setting the @jws:jms−message tag's XML map), the message headers (setting the @jws:jms−header tag's XML map) and the message properties (setting the @jws:jms−property tag's XML map).

The Property XML pane of the Edit Maps and Interface Dialog for a JMS control method is shown here:



The following XML represents the message properties for this method.

```
<property>
  <requestingAccount>{accountID}</requestingAccount>
</property>
```

```
public void sendID(int accountID, String personID)
```

Example

The example below demonstrates setting and extracting both message headers and message properties as well as specifying the XML format of the message body:

```
/**
 * @jws:jms type="queue"
 *          send-jndi-name="OutQueue"
 *          receive-jndi-name="InQueue"
 */
public interface GetPersonControl extends JMSControl
{
    /**
     * @jws::jms-header xml-map::
     *   <header>
     *     <command>GetPerson</command>
     *   </header>
     * ::
     *
     * @jws::jms-property xml-map::
     *   <property>
     *     <requestingAccount>{accountID}</requestingAccount>
     *   </property>
     * ::
     *
     * @jws:jms-message xml-map::
     *   <personIdentifier type={idType}>
     *     <identifier>{personID}</identifier>
     *   </personIdentifier>
     * ::
     */
    public void sendID(int accountID, String personID, String idType);

    public static interface Callback extends JMSControl.Callback
    {

        /**
         * @jws::jms-header xml-map::
         *   <header>
         *     <command>{command}</command>
         *   </header>
         * ::
         *
         * @jws::jms-property xml-map::
         *   <property>
         *     <requestingAccount>{accountID}</requestingAccount>
         *   </property>
         * ::
         *
         * @jws:jms-message xml-map::
         *   <person>
         *     <firstname>{fname}</firstname>
         *     <lastname>{lname}</lastname>
         *   </person>
         * ::
         */
        public void onNameReply(String receivedCommand, int accountID, String fname, String lna
    }
}
```

The sendID method will send a message whose body is an XML document with a <personIdentifier> top−level element. The value of sendID's personID parameter is substituted as the value of the <identifier> element. The personIdentifier element will also include an attribute type whose value is that of the idType method parameter.

The message will include a message header named command with the (hard−coded) value GetPerson. The message will also include a message property named requestingAccount, whose value will be the that of sendID's accountID parameter.

When a message is received by this JMS control, the onNameReply callback will be invoked. If the incoming message includes a header named command, the value of the header will be assigned to the receivedCommand parameter of onNameReply. If the incoming message contains a property named requestingAccount, the value of that property will be assigned to the accountID parameter of onNameReply. The values of the <firstname> and <lastname> elements in the message body will be assigned to the fname and lname parameters of onNameReply. Then onNameReply will be invoked with the newly assigned parameters.

Related Topics

Overview: Messaging Systems and JMS

SimpleJMS.jws Sample

CustomJMSClient.jws Sample

@jws:jms Tag

@jws:jms−message Tag

@jws:jms−header Tag

@jws:jms−property Tag

Using an Existing JMS Control

This topic describes how to use an existing JMS control in your web service.

To learn about controls, see the Controls Overview.

To learn about JMS controls, see JMS Control: Using Java Message Service Queues and Topics from your Web Service.

To learn how to create a JMS control, see Creating a New JMS Control.

Using an Existing JMS Control

All controls follow a consistent model. Therefore, most aspects of using an existing JMS control are identical to using any other existing control.

To learn about using an existing control, see Using an Existing Control.

Related Topics

Overview: Messaging Systems and JMS

Control Factories: Managing Collections of Controls

This topic describes control factories, which are a way to dynamically manage a collection of instances of a control from a web service.

You should first understand WebLogic Workshop controls before reading this topic. To learn about WebLogic Workshop controls, see Controls: Using Resources from a Web Service.

What Is a Control Factory?

A control factory allows a single web service to manage an n−way relationship with a control. For example, a web service can disassemble the line items of an incoming purchase order and conduct a concurrent conversation with a separate Service control for each of multiple vendors.

Automatically Generated Factory Classes

For any control interface called MyControl, WebLogic Server generates a control factory interface called MyControlFactory that has the following very simple shape:

```
interface MyControlFactory {    MyControl create(); }
```

The implicit factory class is located in the same package as the control class. E.g., if the full classname of the control interface is com.myco.mypackage.MyControl then the full classname of the factory is com.myco.mypackage.MyControlFactory. An automatic factory class is not generated if there is a name conflict (i.e., if there is already an explicit user class called MyControlFactory.)

A control factory instance can be put into a JWS file just like a control instance, with the same Javadoc preceding the factory declaration that would precede a single control declaration.

For example, an ordinary Service control would be declared as follows:

```
   /**    * @jws:control    */   MyServiceControl oneService;
```

While a Service control factory would be declared as follows:

```
   /**    * @jws:control    */   MyServiceControlFactory manyServices;
```

Note again that the set of annotations allowed and required on a factory are exactly the same as the set of annotations on the corresponding control. The factory behaves as if those annotations were on every instance created by the factory.

Once a web service includes a control factory declaration, a new instance of a single control can be created as follows:

```
   // creates one control   MyServiceControl c = manyServices.create();

  // then you can just use the control, store it, or whatever.   c.someMethot();

  // For example, let's associate a name with the service...   serviceMap.put("First Service", c);
```

Factory classes are automatically generated on−demand, as follows. When resolving a class named FooFactory:

1. First the class is resolved normally, i.e., if there is a CLASS file or JAVA file or CTRL file that contains a definition for FooFactory, then the explicitly defined class is used.
2. If there is no explicit class FooFactory, then, since the classname ends in "Factory", we remove the suffix and look for an explicit class called Foo (in the same package).
3. If Foo is found but does not implement the Control interface (i.e., is not annotated with @jws:control), it's considered an error (as if Foo were never found).
4. However, if Foo is found and implements the Control interface, then the interface FooFactory is automatically created; the interface contains only the single create() method that returns the Foo class.

All instances of the control are destroyed when the web service instance that created them is destroyed.

Parameterized Callback Handlers

Since there may be multiple controls that were created with a single control factory, and they all have the same instance name, a mechanism is provided to enable you to tell which instance of the control is sending a callback.

For example, for the oneService example above, an event handler still has the following form:

```
    void oneService_onSomeCallback(String arg)    {        System.out.println("arg is " + arg);    }
```

For callback handlers that are receiving callbacks from factory−created control instances, the callback handler must take an extra first parameter that is in addition to the ordinary parameters of the callback. The first parameter is typed as the control interface, and the control instance is passed to the event handler.

The manyServices factory callback handler looks like this:

```
   void manyServices_onSomeCallback(MyServiceControl c, String arg)   {       // let's retrieve the
remembered name associated with the control       String serviceName = (String)serviceMap.get(c);

      // and print it out       System.out.println("Event received from " + serviceName);   }
```

Related Topics

[Controls: Using Resources from a Web Service]

[ServiceFactoryClient.jws Sample]

Using Asynchrony to Enable Long−Running Operations

A typical non−distributed software application uses [synchronous] methods between application components. The caller of a synchronous method is blocked from further execution until the method returns. Web services, however, run over networks. The distributed nature of applications using web services introduces unpredictable and sometimes very long latency, meaning the time it takes a particular operation to complete may be long. Some business processes represented by web services may involve human interaction at the back end, meaning an operation may take on the order of days. If all web service interactions were synchronous, each client with an outstanding operation would be consuming resources on its host system for unacceptably long periods of time.

WebLogic Workshop provides facilities that make it easy for you to build asynchronous web services. Using these facilities, you can design web services that don't require clients to block execution waiting for results. You can choose from multiple approaches for returning results to your web service's clients.

Topics Included in This Section

[Asynchronous Web Services]

You can build web services that allow a client to initiate a request, then receive notification from your web service at a later time when results are ready. This is accomplished by using [callbacks]. You can also simulate an asynchronous web service by providing a polling interface to clients that cannot accept callbacks (see below).

[Using Callbacks to Notify Clients of Events]

Callbacks are used to notify a client of your web service that some event has occurred. For example, you may wish to notify a client when the results of that client's request are ready.

[Asynchronous Methods]

Asynchronous methods and callbacks return immediately to the caller, allowing the calling web service or application to continue other processing (or become dormant) until the result of the operation is complete. Asynchronous methods and callbacks are also referred to as being buffered, because the asynchrony is accomplished using message buffers. Buffering methods and callbacks can help your web services handle high loads.

Using Polling as an Alternative to Callbacks

Some clients of your web services may not be capable of handling callbacks. Callbacks are messages that your web service sends to the client, and many systems on which clients run are configured so that they will reject all "unsolicited" incoming traffic. In order to allow clients that operate in this environment to use your web services, you can supply a polling interface as an alternative. In a polling interface, you provide one or more methods that a client can call periodically to determine whether the result of a previous request is ready.

Related Topics

The Web Services Development Cycle

Maintaining State with Conversations

Conversation.jws Sample

Asynchronous Web Services

If a web service you are designing exposes operations that may be long−running, you should design the interface of your web service such that clients do not have to wait in a blocked state for the long−running operations to complete. You do this by making your web service asynchronous. In your design, you provide one or more methods that accept requests from clients but that return quickly. You also provide a mechanism for the client to obtain the results of the long−running operation when the results are ready.

It's important to distinguish between an asynchronous web service and an asynchronous method (a method that always returns immediately, and always returns void). A web service may be asynchronous without using asynchronous methods. An example is a web service that provides a synchronous request operation that returns an acknowledgement but not the result, and later calls a synchronous callback that sends the result and receives an acknowledgement from the client. Both the method and the callback are synchronous, but the web service is asynchronous. To learn more about asynchronous methods and callbacks, see Asynchronous Methods.

There are two ways to design an asynchronous web service:

- Implement methods that initiate requests and callbacks to send results.
  To learn more about callbacks, see Using Callbacks to Notify Clients of Events.
- Implement methods that initiate requests and additional methods that return request status ("pending" or "complete"). The second approach is referred to as a polling interface.
  To learn more about implementing a polling interface, see Using Polling as an Alternative to Callbacks.

You may want to implement both approaches in your web service. Doing so would provide the convenience of callbacks to those clients who can handle them, and a polling interface for clients who cannot accept callbacks.

Related Topics

Building Web Services with WebLogic Workshop

Using Asynchrony to Enable Long−Running Operations

Maintaining State with Conversations

Using Callbacks to Notify Clients of Events

Callbacks are used to notify a client of your web service that some event has occurred. For example, you may wish to notify a client when the results of that client's request are ready.

You can easily add callbacks to your web service's interface using the Add Callback action in Design View. However, you should be aware that doing so places demands on the client that not all potential client applications will be able to meet.

Callbacks Are Messages to the Client

When you add a method to a web service, you are defining a message that a client can send to you to request that some operation be performed. When you define a callback, you are defining a message that the web service will send to the client to notify the client of an event that has occurred in your web service. These are completely symmetrical. WebLogic Server allows your web service to receive XML and SOAP messages and will route them automatically to your web service. In order to receive callbacks, the client must be operating in an environment that provides the same services. This typically means the client must be running in an application server or web server.

If the client is not running in an environment that provides the necessary infrastructure, it will likely not be capable of receiving callbacks from your web service.

Note: The notion that callbacks are messages to the client is important if you want to apply XML maps to callback parameters or return values. The parameters to a callback go into the outgoing message to the client, and the return value is converted from the resulting incoming message from the client. This can seem strange, because programmers typically associate all parameters with incoming data and return values with outgoing data.

Callbacks Can Appear to be Unsolicited

As stated in the previous section, callbacks are messages to the client. Since callbacks are by definition separated from the original request to which the callback is a response, they appear as unsolicited messages to the client's host. Many hosts do not allow receipt of unsolicited network traffic, either by directly rejecting such traffic or by nature of being protected by firewalls or other network security apparatus. Clients that run in such an environment will therefore not be capable of receiving callbacks.

Callback Protocol is Inferred

The protocol and message format used for callbacks is always the same as the protocol and message format used by the start method that started the current conversation. It is an error to attempt to override the protocol or message format of a callback.

What If Clients of My Web Service Can't Receive Callbacks?

If clients of your web service cannot receive callbacks for the reasons described in the preceding sections (or for any reason), but you still want to design your web service to be asynchronous, you may implement a polling interface.

To learn about asynchronous web services, see Asynchronous Web Services.

To learn about polling interfaces, see Using Polling as an Alternative to Callbacks.

Related Topics

Using Asynchrony to Enable Long−Running Operations

Maintaining State with Conversations

Asynchronous Methods

Asynchronous methods and callbacks return immediately to the caller, allowing the calling web service or application to continue other processing until the result of the operation is complete. Asynchronous methods and callbacks are also referred to as being buffered, because the asynchrony is accomplished using message buffers. Buffering methods and callbacks can help your web services handle high loads.

Message buffers may only be added to methods and callbacks that have a void return type. When a method or callback with a message buffer is invoked, the message representing the method or callback invocation is placed in a queue to be delivered to the web service or client when it is available. Since the queue has no information about the semantics of the message or the destination software component, it cannot return anything meaningful to the caller. Since the queue must always return void, any methods or callbacks that buffered must return void.

You may add message buffers to the methods and callback handlers of a web service. It is important, however, to understand what happens when you add message buffers. The queues that provide the buffering always exist on your end of the wire. WebLogic Server has no way to cause configuration actions like message buffering in a client's infrastructure. Design View reinforces this by drawing the message buffer "spring" icons on the end of the wire closest to your web service.

Adding a message buffer to a method makes the method asynchronous, meaning that callers to that method do not wait for a response.

Message Buffers on Web Service Methods

If you add a message buffer to a method of your web service, incoming messages (invocations of the method) are buffered on the local machine. The method invocation returns to the client immediately. This prevents the client from waiting until your web service processes the request. Note that since the buffering occurs on your end of the wire, the client still must wait for the network roundtrip even though it will return a void result. But the client does not have to wait for your service to process the message.

Message Buffers on Web Service Callbacks

In the case of a callback, your web service is acting as the client. Your web service is sending an outgoing messages (callback invocation) to the client, which may respond with a incoming message containing the callback's return value.

If you add a message buffer to a callback, outgoing messages (invocations of the callback) are buffered on the local machine. The callback invocation returns immediately. This prevents your service from having to wait while the message is sent to the remote server and the (void) response is received. In other words, your service doesn't have to wait for the network roundtrip to occur.

The message−buffer Property (@jws:message−buffer Javadoc Tag)

You can easily add a message buffer to a method or callback in Design View. Select the method or callback you want to buffer. Then navigate to the message−buffer property in the Properties editor and set the enabled attribute to true. This will add the Javadoc tag @jws:message−buffer enabled−true to the Javadoc comment preceding the method or callback in the source code.

All Parameters Must Be Serializable

All objects that are passed as method parameters to buffered methods or callbacks must be serializable. Method and callback invocations are placed in the queue through Java serialization.

Many built in Java classes are already Serializable. Most other classes can be made serializable very easily by adding implements java.io.Serializable to the class declaration.

For example, if the Person class below is passed as a method parameter, just add the bold text to the declaration to allow the class to be passed as a parameter to buffered methods or callbacks.

```
public static class Person implements java.io.Serializable
{
    public String firstName;
    public String lastName;
}
```

Execution Order of Methods

When a web service defines buffered (asynchronous) methods, neither the web service designer or the client can make any assumptions about execution order of methods. This is especially true when considering buffered and non−buffered methods in the same web service.

Note that invocations of synchronous methods are guaranteed to be in order of arrival. But if buffered methods are also defined in the same web service, you cannot determine when their invocations will occur relative to the synchronous methods.

Controlling Retry Behavior

From within a buffered method, you may control the retry behavior of WebLogic Server with respect to this method. If the method is invoked, but you do not wish to service the request immediately, you may request that WebLogic Server reissue the request at a later time. You do this by throwing a weblogic.jws.RetryException from within the web service method.

The constructors for the RetryException class take a retry delay as an argument:

RetryException("Try sooner", 5)

The second argument is a long, specifying the number of seconds to delay before retrying.

RetryException("Try later", "5 days")

The second argument is a String specifying the delay before retrying.

RetryException("Try default", RetryException.DEFAULT_DELAY)

Passing the DEFAULT_DELAY constant specifies that the value of the retryDelay attribute of the @jws−message−buffer tag should be used.

Related Topics

Guide to Building Web Services

Using Asynchrony to Enable Long−Running Operations

Asynchronous Web Services

Maintaining State with Conversations

Using Polling as an Alternative to Callbacks

Some clients of your web services may not be capable of handling callbacks. To learn about situations in which a client may not be able to receive callbacks, see Using Callbacks to Notify Clients of Events.

In order to allow clients that can't accept callbacks to use your web services, you can supply a polling interface as an alternative. In a polling interface, you provide one or more methods that a client can call periodically to determine whether the result of a previous request is ready.

Here is an example taken from the Conversation.jws Sample web service:

```
public class Conversation
{
    /**
     * @jws:operation
     * @jws:conversation phase="start"
     */
    public void startRequest()
    {
        ...
    }



    /**
     * @jws:operation
     * @jws:conversation phase="continue"
     */
    public String getRequestStatus()
    {
        ...



    }



    /**
     * @jws:operation
     * @jws:conversation phase="finish"
     */
    public void terminateRequest()
    { }
}
```

A client uses the startRequest method to initiate a request. The client may then call getRequestStatus periodically to check on the result. Between calls to getRequestStatus, the client it free to perform other processing. getRequestStatus returns an indication that the request is "pending" until the request is complete. The next time the client calls getRequestStatus after the request is complete, the result will be returned to the client. The client then calls terminateRequest to finish the conversation.

Related Topics

Web Services Development Cycle

Using Asynchrony to Enable Long−Running Operations

Maintaining State with Conversations

Maintaining State with Conversations

Some web services need to communicate with a client multiple times in connection with a single task. Conversations ensure that the communications are kept straight amid the many requests that make up the communication. When a particular task requires conversations with multiple parties, especially in an asynchronous manner, conversations provide a way to store and remember intermediate results until the task has been completed.

Topics Included in This Section

Overview: Conversations

Provides an introduction to conversations and what you can accomplish with them.

How Do I: Add Support for Conversations?

Provides a step−by−step procedure for setting the property that specifies conversation support.

Life of a Conversation

Describes an example of a web service in a conversation.

Implementing Conversations

Offers details and guidelines related to building a web service that supports conversations.

Managing Conversation Lifetime

Describes how you can write code that controls and responds to aspects of a conversation.

Supporting Serialization

Provides an introduction to serialization.

Samples

The following sample projects include samples that use conversations:

- async

- creditreport
- database
- service
- jms
- timer
- xmlmap

For more information about samples, see Samples.

Related Topics

Using Asynchrony to Enable Long−Running Operations

Overview: Conversations

For web services that may communicate with a client multiple times in connection with a single task, conversations ensure that the communications are kept straight. In addition, web services that communicate asynchronously use conversations to maintain state across asynchronous messages. Conversations meet two challenges inherent in persisting data across multiple communications with a client or resource (such as another service or a data source):

- Maintaining state, or the data held by a web service for a specific task. This is essential when communications are asynchronous.
- Associating the actions a service performs (including its calls to other resources) and its response to a client with the client's original task.

How Conversations Work

A web service participating in a conversation works within a specific context created by WebLogic Server. Using the unique identifier that is part of this context, WebLogic Server saves the service's state and correlates adov−−> that is, tracks communication between the service, its client, and resources used by the service.

State Persistence Through Serialization

State refers to the condition of your web service at a given time, and is made up of the data it holds. Consider the Investigate sample web service (described in Tutorial: Your First Web Service). In that example, the Investigate service takes a taxpayer ID from a client and uses this number to gather credit information. The service stores the taxpayer ID in a member variable so it can retrieved as needed. The taxpayer ID in this case is an example of state−related data — data that makes up the condition of the service while it is processing.

Now imagine that the computer on which the service was running suddenly shut down. If the taxpayer ID were held only by the member variable in the service instance, it would be lost when the instance vanished from memory as the computer shut down.

To ensure that state−related data remains safe through such events, you can specify that calls to the service are part of a conversation. When a conversation with a service begins, WebLogic Server creates a context in which to keep track of the service's state−related data. Data that is part of a service's context is preserved through serialization, a process through which the data can be written to the hard disk.

Correlated Messages Through a Unique Identifier

The context created for a service that participates in conversations includes more than data stored in member variables. It also includes the identity of the client with which the service is conversing and information tying that client to the task begun with the client's initial request. A web service may begin thousands of tasks for client requests over a period of time, from the same and different clients. With multiple communications, a service needs a way to remember which of the potentially thousands of requests its response belongs to.

When the various actions for processing a task are associated with the original request, this is an example of correlation. For each new conversation started with a service — in other words, for each new context — WebLogic Server generates a unique identifier called a conversation ID. The conversation ID is used to correlate the original request (and its client) with subsequent communications with other web services, queries to databases, and exchanges with other resources. Each of these actions and its effect on state−related data is correlated with the original request using the conversation ID so that the client receives a response associated with the original request.

Beginning, Middle and End

As described in Life of a Conversation, conversations for web services have a life of their own. They start when a client calls a web service using a method that is marked to "start" a conversation, they move through methods and callbacks marked to "continue" the conversation, and they end with a call to a method or callback marked to "finish" the conversation. The call to the first method creates a context for the conversation, and subsequent calls continue the conversation, serializing the updated state information so that the context remains current. The last call finishes the conversation, prompting WebLogic Server to release any resources and state data it was holding on behalf of the web service.

How to Build Services That Support Conversations

In practice the processing of adding support for conversations is often as simple as setting the right property for each of the service's operations (methods and callbacks it exposes to other components). For more detailed information a few design suggestions, see Implementing Conversations.

For a step−by−step procedure on adding support for conversations, see How Do I: Add Support for Conversations. You will also find a hands−on introduction to conversations and other WebLogic Workshop technologies in Tutorial: Your First Web Service.

Testing with Conversations

When you test a web service that supports conversations, Test View displays the conversation ID in the Message Log. The Message Log also group the actions associated with a conversation. As you test the service multiple times, each test is added to the Message Log until you clear the log. You can also use Test View to clear conversations.

For more information about Test View, see Test View. For step−by−step information on using Test View, see How Do I: Test A Web Service Using Weblogic Workshop's Test View?

Related Topics

How Do I: Add Support for Conversations?

Tutorial: Your First Web Service

Implementing Conversations

How Do I: Add Support for Conversations?

You can use Design View in WebLogic Workshop to add support for a conversation by choosing the methods, callbacks, and callback handlers that will participate in the conversation, and then setting the conversation phase attribute for those items.

Note that adding conversation support assumes that state−related data is held in member variables whose types support serialization. WebLogic Workshop uses serialization to save state−related data between communications with clients and resources. In most cases, you may find that supporting serialization requires no extra effort on your part. For more information, see Supporting Serialization.

To Add Support for Conversations in Design View

1. In Design View, click the method that will begin this conversation.
2. In the Properties pane, expand the conversation property.
3. From the drop−down list corresponding to the phase attribute, select start.
4. For each method and callback that will participate in the conversation, set the conversation phase attribute to continue.
5. Click the method or callback that will finish this conversation, then set the conversation phase attribute to finish.

Related Topics

Overview: Conversations

Life of a Conversation

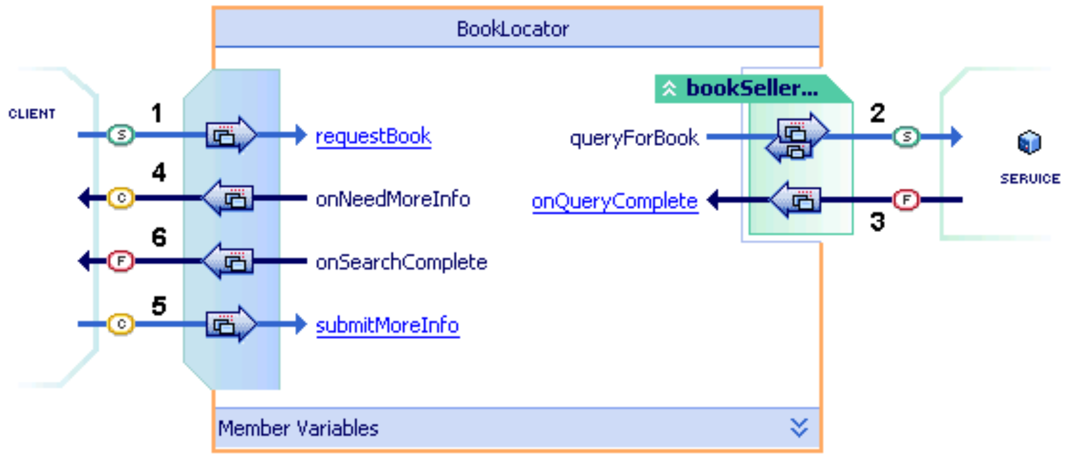
A conversation has a beginning, a middle, and an end. When you design a service that uses conversations you need to know the role for each method and callback in communicating with clients and other resources.

Not only do you set the conversation phases (start, continue, and finish) for the methods and callbacks, you can set service−level properties such as conversation lifetime (max−age, max−idle−time).

When you add support for conversations, you annotate your service's interface adov−−> the methods and callbacks it exposes to the outside world adov−−> with the conversation property's phase attribute. You can do this in Design View using the Properties pane. When you set this property for a method or callback, WebLogic Workshop adds an icon indicating whether the item starts, continues, or finishes a conversation:

Ⓢ Ⓒ Ⓕ

The following example describes the life cycle of a service as it uses two conversations adov−−> one to respond to a client request and another to communicate with a Service control. The BookLocator service searches a network of other book sellers on behalf of a customer. Each of the numbers in the following illustration corresponds to a stage in the life cycle description that follows. Stages 1, 4, 5, and 6 describe the conversation between the BookLocator service and the client. Stages 2 and 3 describe the conversation between the BookLocator service and the bookSeller control. The code within the BookLocater service acts as the middleman and exchanges information between the two conversations. It’s as if the first conversation begins when the client speaks to the BookLocater, requesting a book. The BookLocater then turns to the bookSeller service to ask about the specific title. When the bookSeller service responds to the BookLocater this completes their conversation. The BookLocator turns back to the client who’s been patiently waiting and resumes that first conversation.



1. The client submits a search request by calling the requestBook method, passing information about the book and its author.

Because this method is marked as starting a conversation, WebLogic Server creates a new context for the service and associates it with a unique identifier, called a conversation ID. The conversation ID will be used to correlate further interactions between the client and the service, ensuring that calls back to the client and calls from other controls are correctly associated with this particular instance of the service.

The requestBook method returns void immediately, allowing the client to continue on its way. Such exchanges adov−−> a void return with the promise to call back when the response is ready adov−−> are an essential part of asynchronous communications.

As the method returns, WebLogic Server starts the conversation's idle and age timers, which can later be used to finish the conversation for lack of activity (provoke a timeout); at the same time, the service's state−related data is saved.

2. Code executing as part of the requestBook method's implementation calls the bookSellerControl's queryForBook method.

Because this method begins a conversation implemented by the bookSeller service, WebLogic Server creates a new context for that service; in other words, the call to this method is managed within its own context. However, even though each service context is separate, with separate conversation IDs and separate state−related data, WebLogic Server ensures that their communications are correctly correlated.

3. After the book seller has searched its inventory for the requested book, the bookSeller service returns its results using the onQueryComplete callback that is part of its interface. This callback finishes the bookSeller service's conversation with the BookLocator service, and its context is released by WebLogic Server adov−−> along with any resources it may have held to perform its service. The conversation context for the BookLocator service remains active.

The response is handled by the BookLocator service through the onQueryComplete callback handler. As an added value to clients, the folks behind the BookLocator service examine the data returned by the book seller they have queried to determine if it is likely to be useful to the client. In this case, the data is a long list of books of varying condition and price, so the book locator decides to request more information from the client in order to narrow the list.

4. The BookLocator requests more information through the onNeedMoreInfo callback. Because the callback is part of its interface, the client software is prepared to handle it, accepting parameters that indicate the sort of information that would help the BookLocator shorten the list of possible matches.

The onNeedMoreInfo callback is annotated to "continue" the conversation with the client. This ensures that the callback is sent in response to the correct request. Even if multiple requests to the BookLocator service have been made by multiple clients, the state of each is kept safe by its context.

5. Eventually, the client responds with more information, passing that information back to BookLocator as parameters of the submitMoreInfo method. Of course, the call to submitMoreInfo, like the callback made to the client, is annotated to continue the conversation.
6. BookLocator uses the additional information to shorten the list, then uses another callback, onSearchComplete, to send the short list to the client. This callback is annotated to "finish" the conversation. Once the callback has executed, the context created with a call to the start method is eligible to be removed by WebLogic Server.

Related Topics

How Do I: Add Support for Conversations?

Implementing Conversations

Implementing Conversations

This topic provides information about building web services that support conversations. Conversations are part of a technology through which you can ensure that WebLogic Server maintains your service's state−related data and correlates communications between your service, clients, and resources such as other services.

You should consider supporting conversations in any service design that is asynchronous or involves multiple communications with a client in connection with a single request. Services that support conversations are treated differently by WebLogic Server. When a client calls a service operation that is annotated to start a conversation, WebLogic Server creates a conversation context through which to correlate calls to and from the service and to persist its state−related data.

Understanding Conversation Context

Context is the key to the benefit provided by conversations. Conceptually speaking, multiple related calls to or from a single web service are part of the same context by virtue of being related to the same initial request. With WebLogic Workshop, you can easily build services that support this conceptual reality by using conversations.

When a conversation starts, WebLogic Server does the following:

- Creates a context through which to maintain the scope of the conversation.
- Generates a conversation ID with which to identify the context.
- Saves the service's state, including any data stored in serializable member variables.
- Starts an internal timer to measure idle time.
- Starts an internal timer to measure the conversation's age.

Each piece of information — the conversation ID, persistent data, idle time and age — are part of the conversation's context. In particular, the conversation ID is passed back to the client in the XML message through which web services communicate. A conversation ID is also passed in calls to resources. With each subsequent conversational call to the web service, the conversation ID is used to associate the call with the correct conversation context — in other words, the correct instance of the service, along with whatever state it may currently hold.

Notes About Conversation Context

As you build services that support conversations, there are a few characteristics of conversations you should keep in mind:

- The scope of conversation context is limited to the service itself.
  For example, you cannot assume that the state of another service is being persisted simply because your service calls one of its methods during the course of a conversation. The other service must be responsible for its own state maintenance, with or without conversations.
- Correlation between two services that support conversations is handled automatically by WebLogic Server.
  In other words, if your service supports conversations and calls the conversational methods of another service that supports conversations, WebLogic Server will correlate the two separate contexts automatically.
- Unless you know otherwise, it is not a good practice to assume that conversations will be correctly correlated between your service and one that appears not to support conversations.
- Service state is only updated on the successful completion of methods or callback handlers that are marked with the conversation phase attributes "start," "continue" or "finish."

Note that this excludes internal methods of your service (which are not operations and so can not be conversational) and methods exposed by other web services.

Implementing Conversational Methods and Callbacks

In practice, adding support for conversations is typically a matter of annotating methods and callbacks with the conversation phase attribute (discussed in the following section). Even so, as you write code for methods and callback handlers that will be annotated for use in conversations, you may want to keep the following suggestions in mind.

- Be sure to handle exceptions thrown from conversational methods and callback handlers.
  An unhandled exception thrown from a conversational method or callback handler will cancel any update of state from that method or handler (in other words, state data will not be serialized). As you plan exception handling for conversational methods and callback handlers, you should consider what an appropriate response to the exception is. The JwsContext interface provides an onException callback handler that is invoked whenever an exception is thrown from an operation (a method marked with the @jws:operation tag) or callback handler. Depending on your service's design, your code in the onException handler might:

- Log the error in an application log.
- Notify the client of the failure.
- Finish the conversation.
- Conversational methods and callbacks should be short−lived.
  Only one method or callback may be executing within a conversation at a given time. On one hand, this is a benefit in that it means you needn't worry about a new call interfering with one that is already being processed. On the other hand, because new calls will be blocked until the current one finishes executing, you should try to implement each method or callback so that it executes in a short amount of time.

- Ensure serialization support for variables that will hold state−related data.
  In Java, serialization is a technology that makes it possible to write information about an object instance to disk. This requirement can in most cases be met with no extra effort. For more information, see Supporting Serialization.

- Do not save state from within code that is not part of a conversation.
  If your service must save state, you should always try to do so by writing to member variables from within a method that starts or continues a conversation.

Applying the Conversation Phase Attribute

Applying a conversation phase attribute to a method or callback identifies it as having a role in conversations. In Design View, you can apply the attribute using the Properties pane. When you set the conversation phase property, a corresponding tag like the one below is added to your service's source code immediately preceding the method or callback declaration:

```
@jws:conversation phase="start"
```

Possible values for the conversation phase attribute are as follows:

- start adov−−> May be applied to methods only. Specifies that a call to the method starts a new conversation. Each call will create a new conversation context and an accompanying unique conversation ID, save the service's state, and start its idle and age timer.
- continue adov−−> May be applied to methods and callbacks. Specifies that a call to this method or callback is part of a conversation in progress. WebLogic Server will attempt to use the incoming conversation ID to correlate each call to an existing conversation. Each call to a "continue" method will save the service's state and reset its idle timer.
  Set the phase attribute to "continue" for any method or callback that is likely to be used for communication with a client in connection with an ongoing request adov−−> methods that are clearly intended to be called subsequent to the conversation's start and before its finish. These include requests for or responses with additional information, requests for progress or status, and so on.
- finish adov−−> May be applied to methods and callbacks. Specifies that a call to this method or callback finishes an existing conversation. A call will mark the conversation for destruction by WebLogic Server. At some point after a finish method returns successfully, all data associated with the conversation's context will be removed.
  It is also possible to finish a conversation by calling the JwsContext interface finishConversation method. For more information, see Managing Conversation Lifetime.
- none adov−−> May be applied to methods and callbacks. Specifies that a call to this method or callback has no meaning in the context of the conversation.

For more information adding support for conversations with the conversation phase attribute, see How Do I: Add Support for Conversations?

Notes About Setting Conversation Phase

Unless you know how a given control is implemented, it is not a good practice to apply the conversation phase attribute to methods and callbacks exposed by controls.

In other words, you should ordinarily not change the conversation phase attribute for items that appear on the right side of the design view. When you import the control (CTRL) file corresponding to a resource such as another service, its image in Design View will indicate how and whether it supports conversations. Because the control is merely a proxy for the resource itself, "changing" its conversation support may result in unpredictable behavior.

Related Topics

JwsContext Interface

@jws:conversation Tag

Managing Conversation Lifetime

For a service participating in a conversation, aspects of the conversation adov−−> including how old it is and how long it has been idle adov−−> are maintained by WebLogic Server. Using methods and callbacks exposed by the JwsContext interface, you can write code to control and respond to these aspects at run time.

This can be useful when you want to provide logic that changes the values controlling conversation lifetime according to specific actions your service performs. For example, you might extend maximum age or idle time (or both) when you anticipate that certain processes will take a long time to complete; you might also finish a conversation immediately.

You can also write code that executes when a conversation is about to end, giving you an opportunity to clean up after your service, releasing acquired resources or notifying the client that the conversation is about to end.

When you create a new web service (JWS file) your source code automatically includes the following:

```
/** @jws:context */
weblogic.jws.control.JwsContext context;
```

The @jws:context Javadoc tag specifies that a context should be created within which to manage state and correlate calls for the web service. The declaration of a JwsContext variable is provided so that you can write code calling methods and handling callbacks exposed by the JwsContext interface.

Note: With the JwsContext variable in your code, you will be able to view a list of the interface's methods and callbacks in the WebLogic Workshop Structure Pane.

Using this JwsContext variable, you can manage conversation lifetime by:

- Controlling how long a conversation may remain idle
- Limiting a conversation's duration
- Finishing a conversation
- Doing something when a conversation finishes

Controlling How Long a Conversation Can Remain Idle

Maximum idle time is the amount of time that can go by between incoming messages before your service's conversation automatically ends and WebLogic Server removes it from memory.

You can write code to change the maximum amount of time for which a conversation may remain idle before it automatically finishes. You might write code to change the maximum idle time if you want to allow more time for a client to complete some process. For example, you might temporarily increase the maximum idle time if your code makes an asynchronous follow−up request to that client and you suspect that it will take longer than the maximum idle time to respond; when the client does respond, your code can update the maximum idle time back to a shorter duration.

Note: The timer keeping track of idle time is only reset by interactions with a client. For example, it is not reset when your service receives a callback from another web service (in other words, from entities on the right side of your service in Design View). To reset the idle timer in such circumstances, you can call the resetIdleTime method of the JwsContext interface in your callback handler.

Setting the Initial Maximum Idle Time Value at Design Time

You can set the initial maximum idle time in Design View by clicking the service area, then using the Properties pane to set the conversation−lifetime max−idle−time attribute.

By default, the value of this attribute is "0", meaning that the conversation may remain idle indefinitely. In general, it is a good practice to estimate a reasonable amount of time for your service to remain idle, then set this attribute to some other value. Allowing a conversation to remain idle for a long period of time can absorb system resources, particularly as new conversation contexts are created for new calls to the service (each with indefinite idle timeout values!).

When you change the max−idle−time attribute, WebLogic Workshop will annotate your service class with a Javadoc tag resembling the following (this example sets the value to "2 minutes").

```
/**
 * @jws:conversation−lifetime max−idle−time="2 minutes"
 */
```

The value specified in this attribute will be the starting value for maximum idle time when a new conversation with this service starts. Thereafter, you can update the value through your service's code.

Note: You may change the initial value by editing the Javadoc tag in your source code or through the Properties pane; changes in one will be automatically reflected in the other. Also, note that WebLogic Workshop will not add the Javadoc tag to your source code until you use the Properties pane to change the default value of "0".

For reference information about the @jws:conversation tag, see @jws:conversation−lifetime Tag.

Changing the Maximum Idle Time Value at Run Time

To manage the maximum idle time at run time, you can use the following JwsContext methods:

- void setMaxIdleTime(long seconds) adov−−> Sets a new value to the specified number of seconds.
- void setMaxIdleTime(String duration) adov−−> Sets a new value using a string expressing the idle time duration (such as "2 minutes," "5 days," and so on).
- long getMaxIdleTime() adov−−> Returns the current maximum idle time as a number of seconds.
- void resetIdleTime() adov−−> Sets the maximum idle time back to its initial value. This may be useful when the conversation involves some activity besides receiving messages from the client, but this activity is still considered "non−idle." This could include interacting with other services through controls.
- long getCurrentIdleTime() adov−−> Returns the number of seconds the conversation has been idle.

In the following example involving a database query, maximum idle time is set to allow at least five minutes for the database query to return.

```
public void getEmployeeContactInfo(String employeeID)
{
    context.setMaxIdleTime("5 minutes");
    Employee m_contactInfo =
        employeeDataControl.getContactInformation(employeeID);
}
```

Limiting a Conversation's Duration

A conversation's maximum age is the interval between the time the conversation starts and the time it finishes, before it is finished by WebLogic Server and removed from memory.

You can write code to change the maximum age. This can be useful if, for example, it appears that the combined actions taken by your service to return a response to the client will take longer that originally expected.

Setting the Initial Maximum Age Value at Design Time

You can set the initial maximum age in Design View by clicking the service area, then using the Properties pane to set the conversation−lifetime max−age attribute.

By default, the value of this attribute is "1 day", meaning that the conversation will automatically finish one day after it starts. In general, it is a good practice to estimate a reasonable amount of time for your service to remain active, then set this attribute to that value. Allowing a conversation to remain active for a long period of time can absorb system resources, particularly as new conversation contexts are created for new calls to the service.

When you change the max−age attribute, WebLogic Workshop will annotate your service class with a Javadoc tag resembling the following (this example sets the value to "7 days").

```
/**
 * @jws:conversation−lifetime max−age="7 days"
 */
```

The value specified in this attribute will be the starting value for the maximum age when a new conversation with this service starts. Thereafter, you can update the value through your service's code.

Note: You may change the initial value by editing the Javadoc tag in your source code or through the Properties pane; changes in one will be automatically reflected in the other. Also, note that WebLogic Workshop will not add the Javadoc tag to your source code until you change the default value of max−age in the Properties pane.

For reference information about the @jws:conversation−lifetime tag, see @jws:conversation−lifetime Tag.

Changing the Maximum Age Value at Run Time

To manage a conversation's maximum age at run time, you can use the following JwsContext methods:

- void setMaxAge(String duration) adov−−> Sets a new value using a string expressing the maximum age duration (such as "3 hours," "2 days," and so on).
- void setMaxAge(java.util.Date date) adov−−> Sets a new value to the specified number of seconds.
- long getMaxAge() adov−−> Returns the maximum age of the conversation number in seconds.
- long getCurrentAge() adov−−> Returns the number of seconds since the conversation began.

The following simple example illustrates how you might set the maximum age to a duration intended to allow time for a client to gather more information related to their request and return. Extending the conversation's life allows the service's context (along with any persistent state accrued so far) to remain active until the client returns.

```
public void needMoreInfo(String neededInfoDescription)
```

```
{
    context.setMaxAge("2 days");
    callback.onRequestMoreInfo(neededInfoDescription);
}
```

Finishing a Conversation

The end of a conversation marks the conversation context for removal from memory by WebLogic Server. This means that data that is part of its persistent state will be removed. A conversation may finish in any of the following ways:

- On completion of a service method annotated with "finish" or one that invokes a client callback marked ”finish”.
- The conversation's maximum idle time is exceeded.
- The conversation's maximum age is exceeded.
- Your code calls the JwsContext interface's finishConversation method.

Note that in all but the last case, the conversation may not in fact end immediately. Even though it is marked for destruction, it may still be some time before WebLogic Server actually removes the conversation context.

Calling the finishConversation method, on the other hand, provokes the end of the conversation (and subsequent clean−up by WebLogic Server) upon return of the method or callback in which finishConversation was called. You may find it useful to directly end a conversation in response to client requests, changes in state, or other events. When you want to ensure a timely end to the conversation, call the finishConversation method.

Doing Something When a Conversation Finishes

You can write code that will execute when a conversation finishes, regardless of how the conversation meets its end. To do this, you add code to the JwsContext_onFinish callback handler. You can add this handler to any service that supports conversations.

When a conversation ends your callback handler receives boolean value indicating whether the conversation ended by expiring. A conversation expires if it lasts longer than the value set in the service's conversation−lifetime property max−age attribute. It also expires if your service has remained idle longer than the value set in the conversation−lifetime property max−idle−time attribute.

On the other hand, your handler will receive a value of false if the conversation ended because your code called the JwsContext.finishConversation method. A false value also indicates that the conversation ended through execution of an item (such as a method or callback) whose conversation property phase attribute is set to finish.

To Add Code to Handle the End of a Conversation

1. Locate the variable declared as weblogic.jws.control.JwsContext.
   When you add support for a conversation, WebLogic Workshop automatically adds to your code a variable you can use to manage conversation lifetime. This variable will look something like the following:

/** @jws:context */ weblogic.jws.control.JwsContext context;

2. Using the variable name provided (here, this is context) add a callback handler for the onFinish callback exposed by the JwsContext interface.
   Your callback handler declaration should look something like the following:

```
public void context_onFinish(boolean expired) { }
```

3. Between the braces, add the code you would like to have execute when the conversation finishes. For example, if for debugging purposes you wanted to write a message to the console to indicate how the conversation had ended, you might implement the handler in the following way:

```
public void context_onFinish(boolean expired) {    String finishStatus = "";    if(expired){        finishStatus = "Conversation expired.";    }    else{        finishStatus = "Conversation finished normally.";        }
   System.out.println(finishStatus);     }
```

Related Topics

Life of a Conversation

Defining Conversation Scope

JwsContext Interface

@jws:conversation−lifetime Tag

Supporting Serialization

When you build web services that will support conversations, variables whose data is required throughout the lifetime of the conversation must be declared as types that support serialization. Serialization is a technology through which object data may be written as a stream of bytes to disk.

During a conversation, WebLogic Server uses serialization to write your service's state−related data to disk. This occurs after each completed execution of:

- A method marked with a @jws:operation tag and marked with a @jws:conversation phase attribute.
- A control callback handler.

Note that serialization will not occur for these if the method, callback, or callback handler throws an exception while executing.

You may find that in most cases the serialization prerequisite is easy to meet. For Java primitive types, serialization is supported by default. These types include byte, boolean, char, short, int, long, double, and float. Also, while classes in Java must implement the Serializable interface, many of the Java classes representing common data types already implement do so. These include those that are wrappers around the primitive types adov−−> such as Boolean, Long, Double, Integer, and so on adov−−> as well as String and StringBuffer. (When you are in doubt about whether a particular common class implements Serializable, check reference information for the class before using it.)

When you create your own classes for use in typing member variables, or when you handle classes created by others, you must take care to ensure that these classes implement the Serializable interface. Even so, this is typically easy to do because the interface contains no methods to implement adov−−> implementing the interface merely marks the class as supporting serialization. Your class code must simply import the package containing the interface and its declaration must be marked with the implements keyword, as follows:

```
import java.io.Serializable;
public class MyClass implements Serializable
{...}
```

For more information on supporting serialization, see the serialization information at the Sun Microsystems web site at java.sun.com/j2se/1.3/docs/guide/serialization or your favorite Java book.

Related Topics

Overview: Conversations

How Do I: Add Support for Conversations?

Conversing with Non−Workshop Clients

For services you build with WebLogic Workshop, adding support for conversations is easy. At design time, you simply set the conversation property's phase attribute for your service's methods and callbacks. At run time, WebLogic Server turns these properties into headers within the SOAP messages that web services exchange.

For clients built with tools other than WebLogic Workshop, however, adding conversation support involves a different process. Developers typically add support for conversations with your service by writing their own code to set and retrieve the required SOAP headers. In particular, the headers provide a way for clients to send and receive a conversation ID and URL to which your service should send callbacks. These headers are associated with the publicly exposed methods and callbacks of your service adov−−> known as operations.

Note: If you are developing web services with .NET (or need to interoperate with .NET services), you may be interested in ConversationClient.asmx.cs, a sample available in the interop folder of the Workshop samples project. This file contains code for a web service client written in C#. For more information, see .NET Client Sample.

The three SOAP headers related to conversations are:

- <StartHeader> adov−−> The client sends this header with the call to a method marked to "start" a conversation. The <StartHeader> contains <conversationID> and <callbackLocation> elements.
- <ContinueHeader> adov−−> The client sends this with any subsequent calls to the web service's methods. It contains the conversation ID the client used to start the conversation.
- <CallbackHeader> adov−−> The client receives this header when it receives a callback from the web service. The <CallbackHeader> includes the conversation ID the client used to start the conversation.

As you can see, one piece of information all of the headers carry is the conversation ID. The client sends in the conversation ID with the first request. The same ID is then passed back and forth between the client and your service for each of their exchanges. This way, both know which initial request the message they're currently receiving is related to.

How the Headers Work

A client's first request, which begins a conversation, is always sent to a method marked to "start" the conversation. The SOAP message for this request must include the <StartHeader> header. In the message, this header looks like the following:

```
<SOAP−ENV:Header>
    <StartHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
        <conversationID>someUniqueIdentifier</conversationID>
        <callbackLocation>http://www.mydomain.com/myClient</callbackLocation>
    </StartHeader>
</SOAP−ENV:Header>
```

The <StartHeader> element simply contains the header information. The XML namespace must be used as it is shown here.

The <conversationID> element is technically optional. If it is omitted, WebLogic Server will invent one for use locally. However, the client will have no way of knowing what this conversation ID is if they don't send it. This means that they will be unable to correlate any responses received with the original request. For this reason, the client should only omit the conversation ID if they do not expect any further conversation with the web service.

The <callbackLocation> element, also optional, contains the URL to which callbacks should be sent. The client need send this only if it expects to handle a callback from the web service. When this value is received by the WebLogic Workshop web service, it is stored for later use by WebLogic Server.

After the first request, the client either continues its side of the conversation by calling additional methods of the web service or receives callbacks from the web service. Each subsequent call to the web service must include the <ContinueHeader>. Note that this means even calls to web service methods marked to "finish" the conversation must include the <ContinueHeader>. After the conversation begins, the web service owns its duration adov−−> the web service is responsible for finishing it and for sending a useful response to the client. The <ContinueHeader> might look like the following:

```
<SOAP−ENV:Header>
    <ContinueHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
        <conversationID>theUniqueIdentifierSentWithTheStartHeader</conversationID>
    </ContinueHeader>
</SOAP−ENV:Header>
```

In order to correctly correlate this call with the first, the <conversationID> used here must be the same as the ID sent with the <StartHeader>.

Finally, the client may be designed to handle callbacks from the web service. This means that the client software includes an operation capable of receiving the message sent by the Workshop web service's callback. How this is implemented differs depending on the technology used to build the client. In general, though, the message sent with a callback contains the callback's parameter values.
The callback message also contains the <CallbackHeader>. The <CallbackHeader> includes adov−−> yes, you guessed it adov−−> the conversation ID. It might look like the following:

```
<SOAP−ENV:Header>
    <CallbackHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
        <conversationID>theUniqueIdentifierSentWithTheStartHeader</conversationID>
    </CallbackHeader>
</SOAP−ENV:Header>
```

Notice that there is no "finish" header. The conversation is over when the web service finishes it. This may be through execution of an operation (method or callback) marked with the conversation phase set to "finish". The web service can also call the finish method of the JwsContext interface (something it might do in the event of an exception). Using the WSDL file for the web service, the client knows which operations are designed to finish the conversation.

What to Look for in a Conversational Web Service's WSDL File

By looking at the WSDL file generated from a WebLogic Workshop service, someone developing software that may be a client of the service can discover which of the service's operations start, continue or finish a conversation. They can also see which SOAP header each operation requires.

The following excerpt is from a WSDL generated from the Conversation.jws web service in the WebLogic Workshop samples project. In this example:

- The name of each operation (method and callback) is highlighted in bold.
- The phase attribute describing the operation's role in a conversation is highlighted in blue.
- The part attribute describing which header the service will receive (as an "input" header) and which it will send (as an "output" header) is highlighted in red.

By looking at this excerpt, you can see that:

- The startRequest operation starts the conversation and requires a <StartHeader>.
- The getRequestStatus operation continues the conversation and requires a <ContinueHeader>.
- The terminateRequest operation finishes the conversation and requires a <ContinueHeader>.
- The onResultReady operation finishes the conversation and sends a <CallbackHeader>.

```
<binding name="ConversationSoap" type="s0:ConversationSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
  <operation name="startRequest">
    <soap:operation soapAction="http://www.openuri.org/startRequest" style="document" />
    <cw:transition phase="start" />
    <input>
      <soap:body use="literal" />
      <soap:header wsdl:required="true" message="s0:StartHeader_literal" part="StartHeader" u
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
  <operation name="getRequestStatus">
    <soap:operation soapAction="http://www.openuri.org/getRequestStatus" style="document" />
    <cw:transition phase="continue" />
    <input>
      <soap:body use="literal" />
      <soap:header wsdl:required="true" message="s0:ContinueHeader_literal" part="ContinueHea
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
  <operation name="terminateRequest">
    <soap:operation soapAction="http://www.openuri.org/terminateRequest" style="document" />
    <cw:transition phase="finish" />
    <input>
      <soap:body use="literal" />
      <soap:header wsdl:required="true" message="s0:ContinueHeader_literal" part="ContinueHea
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
  <operation name="onResultReady">
    <soap:operation soapAction="http://www.openuri.org/onResultReady" style="document" />
    <cw:transition phase="finish" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
```

```
        <soap:body use="literal" />
        <soap:header wsdl:required="true" message="s0:CallbackHeader_literal" part="CallbackHea
      </output>
    </operation>
  </binding>
```

Related Topics

Life of a Conversation

Overview: Conversations

How Do I: Tell Developers of Non−WebLogic Workshop Clients How to Participate in Conversations?

Handling and Shaping XML Messages with XML Maps

Web services you build communicate with their clients and other web services through text messages formatted in the syntax of Extensible Markup Language (XML). There may be occasions when the messages your web service receives and sends must conform to a specific format. When this is the case, you can ensure a functional relationship between the parts of a message and the Java types of your service code by using XML maps.

In addition, using XML maps supports loose coupling between your web service's code and other components such as clients and resources used by your service; in other words, you need not change your Java code to meet the needs of components with which your service communicates.

An XML map acts as a bridge between your Java code and an XML message's structure adov−−> or shape. Using XML maps, you can anticipate the shape of incoming messages or control the shape of outgoing messages.

Note: This section assumes that you are familiar with the basics of XML. For basic information about XML, see Introduction to XML.

Topics Included in This Section

Why Use XML Maps?

Introduces XML maps and ECMAScript for mapping.

Getting Started with XML Maps

Offers starting places for working with XML maps.

Matching XML Shapes

Provides topics on specific tasks you can accomplish with XML maps.

Getting Started with Script for Mapping

Offers starting places for working the ECMAScript functions used with XML maps.

Handling XML with ECMAScript Extensions

Provides topics on specific tasks you can accomplish with operators that are part of the ECMAScript extensions for XML.

Functions for Manipulating XML

Offers a reference on the functions available with extended ECMAScript.

A Few Things to Remember About XML Maps and Script

Lists a few guidelines that may be useful when using XML maps and script.

Samples

The following illustrate XML maps:

- SimpleMap.jws
- OutputMap.jws
- OutputScriptMap.jws
- InputMapMultiple.jws
- ConsolidateAddress.jws

For general information about samples, see Samples.

Related Topics

@jws:parameter−xml Tag

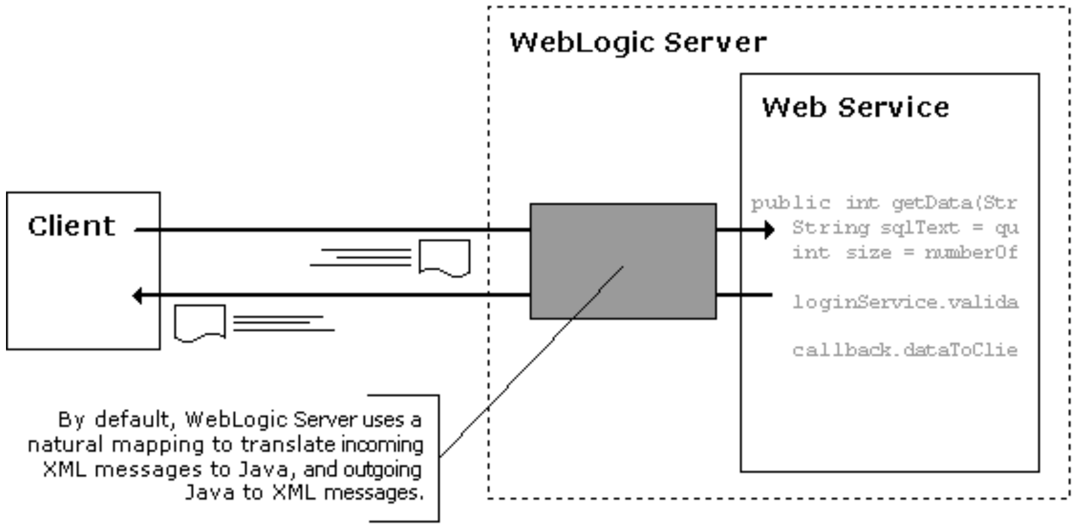
@jws:return−xml Tag

XML Map Tag Reference

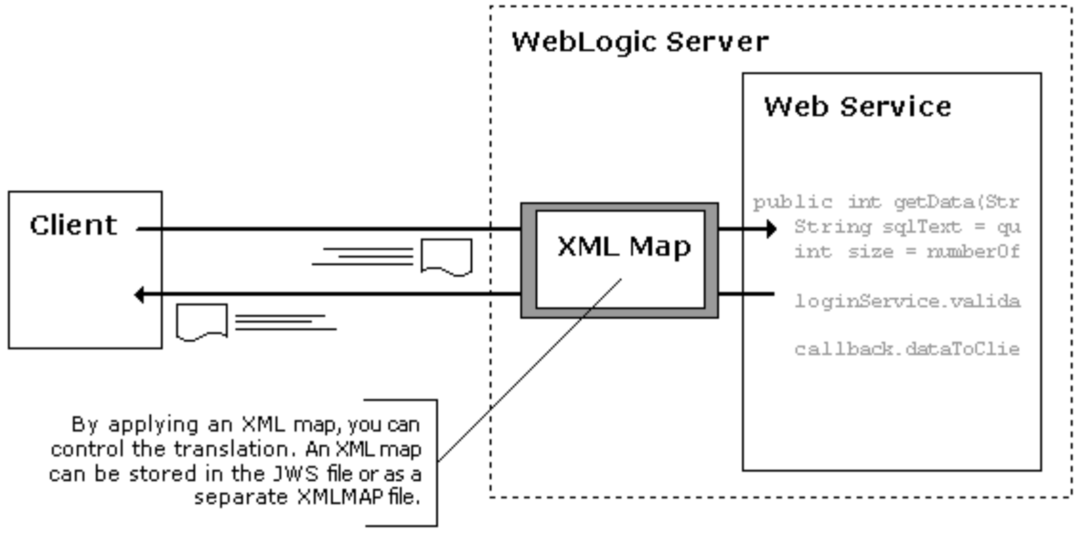
Introduction to XML

Why Use XML Maps?

Web services you build with WebLogic Workshop communicate by sending and receiving XML messages. By default, WebLogic Server translates these messages to and from the types in your Java declaration according to a "natural" map adov−−> a format in which the parts of your Java declaration match the contents of the message.



In some cases, however, you may want to enable your service to receive or send messages that don't match adov−−> perhaps because you want your service to work well with a client or resource whose message format can not be changed. In these cases, mapping through XML maps and script provides a way to handle different message shapes without having to change your Java code.



There are two general cases in which you might want to provide your own maps:

- You want to control the shape of the outgoing message, perhaps because the message's recipient requires a particular format.
- You need to allow for a particular incoming XML message shape, and want to avoid changing your service's code.

For example, you might build a service that has appeal to a potential client whose XML message format is specific to their industry, but differs from what your service is designed to handle. You may want to make it easier for that client to use your service by handling their format rather than requiring them to conform to yours. By overriding natural mapping with your own XML map, you effectively create a translation layer that handles the format of their request messages while allowing your implementation code to remain unchanged.

More specifically, through XML maps you can:

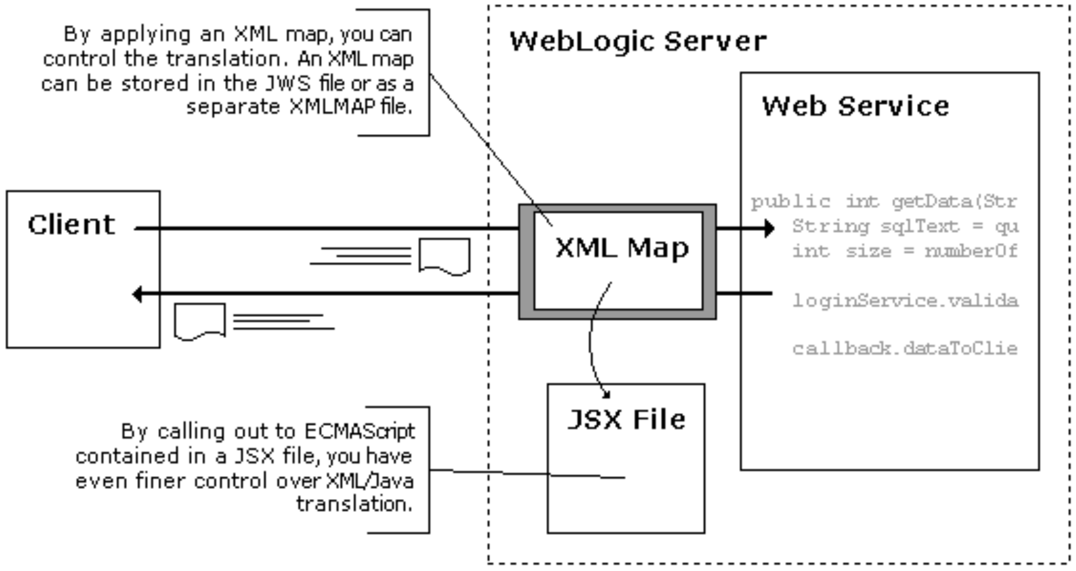
- Map specific XML element content and XML attribute values to Java method parameters and return values.

- Handle more dramatic differences between natural mapping and required formats by diverting translation processing to a script or map that is external to your service method code.

Note: To view the natural map for a Java declaration, open the web service in Design View, then double−click the method, callback, or callback handler whose map you want to see. In the Edit Maps and Interface dialog, with the Default option selected, you can view the natural map for either parameters or return value. If the XML messages sent and received by this member of your service will not match this format, then you will probably need to create a custom map with an XML map or script.

What's the Difference Between Using an XML Map and Using Script?

XML maps and script used for mapping accomplish the same general goal, but maps do it more simply while script does it more powerfully. One rule of thumb might be to first consider using a map, creating one with the Edit Maps and Interface dialog, then incorporate script if it appears that the shape of the XML message differs too greatly from types in your Java declaration. Note that to use script, you must also be familiar with the ECMAScript language.


By applying an XML map, you can control the translation. An XML map can be stored in the JWS file or as a separate XMLMAP file.
WebLogic Server
Web Service
Client
XML Map
public int getData(Str
String sqlText = qu
int size = numberOf
loginService.valida
callback.dataToClie
JSX File
By calling out to ECMAScript contained in a JSX file, you have even finer control over XML/Java translation.

Because maps resemble the message to which they are mapping, working with maps can feel like a natural way to express how parts of your Java declaration correspond to the message format. Creating a map is a little like aligning two sections of a puzzle that have been put together independently.

In contrast, creating a script for mapping is more like cutting out a brand new puzzle based on a picture of what it should look like, then assembling the pieces into a completed whole. With the extension to the ECMAScript language provided with WebLogic Workshop (including the ability to handle XML as a native data type), you can construct outgoing XML messages from scratch or access incoming messages as you would other data structures. ECMAScript is capable of handling pretty much all of your mapping needs. If you are familiar with ECMAScript, there is no reason why you couldn't use script in every case.

How Do I Get Started Creating an XML Map or Script?

One good way to start creating the map is to begin with the XML schema adov−−> or an example adov−−> of the message format your map will accommodate. Looking at the schema, identify the parts of the message that correspond to the parts of your Java declaration. (You can learn more about XML schema at Introduction to XML.) If you will be mapping from an incoming message, ask yourself which elements and attributes will contain data that will be needed by your Java code; if you are mapping to an outgoing message, decide how the data in your Java declaration that will be sent to recipients should be parsed into the structure of the XML message.

Once you have an understanding of how the message shape and your Java declaration correspond to one another, you are ready to begin creating an XML map. In practice, for simple needs, you may find it easiest to always begin an XML map with the Edit Maps and Interface dialog, as described in How Do I: Add or Edit an XML Map with the Edit Maps and Interface Dialog? However, there are a couple of questions you should answer before setting out to make a map, especially as your web service's needs become more complex.

- Decide whether the map should be implemented inline with Java source code or as a standalone map file.

Storing your XML map inline with service class source code is convenient from an editing perspective; storing the map in a separate file makes it reusable with other methods, even other web services. For more information on implementing inline XML maps, see How Do I: Add or Edit an XML Map with the Edit Maps and Interface Dialog? For more information on creating standalone maps, see How Do I: Begin a Reusable XML Map?

- Decide whether it will be necessary to manipulate the shape of the XML message with script.

The differences between Java types in a given implementation and the shape of an XML message may be so dramatic that it is not possible to create a map without manipulating the shape of the XML with script. When your needs require that you manipulate the shape of XML messages, WebLogic Workshop provides extensions to the ECMAScript language through which you can manipulate XML documents as you would other data structures. For more information, see Using Script Functions From XML Maps.

Related Topics

How Do I: Add or Edit an XML Map with the Edit Maps and Interface Dialog?

Using Script Functions From XML Maps

Getting Started with XML Maps

This section provides introductory information about XML maps. If you're already familiar with XML map basics, the topics in the Matching XML Shapes section might be useful.

Topics Included in This Section

How Do XML Maps Work?

Gives an overview of what XML maps look like and how you can use them to translate to and from XML.

How Do I: Add or Edit an XML Map with the Edit Maps and Interface Dialog?

Provides a step−by−step introduction to using the dialog.

Creating Reusable XML Maps

Introduces XMLMAP files, which contain standalone XML maps.

Type Support in XML Maps

Lists the Java types supported by XML maps and gives examples of how they are translated into schema types.

Related Topics

Getting Started with Script for Mapping

Handling and Shaping XML Messages with XML Maps

How Do XML Maps Work?

XML maps and script act as a translation layer between your web service (JWS file) and the network that carries XML messages between your service and components with which it communicates. You create maps or script to tell WebLogic Server how types in your Java declaration correspond to the contents of the XML message. At run time WebLogic Server executes your map or script when it passes XML messages between your service and the network.

You can include an XML map in a JWS or CTRL file, immediately preceding the Java declaration with which it is intended to be used. You can also create an XML map or script in a separate file and reference it from an annotation preceding the declaration.

For a more detailed overview of what happens when a map or script executes, see Using Script Functions from XML Maps.

A given method or callback may have two XML maps: one to map parameter values and one to map return values adov−−> these are called the parameter−xml map and the return−xml map, respectively. For example, consider a method declaration such as the following:

```
public String searchRequest(String productName, String serialNumber, int quantity)
```

The parameters in this declaration are serialNumber and quantity; the return type is a String. The default maps will look something like the following.

```
/**
 * @jws:operation
 * @jws:parameter−xml xml−map::
 *     <searchRequest>
 *         <productName serialNumber="{serialNumber}">{productName}</productName>
 *         <quantity>{quantity}</quantity>
 *     </searchRequest>
 *
 * ::
 * @jws:return−xml xml−map::
 *     <searchRequestResponse>
 *         <return>{return}</return>
 *     </searchRequestResponse>
 *
 * ::
 */
public String searchRequest (String productName, String serialNumber, int quantity)
```

Notice in this example that the XML element and attribute names adov−−> <searchRequest>, <productName>, serialNumber, <return>, and so on adov−−> reflect what is in the method signature. Also, the method's parameters are enclosed in {} as substitutions to capture what will arrive in the message as actual values.

Note: While this map is stored in an annotation immediately preceding the declaration to which the map applies, you can also store the map in a separate XMLMAP file, then refer to it from the annotation in your JWS file. For more information about storing maps in a separate file, see Creating Reusable XML Maps.

This default parameter−xml map assumes that an XML message carrying an incoming method call will look like the following:

```
<searchRequest>
    <productName serialNumber="12345">Widget</productName>
    <quantity>3</quantity>
</searchRequest>
```

But imagine a case in which a client wants to use your service, but they are already committed to a different message format based on an agreement within their clients' industry. For example, a message from them might appear as follows:

```
<queryData>
    <partName partID="12345">Widget</partName>
    <partQuantity>3</partQuantity>
</queryData>
```

This is just the sort of situation that XML maps are designed to address. For the searchRequest method, you could design a map that resembled the XML format used by expected request messages. That map would substitute element content, attribute values, and the like with bracketed placeholders directing that content and those values to parameters of the method. An example of a parameter−xml map for the searchRequest method might appear as follows:

```
/**
 * @jws:operation
 * @jws:parameter-xml xml-map::
 *    <searchRequest>
 *        <queryData>
 *            <partName partID="{serialNumber}">{productName}</partName>
 *            <partQuantity>{quantity}</partQuantity>
 *        </queryData>
 *    </searchRequest>
 * ::
 */
public String searchRequest (String productName, String serialNumber, int quantity)
{ ... }
```

Because XML maps are designed according to the expected shape of an XML message, the process of creating an XML map always begins with an example XML document of the sort to be matched. For example, if the map is being created in keeping with a client's constraints for the shape of XML message, obtaining an example message from a the client should be the first step.

Note: The root tags (such as the <searchRequest> tag in the preceding example) must be unique across maps within a given JWS file. Because of this, it is a good practice to place your map within tags whose names match your method name; after all, methods must also be unique.

Applying parameter−xml and return−xml Maps

There are two kinds of maps you can create: parameter−xml and return−xml. As you might expect, you use a parameter−xml map when you want to map XML values to or from the parameters of a method, callback or callback handler; you use a return−xml map for the return values.

The following table describes where each kind of map is used.

Interface Member	Direction of Message Travel	Kind of Map to Use
Method of your service (JWS file)	Incoming (from a client)	parameter−xml
	Outgoing (toward a client)	return−xml
Callback of your service (JWS file)	Incoming (from a client)	return−xml
	Outgoing (toward a client)	parameter−xml
Method of a control (CTRL file)	Incoming (from the resource)	return−xml
	Outgoing (toward the resource)	parameter−xml
Callback handler of your service (JWS file)	Incoming (from the resource)	parameter−xml
	Outgoing (toward the resource)	return−xml

For more information on handling incoming messages or shaping outgoing messages, see How Do I: Handle Incoming XML Messages of a Particular Shape? and How Do I: Ensure That Outgoing Messages Conform to a Particular Shape?

Syntax for parameter−xml and return−xml Maps

When mapping parameters with a parameter−xml map, you substitute parameter names for XML values. When mapping return values, you substitute the word "return" for XML values. The following example illustrates each; bold text indicates where substitutions occur.

For reference information, see @jws:parameter−xml Tag and @jws:return−xml Tag.

```
/**
 * @jws:operation
 * @jws:parameter-xml xml-map::
 *    <getInventory>
 *     <queryData>
 *         <part>{serialNumber}</part>
 *     </queryData>
 *    </getInventory>
 * ::
 * @jws:return-xml xml-map::
 *     <getInventoryResponse>
 *     <queryData>
 *         <inventoryCount>{return}</inventoryCount>
 *     </queryData>
 *     </getInventoryReponse>
 *      * ::
 */
public int getInventory(String serialNumber)
{...}
```

Note: The root tags (such as the <queryData> tag in the preceding example) must be unique across maps within a given JWS file. Because of this, it is a good practice to place your map within tags whose names match your method name; after all, methods must also be unique.

Maps in Source Code and in Map Files

You can put XML maps you create in one of two places. First, they can be placed inline with your Java source code (for ease of editing). They can also be placed in a separate map file (for reusability).

When you put a map inline in Java source code, you use the Edit Maps and Interface dialog. The dialog puts the map you create immediately preceding the declaration for the method, callback, or callback handler to which the map applies. When put in this location, the map may only be used with the corresponding Java declaration. However, you may also find that putting the map preceding the declaration makes it easier to find and edit. For example, an XML map put with source code is readily available for editing with the Edit Maps and Interface dialog simply by double−clicking the corresponding map icon in Design View.

For more information on using the Edit Maps and Interface dialog, see How Do I: Add or Edit an XML Map with the Edit Maps and Interface Dialog?

When creating a separate map file, you create a text file with an .xmlmap extension and put XML maps into it. Putting maps into a separate map file enables you to use that file as a common resource; it can be used with, for example, all of the methods, callbacks and callback handlers in a web service.

Note that the code providing map functionality is the same for an XML map in a map file as it is for a map in source code. In other words, you can create a map and put it with your Java source code, then later move it to a map file without making changes to the way the map works. Map code in a map file differs only in that it must be enclosed in an <xm:xml−map> tag that enables you to invoke it from source code.

For more specific information on map files, see Creating Reusable XML Maps.

Related Topics

Matching XML Shapes

How Do I: Add or Edit an XML Map with the Edit Maps and Interface Dialog?

You can use the Edit Maps and Interface dialog (shown in this topic) to override natural mapping for XML messages sent between your service and clients, other services, or other resources. You display the dialog by locating the item in Design View to which the map will apply and double−clicking the map icon (shown here) corresponding to it.

In the icon, an arrow pointing inward toward the area of your service represents a map applied to an incoming message; an arrow pointing outward represents a map applied to the outgoing data.

Note: When you use the Edit Maps and Interface dialog, you create an XML map that is stored with the source code of your JWS or CTRL file. XML maps can also be put in separate map files and used from multiple places in source code. For information differentiating the two locations, see How Do XML Maps Work?

The Edit Maps and Interface dialog provides tabs to display the parameter−xml map and return−xml map. For parameter−xml maps, the default XML displayed in the dialog uses element names that match your parameter names; for return−xml maps, the default element name is simply <return>.

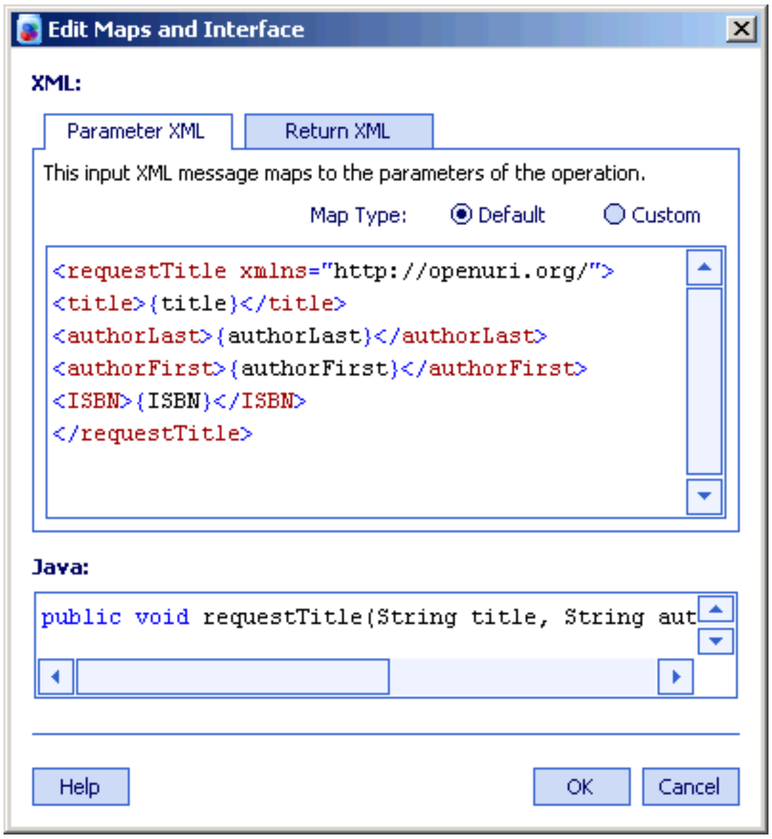
Before beginning an XML map you should obtain a sample copy of an XML message that your method will be expected to handle or generate. The element and attribute names within your XML map must match those in the XML message to be mapped; this is an essential aspect of XML maps. Examine the structure of the example XML to identify the elements and attributes that will contain values that should be mapped to your method's parameters. Identify a section of the XML that contains the elements corresponding to your implementation, then use this section as a basis for the XML map.

To Begin a Custom XML Map

1. Double−click the map icon corresponding to the method, callback, or callback handler to which you

are applying the map.

The Edit Maps and Interface dialog appears, as shown here:



This input XML message maps to the parameters of the operation.

Map Type: [✓ Default] Custom

```
<requestTitle xmlns="http://openuri.org/">
<title>{title}</title>
<authorLast>{authorLast}</authorLast>
<authorFirst>{authorFirst}</authorFirst>
<ISBN>{ISBN}</ISBN>
</requestTitle>
```

```
public void requestTitle(String title, String aut
```

2. Click one of the following tabs, as appropriate:

- Click the Parameter XML tab to map XML values to the parameters of your Java declaration
- Click the Return XML tab to map XML values to the return value of your Java declaration.

For more information on parameter−xml and return−xml maps, see How Do XML Maps Work?

3. In the XML pane, paste the section of example XML you identified.

4. Edit the XML you pasted, replacing actual content with substitution directives. For more information on substitution directives see How Do XML Maps Work?

5. Click OK to store your edited XML map in the source code of your service.
6. To edit the XML map, double−click the corresponding map icon in Design View or edit the map directly in Source View.

Related Topics

Edit Maps and Interface Dialog (General)

Matching XML Shapes

How Do XML Maps Work?

Creating Reusable XML Maps

You can reuse XML maps by putting them into a map file, a file with an .xmlmap extension. While maps in map files function the same as those put into source code, map files have the added benefit of being available to multiple methods, callbacks and callback handlers, instead of just one. This means that you can reuse the maps in a map file from anywhere in your project, or from another project. In addition, by adding multiple maps to a map file, you can group related maps in separate map files.

Note: This topic describes the specific characteristics of XML map files. For more information about mapping tasks in general (which you can perform in both inline maps and map files), see How Do XML Maps Work?

Preparing to use a map file is a two−step process: creating the map file and invoking it from source code.

Creating a Map File

As described in How Do I: Begin a Reusable XML Map?, you create a map file by adding a text file to you project and giving it an .xmlmap extension. The following example describes the contents of this file.

An XML map file must begin with the following <xm:map−file> tag. This indicates to WebLogic Workshop that maps are contained in the file; the tag also declares the "xm" prefix that delimits a namespace for map tags such as <xm:value>.

```
<xm:map-file xmlns:xm="http://www.bea.com/2002/04/xmlmap/">
```

You may optionally follow the <xm:map−file> tag with the <xm:java−import> tag. Use the <xm:java−import> tag if the substitution directives in your maps will require Java classes not available in the scope of the method or callback that invokes the map. The following example imports the Date class and a user−defined class:

```
<xm:java-import class="java.util.Date"/>
<xm:java-import class="com.MyCompany.MyType"/>
```

The <xm:java−import> is similar to the Java import directive. However, note that you may use only fully−qualified class names in the class attribute adov−−> you may not use a * to import all classes in a package.

Each XML map in a map file is contained within <xm:xml−map> tags. The signature attribute of these tags defines the map's signature. This is not to be confused with the signature of a method or callback, although the two may be similar. An XML map's signature specifies the map's name. It is used when referring to the map and any parameters used by the map. The parameters correspond to substitution directives within the map itself. For example, to use the following map, you would invoke it as placeOrder, passing as data the parameter values of the method or callback from which you are invoking it. For more information on invoking a map in a map file, see the following section.

```
    <xm:xml-map signature="placeOrder(String serialNumber, int quantity)">
        <serialNumber>{serialNumber}</serialNumber>


        <quantity>{quantity}</quantity>
    </xm:xml-map>
```

You may put many of these XML maps into a single map file. Finally, an XML map file ends with the following </xm:map−file> tag:

```
</xm:map-file>
```

Invoking a Map in a Map File

You invoke a map in a map file through the <xm:use> tag or its syntax alternative (described at the end of this topic). The <xm:use> tag has one attribute, call, which is used to indicate the location of the map to invoke, as well as the values to pass as parameters.

In the following example, the partID and partQuantity parameters of the requestParts method are passed to a map called "placeOrder" (such as the map in the preceding example) that is in a map file called "CustomerRequests."

```
/*


 * @jws:operation


 * @jws:parameter-xml xml-map::
 * <requestParts>
 *     {CustomerRequests.placeOrder(String partID, int partQuantity)}
 * </requestParts>
 * ::


public void requestParts(String partID, int partQuantity)
{...}
```

Note that the path to the map must be full enough to locate it. For example, if the map file were contained in parallel folder of a project called "CustomerMaps," and the project itself were called "OrderServices," the path used above might instead be:

```
OrderServices.CustomerMaps.CustomerRequests.placeOrder(String partID, int partQuantity)
```

Related Topics

<xm:map−file> Tag

<xm:xml−map> Tag

<xm:java−import> Tag

<xm:use> Tag>

Type Support in XML Maps

This topic describes the default schema output for Java types you translate using an XML map. When you use XML maps to create and outgoing XML message, WebLogic Server maps Java types to XML schema types as described in this topic. This topic lists each Java type supported by XML maps and shows the corresponding schema definition for each.

Note: The default behavior is different for XML generated through ECMAScript in a JSX file, even though it is funneled through an XML map. For XML you create with script, the default is simple untyped XML. You must indicate schema (where needed) in script by creating XML variables, then assigning to them XML values that include literal schema attributes.

Here are the Java types supported by XML maps:

Information Given in This List

Each item in this list includes the following:

Java code that might be used to declared a variable of the type:

String a = "a string";

An example of the variable as it might appear in an XML map:

<mytag>{a}</mytag>

An example of the plain XML the map would produce (or handle, for incoming messages):

<mytag>a string</mytag>

An example of how the variable would be described in a WSDL file. You're unlikely to use this information directly while using WebLogic Workshop because WebLogic Server handles the translation. But in other development models, where messages must be handled directly, this information can be valuable.

<s:element name="mytag" type="s:string"/>

String

Java Variable

String a = "a string";

XML Map Use

<mytag>{a}</mytag>

XML Result

<mytag>a string</mytag>

Schema for Result

<s:element name="mytag" type="s:string"/>

Boolean

Java Variable

boolean f = true;

XML Map Use

<mytag>{f}<mytag>

XML Result

<mytag>true</mytag>

Schema for Result

<s:element name="mytag" type="s:boolean"/>

Byte

Java Variable

byte b = 250;

XML Map Use

<mytag>{b}<mytag>

XML Result

<mytag>−6</mytag>

Schema for Result

<s:element name="mytag" type="s:byte"/>

Short Integer

Java Variable

short s = 537;

XML Map Use

<mytag>{s}</mytag>

XML Result

<mytag>537</mytag>

Schema for Result

<s:element name="mytag" type="s:shortint"/>

Integer

Java Variable

int i = 12345;

XML Map Use

<mytag>{i}</mytag>

XML Result

<mytag>12345</mytag>

Schema for Result

<s:element name="mytag" type="s:int"/>

Long Integer

Java Variable

long l = 123456789;

XML Map Use

<mytag>{l}</mytag>

XML Result

<mytag>123456789</mytag>

Schema for Result

<s:element name="mytag" type="s:longint"/>

Single−Precision Floating Point Number

Java Variable

float f = 1.23f;

XML Map Use

<mytag>{f}</mytag>

XML Result

<mytag>1.23</mytag>

Schema for Result

<s:element name="mytag" type="s:floatingpoint"/>

Double−Precision Floating Point Number

Java Variable

float d = 1.2345;

XML Map Use

<mytag>{f}</mytag>

XML Result

<mytag>1.2345</mytag>

Schema for Result

<s:element name="mytag" type="s:doublefloat"/>

Date

Java Variable

java.util.Date date= new java.util.Date();

XML Map Use

<mytag>{date}</mytag>

XML Result

```
<mytag>2002−04−14T13:57:12.046Z</mytag>
```

Schema for Result

```
<s:element name="mytag" type="s:dateTime"/>
```

Typed Arrays of Any of the Preceding Types

Java Variable

```
String[] sa = new String[] {"first", "second", "third"};
```

XML Map Use

```
<mytag>{sa}</mytag>
```

XML Result

```
<mytag>
  <String>first</String>
  <String>second</String>
  <String>third</String>
</mytag>
```

Schema for Result

```
<s:element name="mytag" type="s0:ArrayOfString"/>
<s:complexType name="ArrayOfString">
 <s:sequence>
  <s:element minOccurs="0" maxOccurs="unbounded" name="String" nillable="true" type="s:string" />
 </s:sequence>
</s:complexType>
```

List

The following examples describe how an entire list is translated to XML when mapped as a single unit. You can also map individual members of a list as described in Handling Repeating XML Values with <xm:multiple> and Declaring Variables with <xm:bind>.

XML maps support subclasses of the Collection class, including the following:

- Collection
- AbstractCollection
- AbstractList
- AbstractSequentialList
- LinkedList
- ArrayList
- Vector
- AbstractSet
- HashSet
- TreeSet
- List

However, the following are not supported:

- AbstractMap
- HashMap
- TreeMap
- WeakHashMap
- Map
- HashTable
- Iterator
- ListIterator
- Enumeration

Java Variable

```
ArrayList list = new ArrayList();
list.add("first");
list.add("second");
list.add(new Integer(71));
```

XML Map Use

```
<mytag>{list}</mytag>
```

XML Result

```
<mytag>

  <anyType xmlns:xsd="http://www.w3.org/2001/XMLSchema" xsi:type="xsd:string">first</anyType>

  <anyType xmlns:xsd="http://www.w3.org/2001/XMLSchema" xsi:type="xsd:string">second</anyType>

  <anyType xmlns:xsd="http://www.w3.org/2001/XMLSchema" xsi:type="xsd:int">71</anyType>

</mytag>
```

Schema for Result

```
<s:element name="mytag" type="s0:List" />
```

<s:complexType name="List">

 <s:sequence>

  <s:element minOccurs="0" maxOccurs="unbounded" name="anyType" nillable="true" type="s:anyType" />

 </s:sequence>

</s:complexType>

Structure

The following examples describe how an entire structure is translated to XML when mapped as a single unit. You can also map individual members of a structure as described in Binding to Java Data Members.

Java Variable

```
static public class Structure
{
    public int intField = 43;
    public String stringField = "member2";
}
Structure s = new Structure();
```

XML Map Use

<mytag>{s}</mytag>

XML Result

<mytag>

  <intField>43</intField>

  <stringField>member2</stringField>

</mytag>

Schema for Result

<s:element name="mytag" type="s0:Structure" />

<s:complexType name="Structure">

 <s:sequence>

  <s:element minOccurs="1" maxOccurs="1" name="intField" type="s:int" />

  <s:element minOccurs="1" maxOccurs="1" name="stringField" nillable="true" type="s:string" />

 </s:sequence>

</s:complexType>

JavaBeans with Public get and set Method

The following examples describe how get/set pairs are translated to XML when mapped as a single unit. You can also map individual members as described in Binding to Java Data Members.

Java Variable

```
static public class Bean
{
    public String getName() { return theName; }
    public void setName(String s) {theName = s;}
    private String theName = "A name";

    public int getNumber() { return theNumber; }
    public void setNumber(int i) {theNumber = i;}
    private int theNumber = 8;
}
Bean b = new Bean();
```

XML Map Use

<mytag>{b}</mytag>

XML Result

```
<mytag>

  <name>A name</name>

  <number>8</number>

</mytag>
```

Schema for Result

```
<s:element name="mytag" type="s0:Bean" />

<s:complexType name="Bean">

 <s:sequence>

  <s:element minOccurs="1" maxOccurs="1" name="name" nillable="true" type="s:string" />

  <s:element minOccurs="1" maxOccurs="1" name="number" type="s:int" />

 </s:sequence>

</s:complexType>
```

XML Element

Java Variable

```
Document myDocument = new weblogic.apache.xerces.dom.DocumentImpl();

Text text = myDocument.createTextNode("This is a root element");

Element root = myDocument.createElement("myRootElement");

root.appendChild(text);

myDocument.appendChild(root);
```

XML Map Use

```
<mytag>{root}</mytag>
```

XML Result

```
<mytag>

  <myRootElement>This is a root element</myRootElement>

<mytag>
```

Schema for Result

```
<s:element name="mytag">

    <s:complexType>

        <s:sequence>

            <s:element minOccurs="0" maxOccurs="1" name="mytag">

                <s:complexType mixed="true">

                    <s:sequence>

                        <s:any />

                    </s:sequence>

                </s:complexType>

            </s:element>

        </s:sequence>

    </s:complexType>

</s:element>
```

XML Document Fragment

Java Variable

```
Document myDocument = new weblogic.apache.xerces.dom.DocumentImpl();

DocumentFragment frag = myDocument.createDocumentFragment();

Text text = myDocument.createTextNode("Some fragment text");

Element sibling = myDocument.createElement("testElement");

sibling.appendChild(text);

frag.appendChild(sibling);

Text text2 = myDocument.createTextNode("More fragment text");

Element sibling2 = myDocument.createElement("testElement2");

sibling2.appendChild(text2);

frag.appendChild(sibling2);
```

XML Map Use

```
<mytag>{frag}</mytag>
```

XML Result

```
<mytag>

  <testElement>Some fragment text</testElement>

  <testElement2>More fragment text</testElement2>

</mytag>
```

Schema for Result

```
<s:element name="mytag">

  <s:complexType>

    <s:sequence>

      <s:element minOccurs="0" maxOccurs="1" name="mytag">

        <s:complexType mixed="true">

          <s:sequence>
```

```
                    <s:any />
                </s:sequence>
            </s:complexType>
        </s:element>
    </s:sequence>
  </s:complexType>
</s:element>
```

Related Topics

Type Support in ECMAScript

Matching XML Shapes

This section provides topics on specific tasks you can accomplish with XML maps. If you aren't already familiar with how XML maps work, see Why Use XML Maps? or How Do XML Maps Work?

Topics Included in This Section

Making Simple Substitutions Using Curly Braces

Introduces the simplest way to map Java types to XML values.

Handling Repeating XML Values with <xm:multiple>

Describes how you can handle or create patterns of repeating XML elements.

Binding to Java Data Structure Members

Describes a simple way to map Java data members to XML.

Declaring Variables with <xm:bind>

Introduces the <xm:bind> attribute, through which you can declare and bind to variables not declared elsewhere.

Namespaces in XML Maps

Shows how to declare a namespace in an XML map.

Simplifying Maps for Optional Elements

Shows how you can keep XML maps simple and efficient by mapping only the relevant portion of an XML message.

Related Topics

Why Use XML Maps?

How Do XML Maps Work?

Making Simple Substitutions Using Curly Braces

The simplest kind of mapping you can do is to map one Java value with one XML value. For these cases, the best way to map the two is by using {} (curly braces) to indicate where the Java values fit into the XML message.

For example, consider the following snippet of an XML message carrying data used to submit a request for information about a manufacturer's inventory:

```
<getInventory>
    <queryData>
        <part id="34860984">Flangyhoffklinger</part>
    </queryData>
</getInventory>
```

The following example might be designed to respond to the preceding message snippet. As you can see, by enclosing the method's parameter names in {}, you tell WebLogic Server what belongs where. The serialNumber parameter value is mapped to the id attribute, and the partName parameter value is mapped to the <part> element.

```
/**
 * @jws:operation
 * @jws:parameter-xml xml-map::
 *     <getInventory>
 *     <queryData>
 *         <part id="{serialNumber}">{partName}</part>
 *     </queryData>
 *    </getInventory>
 * ::
 */
public int getInventory(String serialNumber, String partName)
{...}
```

Note: The <xm:value> and <xm:attribute> tags are alternatives to using curly braces.

You also use the curly braces to include a reference to ECMAScript or a separate map file. For more information, see Using Script Functions From XML Maps and Creating Reusable Maps.

Related Topics

Matching XML Shapes

<xm:value> Tag

<xm:attribute> Tag

<xm:use> Tag

Handling Repeating XML Values with <xm:multiple>

An XML message may contain elements that occur multiple times within the message's structure. In the following example, the <part> element and its children repeat three times to make up a single order.

```
<order>
    <part>
        <partID>19573</partID>
        <partQuantity>4</partQuantity>
    </part>
    <part>
        <partID>28912</partID>
        <partQuantity>1</partQuantity>
    </part>
    <part>
        <partID>39485</partID>
        <partQuantity>57</partQuantity>
    </part>
</order>
```

To handle all of the repeating elements adov−−> perhaps choosing from among them iteratively adov−−> you can capture the values for repeating elements in a Java data structure such as an array. You can use the <xm:multiple> attribute to capture the values for the repeating elements in a Java data structure, then iterate through the structure in your code.

The following example is designed to operate on an incoming XML message like the preceding example. In this example, the <xm:multiple> attribute specifies that the contents of the <partID> and <numberOfItems> elements should be added as members of the serialNumber and quantity arrays.

```
/**
 * @jws:operation
 * @jws:parameter−xml xml−map::
 * <placeOrder>
 * <order>
 *      <part xm:multiple="String serial in serialNumber, int quant in quantity">
 *         <partID>{serialNumber}</partID>
 *         <numberOfItems>{quantity}</numberOfItems>
 *      </part>
 * </order>
 * </placeOrder>
 * ::
 */
public void placeOrder(String[] serialNumber, int[] quantity)
{
      for (int i = 0; i < serialNumber.length; i++)
      {
        System.out.println("Ordered " + quantity[i] + " of part " + serialNumber[i]);
      }
}
```

Note that this also works in reverse. For example, if this were a callback, and the parameters were being mapped to an outgoing (rather than incoming) message, the map above would result in XML like the example preceding it.

Mapping Repeating XML Values to a Java Return Values

This technique is also useful when the repeating values correspond to a Java return value, rather than a parameter. Note that in the following example, the word return is used to indicate that the return value is being mapped.

```
/**
 * @jws:operation
 * @jws:return−xml xml−map::
```

```
 * <returnPartNames>
 *     <partName xm:multiple="String i in return">{i}</partName>
 * </returnPartNames>
 * ::
 */
public String[] getPartNames(String[] serialNumber)
{...}
```

For reference information on using the <xm:multiple> attribute, see <xm:multiple> Attribute.

Related Topics

Matching XML Shapes

<xm:multiple> Attribute

<xm:bind> Attribute

Binding to Java Data Members

You can use the . (dot) operator to assign the values of Java data members to XML, and vice versa. In other words, in addition to using the . operator for public data members, XML maps enable you to use it with accessor pairs.

For each of the examples below, you can write a map that accesses the data using dot notation syntax like the following:

```
<book_data>
    <book_title>{Book.title}</book_title>
    <in_print>{Book.isInPrint}</in_print>
</book_data>
```

- Public data members such as fields, declared as follows:

```
public class Book
{
    public String title;
    public boolean inPrint;
}
```

- Non−Boolean and boolean data exposed as properties through public accessor methods:

```
public class Book
{
    public String getTitle
    public void setTitle(String newTitle)
    public boolean isInPrint
    public setInPrint(boolean f)
}
```

Accessing Data Structures in Return Values

When you are creating a return−xml map with a return type that contains structured data (such as an array, a class with public fields, or a class that provides paired get and set methods), you can map the individual members just as you would with a parameter−xml map. The following example illustrates how to parse the data members of an object to individual XML elements.

```
/*
 * @jws:operation
 * @jws:return-xml xml-map::
 *     <book>
 *         <title>{return.title}</title>
 *         <isbn>{return.isbn}</isbn>
 *         <price>{return.price}</price>
 *     </book>
 * ::
 */
public BookDataControl.BookData getPriceByISBN(String ISBN)
{
    /* Code to query a book inventory database using the BookDataControl database control and r
        a BookData object containing data about the book. The BookData object exposes
        public data members for title, isbn, and price. */
}
```

Related Topics

Matching XML Shapes

Database Control: Using a Database from Your Web Service

Making Simple Substitutions Using Curly Braces

Declaring Variables with <xm:bind>

You can use the <xm:bind> attribute to declare a new variable for use in an XML map. Note that the variable you declare with <xm:bind> is available only within the scope of the element in which you declare it, and children of that element.

The following example declares a new Address variable a and binds it to the address within the method's customerData parameter. Because it is declared in the <address> element, the new variable is available to the <street> and <zip> elements, which are its children.

```
/**
 * @jws:operation
 * @jws:return-xml xml-map::
 *     <customer>
 *         <name>{String customerData.name}</name>
 *         <address xm:bind="Address a is customerData.address">
 *             <street>{a.street}</street>
 *             <zip>{a.zip}</zip>
 *         </address>
 *     </customer>
 *     ::
 */
public void addCustomerData(MyStructure customerData)
{
    System.out.println("Customer name is " + customerData.get("name"));
    System.out.println("Customer zipcode is " +
        ((Address)customerData.get("address")).zip);
}
```

For reference information on <xm:bind>, see <xm:bind> Attribute.

Related Topics

Matching XML Shapes

Binding to Java Data Structure Members

<xm:bind> Attribute

Namespaces in XML Maps

In XML, a namespace defines the scope in which tag names should be unique. For introductory information on namespaces, see Introduction to XML.

You can declare and use your own namespaces within an XML map. Namespaces provide a way for you to ensure that element names are unique within a given XML document. For example, note that in the following example the use of namespaces (and their accompanying prefixes) allows two tags named "value" adov−−> <biblio:value> and <xm:value> adov−−> to coexist in the same document. (The namespace specified by the xm prefix is implicitly declared in all files where maps may be used, but it may be overridden by another prefix you define.)

```
/*
 * @jws:operation
 * @jws:parameter-xml xml-map::
 *     <biblio:book xmlns:biblio="http://myBookNamespace.org/">
 *         <biblio:title>{productName}</biblio:title>
 *         <biblio:isbn>{productID}</biblio:isbn>
 *         <biblio:value>{productPrice}</biblio:value>
 *     </biblio:book>
 * ::
 */
```

Related Topics

Matching XML Shapes

Introduction to XML

Filtering By Namespace

Simplifying Maps for Optional Elements

Because the rules behind XML maps allow for multiple overlapping assignments, you can simplify your maps when handling alternate or optional elements. For example, consider the following XML snippet, which contains common <name> and <address> elements, but different elements for the postal code value.

```
<address>
    <name>Don Rumsfeld</name>
    <usZipCode>52332</usZipCode>
</address>
<address>
    <name>Tony Blair</name>
    <britishPostalCode>4F3−G5H</britishPostalCode>
</address>
```

When you expect to receive a message that will contain one or the other, you can simplify your XML map as follows:

```
/*
  * @jws:parameter-xml xml-map::
  * <getFullAddress>
  * <address>
  *     <name>{name}</name>
  *     <usZipCode>{postalCode}</usZipCode>
  *     <britishPostalCode>{postalCode}</britishPostalCode>
  * </address>
  * </getFullAddress>
  * ::
  */
public int getFullAddress(String name, String postalCode)
{...}
```

In this example, either the value for <usZipCode> or for <britishPostalCode> will be mapped to the postalCode parameter, depending on which the message contained.

Assignment Order for Alternate Mapping

Note that if both the <usZipCode> and <britishPostalCode> elements included content in the same message, the last one in the order would overwrite the previous (from the preceding example, <britishPostalCode> would overwrite <usZipCode>. This is just what would occur if you assigned two values to the same variable in Java code: the first would be displaced by the second.

Related Topics

Matching XML Shapes

Making Simple Substitutions Using Curly Braces

Getting Started with Script for Mapping

This section introduces ECMAScript as it can be used to translate XML to and from the Java types used by your web services. If you aren't already familiar with XML maps, see Why Use XML Maps?

Topics Included in This Section

Using Script Functions From XML Maps

Describes the role of ECMAScript in translating between XML and Java.

Creating and Using XML Variables

Introduces the XML and XMLList data types, and shows how you can create XML variables.

How Do I: Create a Script for Use with a Web Service?

Provides a step−by−step introduction to creating an

How Do I: Use Script from a Web Service?

Describes how to connect script functions you write to a JWS file.

Setting Environment Attributes for XML in ECMAScript

Lists attributes you can use to control XML output from ECMAScript.

Type Support in ECMAScript

Describes how types are converted between Java and ECMAScript when using JSX files.

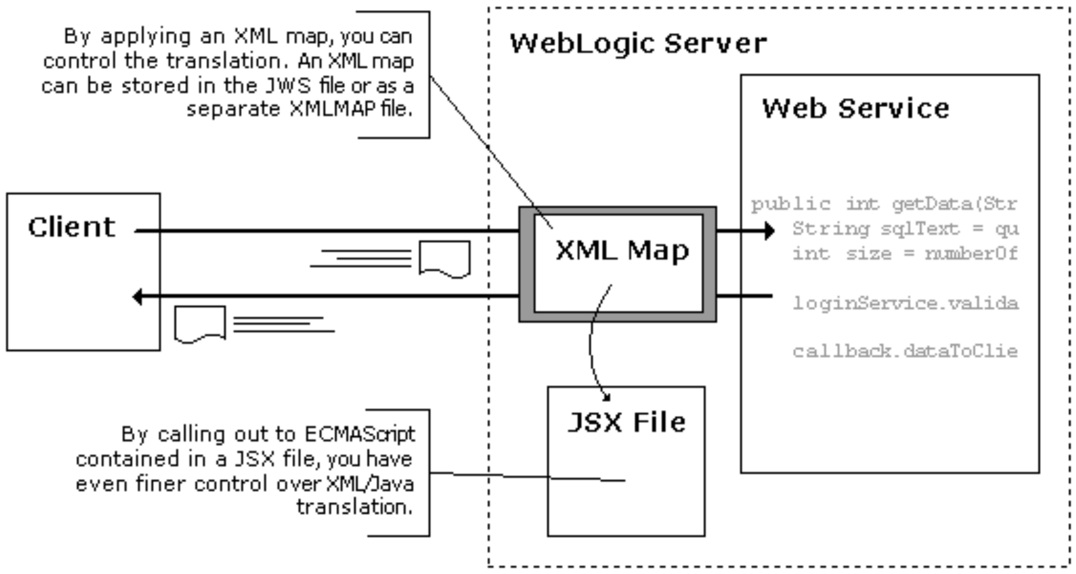
Related Topics

[Why Use XML Maps?]

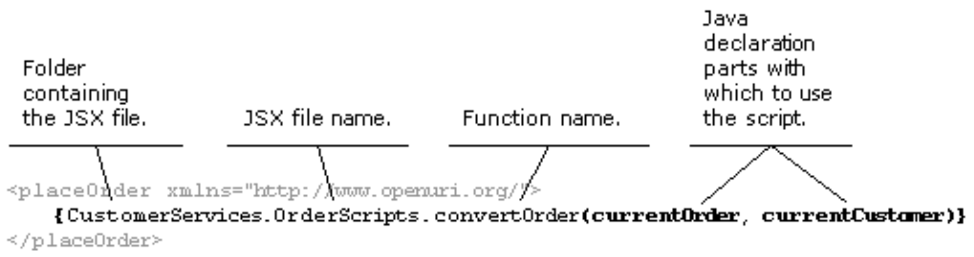
[Getting Started with XML Maps]

Using Script Functions From XML Maps

Just as with XML maps you create, ECMAScript you write for mapping acts as a translation layer. For example, in a map designed to translate an incoming XML message to Java, you can include a reference to a script function that is designed to do a calculation, separate or combine pieces of data, and so on. Some tasks, such as calculations, can't be done in XML maps without script. And sometimes the shape of the XML message differs so dramatically from the shape suggested by the Java declaration that there is no easy way to map them except by enhancing the map with script.

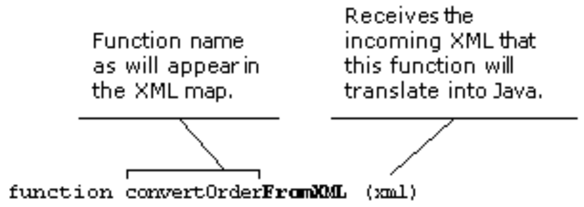

By applying an XML map, you can control the translation. An XML map can be stored in the JWS file or as a separate XMLMAP file.
WebLogic Server
Web Service
Client
XML Map
public int getData(Str
String sqlText = qu
int size = numberOf
loginService.valida
callback.dataToClie
JSX File
By calling out to ECMAScript contained in a JSX file, you have even finer control over XML/Java translation.

You store an ECMAScript function for mapping in a JSX file. From your XML map you point to the function. A map with a reference to a script function might look like this example:


Folder containing the JSX file.
JSX file name.
Function name.
Java declaration parts with which to use the script.

```
<placeOrder xmlns="http://www.openuri.org/">
    {CustomerServices.OrderScripts.convertOrder(currentOrder, currentCustomer)}
</placeOrder>
```

A script function for mapping is designed to map an incoming XML message to Java or to map Java to outgoing XML adov−−> but not both. The function referred to by this example maps XML to Java. In fact, the function's actual name (in the JSX file) is convertOrderFromXML. At run time, WebLogic Server invokes the function when translating data from XML. Here is what the function's actual declaration looks like:

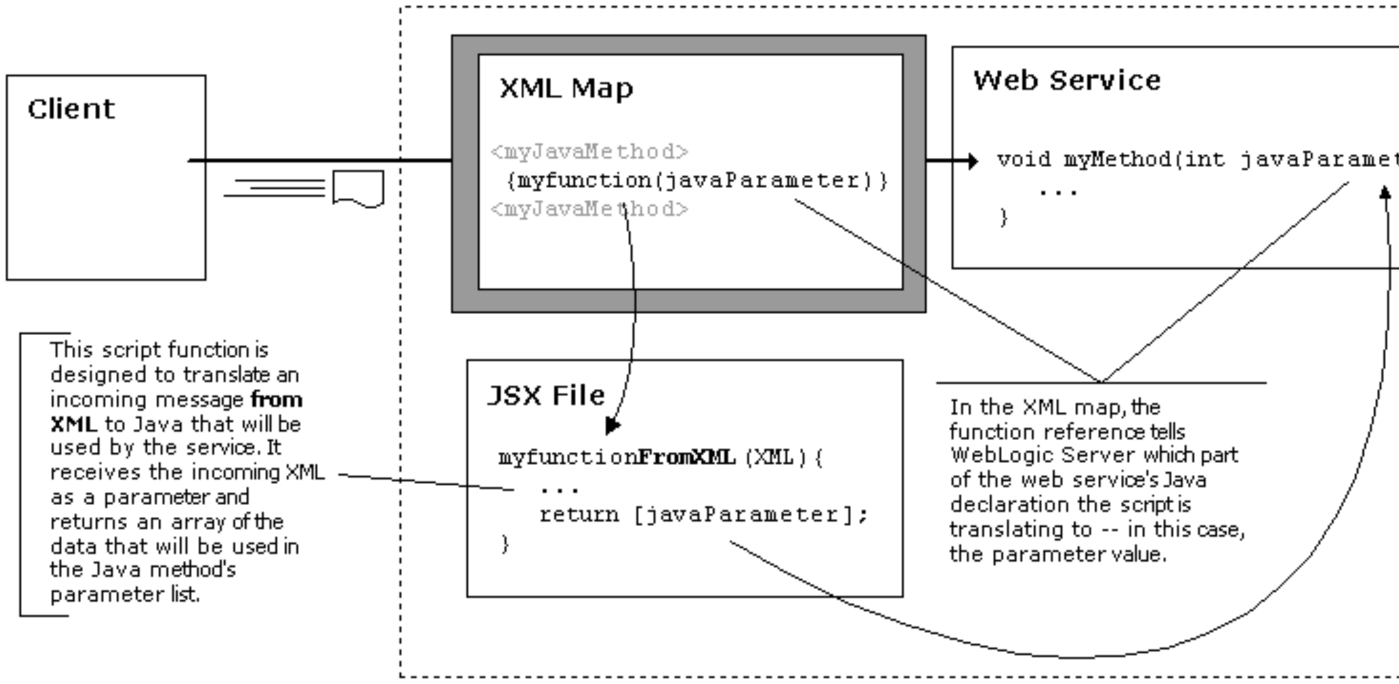

Function name as will appear in the XML map.
Receives the incoming XML that this function will translate into Java.
function convertOrder**FromXML** (xml)

Notice anything unusual? You probably noticed that the single xml parameter in this function declaration doesn't match the items in parenthesis (currentOrder and currentCustomer) in the convertOrder example above. In the example, the items in parenthesis tell WebLogic Server which parts of the Java declaration the script should handle; currentOrder and currentCustomer are parameters of the Java method to which this XML map applies. If this map were designed to handle the method's return value adov−−> say, to map it to an outgoing XML message adov−−> the item in the parenthesis would simply be return. The function itself is more straightforward:

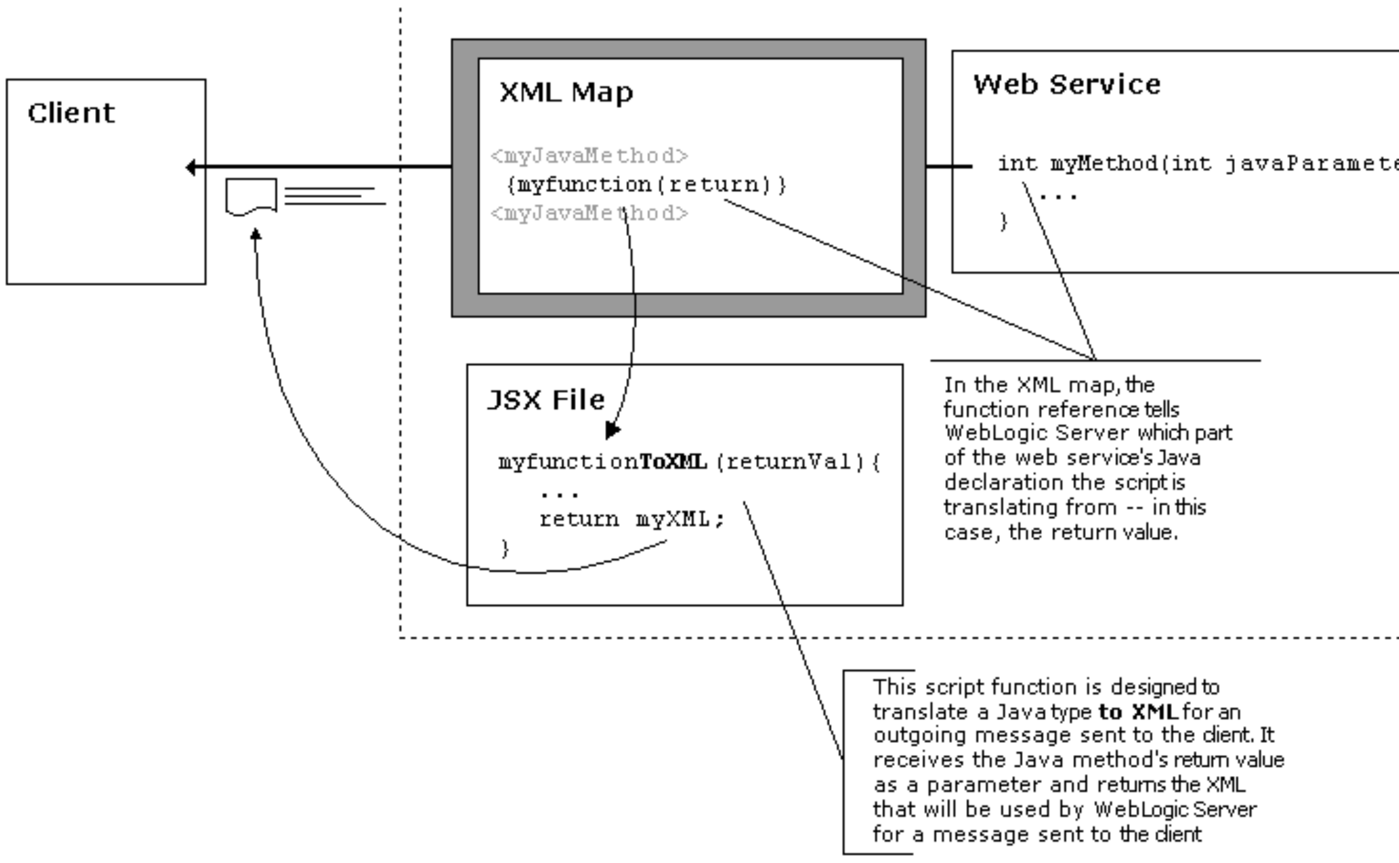
- If the function maps XML to Java, the parameter is XML and the function's return value is an array containing the types to be fed to the Java code.
- If the function maps Java to XML, its parameter is the set of types (one or more) from the Java declaration and its return value is the outgoing XML.

If this is a little confusing, the following diagrams might help. The first one shows mapping for parameters of a method in a web service; the second show mapping for the method's return value. In the first, XML is translated to Java; in the second, Java is translated to XML.

Using Script to Map Parameters on a JWS Method


Client
XML Map
<myJavaMethod>
{myfunction(javaParameter)}
<myJavaMethod>
Web Service
void myMethod(int javaParamet
...
}
JSX File
myfunction**FromXML** (XML) {
...
return [javaParameter];
}
This script function is designed to translate an incoming message **from XML** to Java that will be used by the service. It receives the incoming XML as a parameter and returns an array of the data that will be used in the Java method's parameter list.
In the XML map, the function reference tells WebLogic Server which part of the web service's Java declaration the script is translating to -- in this case, the parameter value.

Using Script to Map the Return Value on a JWS Method


Client
XML Map
<myJavaMethod>
{myfunction(return)}
<myJavaMethod>
Web Service
int myMethod(int javaParamete
...
}
JSX File
myfunctionToXML(returnVal){
...
return myXML;
}
In the XML map, the function reference tells WebLogic Server which part of the web service's Java declaration the script is translating from -- in this case, the return value.
This script function is designed to translate a Java type to XML for an outgoing message sent to the client. It receives the Java method's return value as a parameter and returns the XML that will be used by WebLogic Server for a message sent to the client

Related Topics

[Why Use XML Maps?]

Creating and Using XML Variables

Using the ECMAScript extensions included with WebLogic Workshop, creating a variable for handling XML is as simple as the following:

```
var myXML = <employees>
        <employee id="111111111">
            <firstname>John</firstname>
            <lastname>Walton</lastname>
            <age>25</age>
        </employee>
        <employee id="222222222">
            <firstname>Sue</firstname>
            <lastname>Day</lastname>
            <age>32</age>
        </employee>
    </employees>;
```

When you create a variable and assign it a value that begins with a < symbol, the value is automatically interpreted as XML.

In other words, it is not necessary to create an instance of an XML parser or to create a document instance conforming to a Document Object Model (DOM). These extra steps, common to other XML programming models, are unnecessary. Instead, you create a variable either by assigning a literal XML value (as above) or by creating an XML variable using the new operator:

```
var myXMLVariable = new XML("<employee id='111111111'><firstname>John</firstname></employee>");
```

You can use a variable containing XML to access and manipulate the XML as you would other types that contain hierarchical or listed data, such as collections or arrays. For example, the following code uses the myXML variable to change John's name to Roger:

```
/* Change the <firstname> value of the first employee to Roger. */
myXML.employees.employee[0].firstname = "Roger";
```

Data Types for XML

Under the covers, there are actually two new data types at work: XML and XMLList. As with many other types in ECMAScript, you do not need to explicitly declare these types. But code you write will create and manipulate them implicitly depending on the code.

XML Data Type

In general, an XML variable represents XML that has a root, such as the preceding myXML variable. The root element of this variable is the <employees> element adov−−> it contains the rest of the XML. Querying the myXML variable for one of the employees also returns an XML variable, as in the following example:

```
/* Create an XML variable from the first employee element. */
var anEmployee = myXML.employees.employee[0];
```

The content of the resulting anEmployee variable looks like this:

```
<employee id="111111111">
    <firstname>John</firstname>
    <lastname>Walton</lastname>
    <age>25</age>
</employee>
```

XMLList Data Type

An XMLList variable, on the other hand, generally represents XML that has no actual root. Without the <employees> tags, for example, myXML would be an XMLList. If you wanted to create an XMLList variable, you could do it as follows:

```
var myXMLList = <>
    <employee id="111111111">
            <firstname>John</firstname>
            <lastname>Walton</lastname>
            <age>25</age>
        </employee>
        <employee id="222222222">
            <firstname>Sue</firstname>
            <lastname>Day</lastname>
            <age>32</age>
        </employee>
</>;
```

Note that the value assigned to the variable here doesn't have an actual root, but an anonymous root expressed as <> and </>. Because it has no actual root, the XML in myXMLList is known as an XML fragment.

Just as a query can return an XML type, a query for a particular piece of XML can also return an XMLList. By not specifying an index number for an <employee>, the following line of code returns all of the

<employee> elements:

```
var listOfEmployees = myXML.employees.employee;
```

The resulting XML looks like the following (note the absence of a single containing element):

```
<employee id="111111111">
    <firstname>John</firstname>
    <lastname>Walton</lastname>
    <age>25</age>
</employee>
<employee id="222222222">
    <firstname>Sue</firstname>
    <lastname>Day</lastname>
    <age>32</age>
</employee>
```

One of the most useful aspects of the XMLList type is the ability to access its contents iteratively. The XMLList type is returned when you query for particular parts of the XML using the . (dot) operator. For example, using the myXML variable in the first example, you could write code such as the following:

```
/* Create an XMLList containing the <employee> elements. */
var listOfEmployees = myXML.employees.employee;
for (var e in listOfEmployees) {
    // Create a new variable with the content of the <age> element converted to a number.
    var age = new Number(e.age);
    // Raise everyone's age by 1 year.
    age += 1;
    // Assign the new age value to the content of the element.
    e.age = age;
}
```

Notes About Using the Data Types for XML

- The variable name corresponds to the container for the XML, not to the XML's root.

When accessing XML through a variable, remember that the highest−level element in the XML itself is just beneath the variable in the hierarchy. Compare the following two lines of code:

/* This is how it's done! */ var johnFirstName = myXML.employees.employee.firstname; /* This won't return the correct result! */ var johnFirstName = myXML.employee.firstname;

- The script interpreter will read PIs and comments, but does not preserve them in the XML value adov−−> they are silently ignored.
- The script interpreter does not support literal XML values that do not contain at least one element.

Related Topics

Accessing Element Children With the . Operator

Using Script Functions From XML Maps

How Do I: Create a Script for Use with a Web Service?

You can create ECMAScript functions to customize WebLogic Server's translation between XML messages your service receives or sends and your Java code. You use the script by placing a reference to it in your JWS file. At run time, WebLogic Server uses the script for translation. Scripts you create are stored in files with a JSX extension.

The following steps describe what to do at a high level, and offer links to more detailed information.

1. Open the JSX file that will contain your new script function, or create a new file.
2. Write a function to translate incoming XML for use by your Java code, or write a function that will translate your Java data into outgoing XML.
3. Connect the script function to your Java code in a JWS through an XML map.

The following procedures offer step−by−step instructions on the basics of writing an ECMAScript function for translating XML and Java.

To Create a New JSX File

1. Locate the folder in your project where you will store the script (JSX) file.

Where you put the file depends on how your project is structured. For example, you might put it in the same folder as a JWS file that will be using it. You could also create a separate, or subordinate, "scripts" folder where you store all JSX files for this project.

2. Right−click the folder, then click New File.
3. In the Create New File dialog, click JavaScript.
4. In the File name box, enter the name of your script file.

Note that the File extension box contains "jsx" by default for script files.

5. Click OK.

The new script file should open in Source View. The Project Tree should list the file you have just created.

To Create ECMAScript for Translating to XML from Java

1. Open the JSX file to which you will be adding a new function.

Note: If you are working with a JSX file you have just created, there should already be a placeholder for this function.

2. If needed, use the import statement to import Java classes for use in your ECMAScript code.

This statement is similar to the import directive in Java. You will need to import any Java classes that will be used by your script. For more information about import, see Importing Java Classes to ECMAScript with the import Statement.

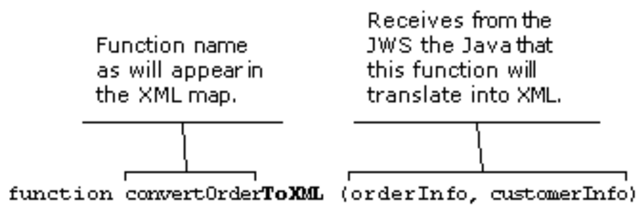
```
import mypackage.MyOuterClass.MyClass;
```

3. Declare a function for translating to XML.

The function declaration can have any name you like, but it:

- Must end with the phrase "ToXML".
- Must have a parameter representing each part of the Java declaration (in the JWS file) that this script will use for translating to XML.

For example, the following illustration describes a function declaration designed to receive two Java parameters:


Function name as will appear in the XML map.
Receives from the JWS the Java that this function will translate into XML.
function convertOrderToXML (orderInfo, customerInfo)

You will refer to your function from an XML map in a JWS file. This illustration, for example, could receive two parameters from a callback to a client or method on a ServiceControl.

For more detail on connecting script functions to a JWS file, see How Do I: Use Script from a Web Service?

4. Add ECMAScript code that will use the data represented by the parameters to form XML.

Use the extended version of ECMAScript to make creating and handling XML easier. For example, you might:

- Declare an XML variable. For more information, see Creating and Using XML Variables.
- Assign data from your parameters to the elements and attributes of the XML contained in the XML variable.

5. At the end of the function, return the XML variable your code has created.

The XML you return will be added to the message sent by WebLogic Server. For example, a simple script that creates XML and adds two values to it might look like the following:

/* Create an XML variable. */ var orderMessage = <order/>; /* Add two child elements, inserting values from parameters received by the script. */
orderMessage.appendChild(<item_name>{orderInfo.name}</item_name>);
orderMessage.appendChild(<customer_name>{customerInfo.name}</customer_name>): /* Return the new XML */ return orderMessage;

After you have created your script, you use it by connecting it to your JWS file through an XML map. For more information, see How Do I: Use Script from a Web Service?

To Create ECMAScript for Translating from XML to Java

1. Open the JSX file to which you will be adding a new function.

Note: If you are working with a JSX file you have just created, there should already be a placeholder for this function.

2. If needed, use the import statement to import Java classes for use in your ECMAScript code.

This statement is similar to the import directive in Java. You will need to import any Java classes that will be used by your script. For more information about import, see Importing Java Classes to ECMAScript with the import Statement.

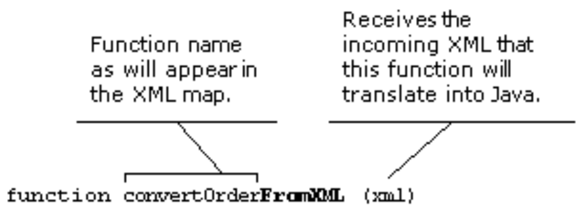
import mypackage.MyOuterClass.MyClass;

3. Declare a function for translating from XML to Java.

The function declaration can have any name you like, but it:

- Must end with the phrase "FromXML".
- Must have a parameter representing the incoming XML that this script will translate to Java.

For example, the following illustration describes a function declaration designed to receive the XML:


Function name as will appear in the XML map.
Receives the incoming XML that this function will translate into Java.
function convertOrderFromXML (xml)

You will refer to your function from an XML map in a JWS file. For more detail on connecting script functions to a JWS file, see How Do I: Use Script from a Web Service? For more information on how this kind of function is referenced from an XML map, see Using Script Functions From XML Maps.

4. Add ECMAScript code that will use the incoming XML represented by the function's parameter to create the data needed by the JWS file.

Use the extended version of ECMAScript to make handling XML easier. For example, you might:

- Declare variables to hold the data that will be needed by your JWS code. These will be returned by your function.
- Access the xml parameter using the . operator, as described in Accessing Element Children With the . Operator. Or see information on related operators in Summary of ECMAScript Language Extensions.
- Assign data from the xml variable to variables that will be returned by the function.

5. At the end of the function, return an array containing the variables needed by your Java code.

The variables you return will be passed to your Java declaration. For example, a simple script that accesses XML and extracts two values from it might look like the following:

/* Create variables you will later return to your Java declaration. */ var orderInfo = xml.order.order_info; var customerInfo = xml.order.customer_info; /* Return an array containing the variables. */ return [orderInfo, customerInfo];

After you have created your script, you use it by connecting it to your JWS file through an XML map. For more information, see How Do I Use Script from a Web Service?

Related Topics

How Do I: Use Script from a Web Service?

Summary of ECMAScript Language Extensions

Creating and Using XML Variables

How Do I: Use Script from a Web Service?

You can connect an ECMAScript function to your web service when you want to use the function for translating XML messages to Java or vice versa. When you have created the function in a JSX file, you connect the function to your web service by referencing it within an XML map.

Note: For step−by−step information on creating the script function itself, see How Do I: Create a Script for Use with a Web Service?

The procedures in this topic cover how to use script with parameters, or use script with a return value.

To Use Script with Parameters

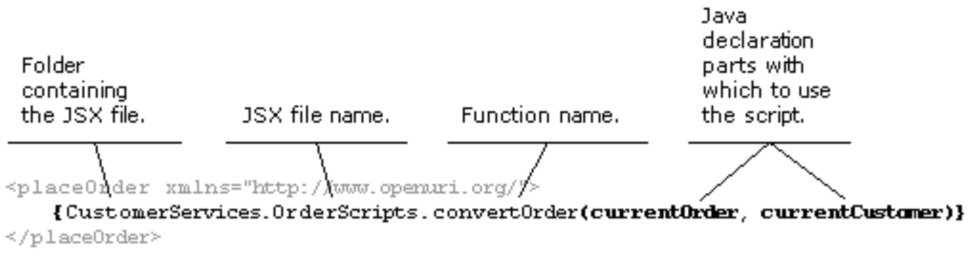
1. In Design View, double−click the arrow corresponding to the item (method, callback, or callback handler) with which the script function will be used. The Edit Maps and Interface dialog appears.
2. In the top pane, locate the place in your XML map where you will reference your script function.

This will vary depending on how you are using script.

- For example, your script might translate the parameter values into only part of an outgoing XML message. In that case you may want to include the script reference as one line of a larger XML map adov−−> the XML returned from the function will be inserted where the reference is.
- On other other hand, if your script translates from incoming XML to the parameter values, it may be easier to replace the entire map with the function reference. In other words, the reference would be the only text in the root element (the tags containing the name of the Java method or callback).

3. In the top pane, enter a reference to your script function.

Your function reference should have a syntax similar to the following:


Folder containing the JSX file.
JSX file name.
Function name.
Java declaration parts with which to use the script.
<placeOrder xmlns="http://www.openuri.org/">
{CustomerServices.OrderScripts.convertOrder(currentOrder, currentCustomer)}
</placeOrder>

The items in parenthesis after the function name must correspond in name and order to the Java parameters that the script will use:

<placeOrder xmlns="http://www.openuri.org/"> <order_info>
{CustomerServices.OrderScripts.convertOrder(currentOrder, currentCustomer)} </order_info>
<customer_id>{currentCustomer.custID}</customer_id> </placeOrder>

This example might correspond to the following Java declaration:

public int placeOrder(Order currentOrder, Customer currentCustomer) {...}

4. Click OK to close the Edit Maps and Interface dialog.

To Use Script with a Return Value

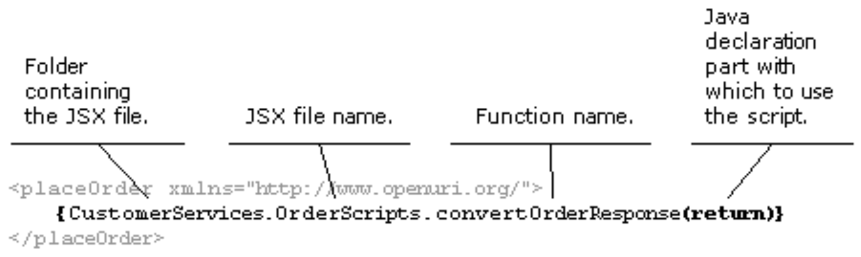
1. In Design View, double−click the arrow corresponding to the item (method, callback, or callback handler) with which the script function will be used. The Edit Maps and Interface dialog appears.
2. In the top pane, locate the place in your XML map where you will reference your script function.

This will vary depending on how you are using script.

- For example, your script might translate the return value into only part of an outgoing XML message. In that case you may want to include the script reference as one line of a larger XML map adov−−> the XML returned from the function will be inserted where the reference is.
- On other other hand, if your script translates from incoming XML to the return value, it may be easier to replace the entire map with the function reference. In other words, the reference would be the only text in the root element (the tags containing the name of the Java method or callback).

3. In the top pane, enter a reference to your script function.

Your function reference should have a syntax similar to the following:


Folder containing the JSX file.
JSX file name.
Function name.
Java declaration part with which to use the script.

```
<placeOrder xmlns="http://www.openuri.org/">
    {CustomerServices.OrderScripts.convertOrderResponse(return)}
</placeOrder>
```

The item in parenthesis after the function name must be "return", as in the following example:

<placeOrderReponse xmlns="http://www.openuri.org/">   <order_confirmation>
      {CustomerServices.OrderScripts.convertOrderResponse(return)}   </order_confirmation>
</placeOrderResponse>

This example might correspond to the following Java declaration:

public int placeOrder(Order currentOrder, Customer currentCustomer) {...}

4. Click OK to close the Edit Maps and Interface dialog.

Function References May Not Have Siblings

The function reference (everything between and including the curly braces) must be the only child of its parent element. For example, the following XML map fragment will cause an error:

```
<placeOrder xmlns="http://www.openuri.org/">
    {CustomerServices.OrderScripts.convertOrder(currentOrder, currentCustomer)}
    <customer_info>{currentCustomer}</customer_info>
</placeOrder>
```

But something like the following would work:

```
<placeOrder xmlns="http://www.openuri.org/">
    <order_info>
        {CustomerServices.OrderScripts.convertOrder(currentOrder, currentCustomer)}
    </order_info>
    <customer_id>{currentCustomer.custID}</customer_id>
</placeOrder>
```

Related Topics

How Do I: Create a Script for Use with a Web Service?

Using Script Functions From XML Maps

Setting Environment Attributes for XML in ECMAScript

When you are using ECMAScript to handle XML, you can tailor how the parser renders your output. In particular, there are three global attributes you may be interested in setting. To set these, simply type or paste them in at the top of your JSX file with the values you want.

Global Environment Attributes

prettyIndent adov−−> Specifies the number of spaces each child of your XML will be indented when printed to the console.

Usage: XML.environment.@prettyIndent = 2;

prettyPrint adov−−> Specifies whether the your XML should be arranged with indents and line breaks when printed to the console.

Usage: XML.environment.@prettyPrint = true;

whitespace adov−−> Specifies whether whitespace characters should be removed from the beginning or end of any textnode. Whitespace is defined as:

- '\t' \u0009 HORIZONTAL TABULATION
- '\n' \u000A NEW LINE
- '\f' \u000C FORM FEED
- '\r' \u000D CARRIAGE RETURN
- ' ' \u0020 SPACE
- Any character who's ASCII value <= 0x20 (including control characters)

Usage: XML.environment.@whitespace = false;

Related Topics

Getting Started with Script for Mapping

Type Support in ECMAScript

When you add an XML map that references an ECMAScript function, data is converted between native ECMAScript types and native Java types. For example, consider an XML map translating an incoming XML message to your Java types through ECMAScript you've written. In this case WebLogic Server may be converting data from its format in script to the required Java format.

This topic describes some aspects of this conversion you might want to be aware of for conversions made in a function ending "FromXML". Remember that in this case, the data moves from incoming XML to ECMAScript in a JSX file to Java code in a JWS file.

Note that the other caseadov−−>from Java through a "ToXML" function to XMLadov−−>is much simpler. Because the result is simply XML, there aren't any conversion issues to be aware of.

Conversions During Mapping from XML to Java

The following lists specific kinds of conversion you should be aware of when using ECMAScript for mapping. Each item in the list describes conversion for a particular type or category of types.

- The type in Java is the same as the type in ECMAScriptadov−−>There are no limitations when there is no type conversionadov−−>when the type used in script is the same as the type used in the JWS file. (Remember that you can import Java classes for use in script using the import statement. For more information, see Importing Java Classes to ECMAScript with the import Statement.)
- Java primitive typeadov−−>There are no limitations for each of the Java classes corresponding to primitive types, including the following: Integer, String, Float, Double, Boolean, Character, Byte, Long, Short, and Date.

Note: To declare and use any Java types in ECMAScript (including the wrapped primitive types such as Integer and so on), you must import them using the import statement. For more information, see Importing Java Classes to ECMAScript with the import Statement.

- Implicit ECMAScript typesadov−−>ECMAScript types you declare with the var statement (including boolean, number, object, string, and so on) are converted to the corresponding types in the java.lang package before they reach your Java code.
- Java arraysadov−−>Java arrays based on types in the java.lang package are carried without conversion from the ECMAScript function to the Java code. ECMAScript arrays, on the other hand, are converted to corresponding Java types.

Related Topics

Type Support in XML Maps

Handling XML with ECMAScript Extensions

WebLogic Workshop includes an extended version of the ECMAScript (also known as JavaScript) programming language with enhanced support for handling XML. In particular, the extended ECMAScript includes native support for XML as a type you can access as you would access other data structures. The ECMAScript extensions are especially useful when you are manipulating the shape of XML messages received and sent by your web service.

Topics Included in This Section

Accessing Element Children with the . Operator

Describes how you can use the . (dot) operator to construct a complete path to a child of a given element just as you would specify a member of a collection object.

Accessing Element Descendants with the .. Operator

Describes how you can use the .. (double dot) operator to specify any child contained by the element left of the operator, regardless of how far down the hierarchy the child is.

Accessing Element Children Through Their Index

Describes how you can access the children of an element by specifying their index in the list.

Accessing Element Children Iteratively

Describes how you can access element children in a loop structure.

Accessing Attributes with the .@ Operator

Describes how you can access attributes.

Resolving XML Dynamically with Embedded Expressions

Describes how you can use {} (curly braces) to embed variables and expressions in XML.

Filtering Multiple Children with Predicates

Describes how you can filter for particular elements in a list.

Filtering By Namespace

Describes how you can access elements based on their namespace URI.

Inserting Elements with the += Operator

Using the += operator, you can insert an XML element after another element. This can be useful when you want to insert a new element into an existing list.

Combining XML With the + Operator

You can use the + operator to combine XML elements. This can be useful when you want to create a new list of XML elements from XML elements returned from another expression.

Removing Elements and Attributes with the delete Operator

You can use the delete operator to remove specified elements and attributes.

Specifying the Current XML with the thisXML Keyword

You can use the thisXML property to specify the current XML, such as XML returned from an expression. The thisXML property works in a manner similar to the this keyword in ECMAScript, which is used to specify objects or functions from within their own code.

Importing Java Classes with ECMAScript with the import Statement

You can use the import statement to specify Java classes that you may use in ECMAScript.

Related Topics

Handling and Shaping XML Messages with XML Maps

Accessing Element Children With the . Operator

You can use the . (dot) operator to construct a path to an element just as you would specify a member of a collection object. The . operator examines the children of its left operand and returns those with names that match its right operand. It returns them in the order in which they appear in the document.

The following example illustrates using the . operator to return a single value or an element.

```
/* Declare an XML variable with a literal XML value. */
var xmlEmployees = <employees>
    <employee id="111111111">
        <firstname>John</firstname>
        <lastname>Walton</lastname>
        <age>25</age>
    </employee>
    <employee id="222222222">
        <firstname>Sue</firstname>
        <lastname>Day</lastname>
        <age>32</age>
    </employee>
</employees>;
/*
 * Return the first name of the first employee in the list.
 * This code returns an XMLList type containing "John".
 */
var name = xmlEmployees.employees.employee[0].firstname;
/*



 * Return the first <employee> element.
 * This code returns an XML type with all of the first <employee> element:
 *    <employee id="111111111">
 *        <firstname>John</firstname>
 *        <lastname>Walton</lastname>
 *        <age>25</age>
 *    </employee>
 */
var firstEmployee = xmlEmployees.employees.employee[0];
```

Using the . operator to specify a repeating element will return an XMLList with multiple items, as in the following example.

```
/*
 * Return all of the <employee> elements.
 * This code returns an XMLList containing the following:
 *    <employee id="111111111">
 *        <firstname>John</firstname>
 *        <lastname>Walton</lastname>
 *        <age>25</age>
 *    </employee>
 *    <employee id="222222222">
 *        <firstname>Sue</firstname>
 *        <lastname>Day</lastname>
 *        <age>32</age>
 *    </employee>
 */
var employees = xmlEmployees.employees.employee;
```

You can also assign new values using the . operator, as in the following example.

```
/*
 * Change John's name to Biff.
 */
xmlEmployees.employees.employee[0].firstname = "Biff";
/*
 * Set the entire first <employee> element to a new value.
 */
xmlEmployees.employees.employee[0] = <employee id="111111111">
    <first>Armin</first>


    <last>HooHaw</last>


    <age>91</age>


</employee>;
```

When elements have names that conflict with ECMAScript keywords or function names, you can use the following syntax instead. Note that there are two different workarounds adov−−> one is for use with ECMAScript keywords and the other is for use with function names.

```
/*
 * Return all of the <if> elements.
 * This avoids a conflict with the "if" ECMAScript keyword.
 */
var ifs = xmlData.product_description["if"];
/*
 * Return the first <parent> element.
 * This avoids a conflict with the parent function.
 */
var firstParent = xmlFamilies.kid.::parent[0];
```

Related Topics

Accessing Element Descendants with the .. Operator

Accessing Element Descendants With the .. Operator

You can use the .. (double dot) operator to specify any child contained by the element left of the operator, regardless of how far down the hierarchy the child is. This gives you easy access to descendents (children, grandchildren, and so on) of an element. Through this operator, you can avoid having to build a complete path to an element that may be deeply nested.

The .. operator examines all of the descendent elements of its left operand and returns those with names that match its right operand, and returns them in the order in which they appear in the document. When the left operand is a list of elements, each of these elements is examined according to their order in the document.

The following example illustrates using the .. operator to return a single value or an element.

```
/* Declare an XML variable with a literal XML value. */
var xmlEmployees = <employees>
    <employee id="111111111">
        <firstname>John</firstname>
        <lastname>Walton</lastname>
        <age>25</age>
    </employee>
    <employee id="222222222">
        <firstname>Sue</firstname>
        <lastname>Day</lastname>
        <age>32</age>
    </employee>
</employees>;
/*
 * Return the first name of the first employee in the list.
 * This code returns an XMLList type containing "John".
 */
var name = xmlEmployees..employee[0].firstname;
/*
 * Return the first <employee> element.
 * This code returns an XML type with all of the first <employee> element:
 *    <employee id="111111111">
 *        <firstname>John</firstname>
 *        <lastname>Walton</lastname>
 *        <age>25</age>
 *    </employee>
 */
var firstEmployee = xmlEmployees..employee[0];
```

When elements have names that conflict with function names, you can use the following syntax instead. Note that there is currently no workaround for using the .. operator when its right operand conflicts an ECMAScript keyword.

```
/*
 * Return the first <parent> element.
 * This avoids a conflict with the parent function.
 */
var firstParent = xmlFamilies..::parent[0];
```

Related Topics

Accessing Element Children Iteratively

Accessing Element Children Through an Index

Accessing Element Children Through Their Index

You can use the [] operator to access the children of an element as if they were members of an array. The children are indexed in the order in which they appear in the document, with the first member starting at 0.

The following example illustrates using the [] operator to return a single item in an XMLList.

```
/* Declare an XML variable with a literal XML value. */
var xmlEmployees = <employees>
    <employee id="111111111">
        <firstname>John</firstname>
        <lastname>Walton</lastname>
        <age>25</age>
    </employee>
    <employee id="222222222">
        <firstname>Sue</firstname>
        <lastname>Day</lastname>
        <age>32</age>
    </employee>
</employees>;
/*
 * Return the first name of the first employee in the list.
 * This code returns an XMLList type containing "John".
 */
var name = xmlEmployees.employees.employee[0].firstname;
/*



 * Return the first <employee> element.
 * This code returns an XML type with all of the first <employee> element:
 *    <employee id="111111111">
 *        <firstname>John</firstname>
 *        <lastname>Walton</lastname>
 *        <age>25</age>
 *    </employee>
 */
var firstEmployee = xmlEmployees.employees.employee[0];
```

You can also use multiple bracketed indices to build a path to a specific item, as in the following example:

```
/*
 * Return the last name of the first employee in the list.
 * This code returns an XML type containing <lastname>Walton</lastname>.
 */
var name = xmlEmployees.employees.employee[0][1];
```

Finally, you can enclose an expression in the brackets, rather than a literal number. The following example adds a <phone> element to the end of the information about Sue.

```
/*
 * Return the last name of the second employee in the list.
 * This code returns an XML type containing <lastname>Day</lastname>.
 */
var e = xmlEmployees.employees.employee[1];
/*
 * Use the expression to calculate the number of children in the second <employee> element
 * then add a new XML value at the end.
 */
e[e.children().length] = <phone>foo</phone>;
```

You can also query for specific element through predicate expressions. For more information on predicates, see Filtering Multiple Children with Predicates.

Related Topics

Accessing Element Children Iteratively

Filtering Multiple Children with Predicates

Creating and Using XML Variables

Accessing Element Children Iteratively

You can write code that iteratively accesses an XML element list in a manner similar to accessing an array. Using the XMLList type, the extended ECMAScript interpreter automatically handles repeating elements as if they were members of an array.

```
/* Declare an XML variable with a literal XML value. */
var xmlEmployees = <employees>
    <employee id="111111111">
        <firstname>John</firstname>
        <lastname>Walton</lastname>
        <age>25</age>
    </employee>
    <employee id="222222222">
        <firstname>Sue</firstname>
        <lastname>Day</lastname>
        <age>32</age>
    </employee>
</employees>;
/*
```

```
 * Return the average age of the employees.
 * This code produces an ECMAScript number containing "28.5".
 */
    var intAgeTotal = 0;
    var ageValues = xmlEmployees..age;
    /* Loop through the <age> element values, adding them together. */
    for (a in xmlEmployees..age) {
        intAgeTotal += new Number(a);
    }
    /* Compute the average age. */
    var intAgeAvg = intAgeTotal / ageValues.length;
```

Note: The for...in statement works differently for XML than it does for native ECMAScript arrays. With native ECMAScript arrays, for...in assigns the loop variable over the domain of the array. In other words, the variable represented by a above would be the index of the current item. With XML, for...in assigns the loop variable over the range of the array. In other words, because the example above is operating on an XMLList, a holds the current item's value.

Related Topics

Accessing Element Children Through Their Index

Accessing Attributes With the .@ Operator

In the way that you access elements with the . operator, you can access attributes using the .@ (attribute) operator.

```
/* Declare an XML variable with a literal XML value. */
var xmlEmployees = <employees>
    <employee id="111111111">
        <firstname>John</firstname>
        <lastname>Walton</lastname>
        <age>25</age>
    </employee>
    <employee id="222222222">
        <firstname>Sue</firstname>
        <lastname>Day</lastname>
        <age>32</age>
    </employee>
</employees>;
/*
 * Assign a variable with John's id number.
 * This code returns a string containing "111111111".
 */
var xmlEmployeeID = xmlEmployees.employees.employee[0].@id;
/*
 * Set Sue's id number to "555555555".
 */
xmlEmployees.employees.employee[1].@id = "555555555";
```

Note: The nature of attributes and the .@ operator makes it necessary to use the thisXML property when filtering using the .@ operator:

```
/*
 * Find the <employee> element where the id attribute is 222222222.
 * Return the corresponding last name.
 * This code returns <lastname>Day</lastname>.
 */
var ageXML = xmlEmployees..employee.(thisXML.@id == "222222222").lastname;
```

Related Topics

attribute Function

attributes Function

Resolving XML Dynamically with Embedded Expressions

When working with XML, you can use embedded expressions as a shorthand means to resolve or change element and attribute values. You can use the {} (curly braces) operator to enclose an expression that should be evaluated before the entire string is converted to XML for assignment to a variable. The ECMAScript in the following example restructures the XML in the xmlEmployees variable so that the id value is no longer stored in a child element of <employee>, but in an id attribute of that element.

```
/* Declare an XML variable with a literal XML value. */
var xmlEmployees = <employees>
    <employee>
        <id>111111111</id>
        <firstname>John</firstname>
        <lastname>Walton</lastname>
        <age>25</age>
    </employee>
    <employee>
        <id>222222222</id>
        <firstname>Sue</firstname>
        <lastname>Day</lastname>
        <age>32</age>
    </employee>
</employees>;
/*
 * Loop through the list of employees, restructuring the <employee> element
 * to make the id element an attribute of the <employee> element.
 */
for(e in xmlEmployees..employee){
    var newStructure =
        <employee ssn={e.id}>
            <first_name>{e.firstname}</first_name>
            <last_name>{e.lastname}</last_name>
            <age>{e.age}</age>
        </employee>;
}
```

Note that while you can also use {} to dynamically resolve element names, the value between {} may not be an empty string. For example, this is illegal: <{}>someValue</{}>.

Related Topics

Summary of ECMAScript Language Extensions

Filtering Multiple Children With Predicates

When using the . or .. operator returns multiple children, you can use the .() (predicate) operator to further filter your results based on specific criteria. The .() operator works in a manner similar to the XPath [ ] operator, which constitutes what is known as a "predicate" of the path.

The following function examples, based on this xmlEmployees variable, return information about an employee based on a filtering value.

```
/* Declare an XML variable with a literal XML value. */
var xmlEmployees = <employees>
    <employee id="111111111">
```

```
        <firstname>John</firstname>
        <lastname>Walton</lastname>
        <age>25</age>
    </employee>
    <employee id="222222222">
        <firstname>Sue</firstname>
        <lastname>Day</lastname>
        <age>32</age>
    </employee>
</employees>;
/*
 * Return the <employee> element for the employee whose last name


 * is "Day".
 * This code returns an XMLList type containing:
 *    <employee id="222222222">
 *        <firstname>Sue</firstname>
 *        <lastname>Day</lastname>
 *        <age>32</age>
 *    </employee>
/*
    var empDay = xmlEmployees..employee.(lastname == "Day");



/*
 * Return the age of the employee whose id is "111111111".
 * This code returns an XMLList type containing "25".
 */
var empAge = xmlEmployees..employee.(thisXML.@id == "111111111").age;
```

When elements have names that conflict with ECMAScript keywords or function names, you can use the following syntax for filtering instead. Note that there are two different workarounds adov−−> one is for use with ECMAScript keywords and the other is for use with function names. Both require using the thisXML property.

```
/*
 * Return all of the <if> elements.
 * This avoids a conflict with the "if" ECMAScript keyword.
 */
var ifs = xmlData.product_description.(thisXML["if"] == "43−0t654−09");
/*
 * Return the first <parent> element.
 * This avoids a conflict with the parent function.
 */
var firstParent = xmlFamilies.kid.(thisXML.::parent == "Johnson");
```

Related Topics

Accessing Element Descendants with the .. Operator

Filtering By Namespace

When you expect XML messages with elements that use namespace prefixes, you can use a namespace variable to query the XML for elements contained in a specified namespace.

The syntax for the namespace declaration is as follows:

namespace namespaceVariable [as URIString]

The URIString value is the URI that uniquely identifies the namespace. By associating the variable with the namespace URI, you can then use the variable as you might use a namespace prefix. The syntax for using the namespace variable can be one of the following two variations. The first uses the . operator to separate elements in the hierarchy, which the second uses square brackets.

value = namespaceVariable::elementName1.namespaceVariable::elementName2;

value = ["namespace:elementName1"]["namespace:elementName2"];

The second variation is especially useful in cases where the element name conflicts with a script keyword (such as if or return). For more information about working around such conflicts, see A Few Things to Remember About XML Maps and Script.

In the following example, the "http://openuri.org/" namespace is associated with the myco variable.

```
/* Declare an XML variable with a literal XML value. */
var xmlEmps = <employeeData>
    <inc:employees xmlns:inc="http://openuri.org/">
        <inc:employee inc:id="111111111">
            <inc:name>John</inc:name>
            <inc:age>25</inc:age>
        </inc:employee>
    </inc:employees>
</employeeData>;
/* Declare a namespace variable to represent the namespace whose prefix is "inc". */
namespace myco as "http://openuri.org/";
/* Create a new variable to hold the <employee> node containing John.
 * Access the attributed elements by using the namespace variable in double-colons in place of
 * prefix-colon combination used in the XML literal above.
 */
var xmlName = xmlEmps.employeeData.myco::employees.myco::employee[0];
```

Note that accessing namespaced attributes works a little differently. Rather than using the .@ operator, as you might ordinarily when accessing attributes, you use the attribute method. In other words, instead of .@inc::id to access the inc:id attribute in the example, you would use the following:

```
/* Create a new variable to hold the <employee> node containing John. */
var xmlName = xmlEmps.employeeData.myco::employees.myco::employee.attribute("inc:id");
```

Related Topics

[Accessing Element Children With the . Operator]

Inserting Elements With the += Operator

Using the += operator, you can insert an XML element after another element. This can be useful when you want to insert a new element into an existing list, as in the following example.

```
/* Declare an XML variable with a literal XML value. */
var xmlEmployees =
    <employees>
        <employee>
            <firstname>John</firstname>
            <lastname>Walton</lastname>
            <age>25</age>
        </employee>
        <employee>
            <firstname>Gladys</firstname>
            <lastname>Cravits</lastname>
            <age>53</age>
        </employee>
<employees>;



/* Insert a new node with information about Rob Petrie immediately after the node

    containing information about John. */

xmlEmployees..employee[0] +=

        <employee id = "333333333">

            <firstname>Rob</firstname>

            <lastname>Petrie</lastname>

            <age>34</age>

        </employee>;
```

Note: This operator can't be used in cases where the left operand is an XMLList, as in the following example. This example attempts to insert the new <employee> node at the end of the list by omitting a specific reference to a particular elements (as in the preceding example). To insert an elements as the last in a list, use the appendChild function.

```
/* This causes a run-time error. */
xmlEmployees.employee +=
        <employee>
            <firstname>Sue</firstname>
            <lastname>Day</lastname>
            <age>32</age>
        </employee>;
```

Related Topics

appendChild function

Combining XML With the + Operator

You can use the + operator to combine XML elements. This can be useful when you want to create a new list of XML elements from XML elements returned from another expression. You can also use the + operator to add new XML to existing XML.

```
/* Declare an XML variable with a literal XML value. */
var xmlEmployees =
    <employees>
        <employee>
            <firstname>John</firstname>
            <lastname>Walton</lastname>
            <age>25</age>
        </employee>
        <employee>
            <firstname>Sue</firstname>
            <lastname>Day</lastname>
            <age>32</age>
        </employee>
        <employee>
            <firstname>Gladys</firstname>
            <lastname>Cravits</lastname>
            <age>53</age>
        </employee>
<employees>;
/* Collect the first and third <employee> nodes into a new XML snippet. */


var xmlSelectedEmployees = xmlEmployees..employee[0] + xmlEmployees..employee[2];


/* Add a new <employee> element to the list. */


var xmlCompleteList = xmlSelectedEmployees.employee + <employee><firstname>Bruce</firstname><la
```

You might also find the appendChild function useful when combining XML elements. For more information, see appendChild function.

Related Topics

appendChild function

Removing Elements and Attributes With the delete Operator

You can use the delete operator to remove specified elements and attributes, as shown in the following example.

```
/* Declare an XML variable with a literal XML value. */
var xmlEmployees = <employees>
    <employee id="111111111">
        <firstname>John</firstname>
        <lastname>Walton</lastname>
        <age>25</age>
    </employee>
    <employee id="222222222">
        <firstname>Sue</firstname>
        <lastname>Day</lastname>
        <age>32</age>
    </employee>
</employees>;
/*
 * Remove the information about John.
 */


delete xmlEmployees.employees.employee[0];



/*
 * Remove the entire <employees> node, leaving an empty XML variable.
 */


delete xmlEmployees.employees;
```

Note that when using delete with a path that ends with a filter predicate, you must append the thisXML property, as in the following example:

```
/*
 * A delete operation with a predicate expression must end with the thisXML property.


 */
delete xmlEmployees.employees.employee.(firstname == "John").thisXML;
```

Related Topics

Filtering Multiple Children With Predicates

Specifying the Current XML with the thisXML Keyword

You can use the thisXML property to specify the current XML, such as XML returned from an expression. The thisXML property works in a manner similar to the this keyword in ECMAScript, which is used to specify objects or functions from within their own code.

The thisXML property is especially useful when filtering the contents of an XMLList. In the following example, thisXML is used to ensure that there aren't any <employee> elements that may be lacking data. Using thisXML ensures that the filter expression adov−−> children().length < 4 adov−−> is evaluated against the XMLList resulting from expression that precedes thisXML. In other words, in this example thisXML is equivalent to the XML returned by xmlEmployees..employee.

```
/* Declare an XML variable with a literal XML value. */
var xmlEmployees = <employees>
    <employee id="111111111">
        <firstname>John</firstname>
        <lastname>Walton</lastname>
        <age>25</age>
    </employee>
    <employee id="222222222">
        <firstname>Sue</firstname>
        <lastname>Day</lastname>
            <age>32</age>
            <homeoffice>Alburquerque</homeoffice>
        </employee>
</employees>;
/*
 * Find the <employee> elements that have less than 4 child elements.
 * Return the id attributes for those elements.
 * This code returns "111111111".
 */
var ageXML = xmlEmployees..employee.(thisXML.children().length < 4).@id;
```

Note: You may also find yourself using thisXML when filtering by attribute. The nature of attributes and the .@ operator makes it necessary to use thisXML in cases like the following:

```
/* Declare an XML variable with a literal XML value. */
var xmlEmployees = <employees>
    <employee id="111111111">
        <firstname>John</firstname>
        <lastname>Walton</lastname>
        <age>25</age>
    </employee>
    <employee id="222222222">
        <firstname>Sue</firstname>
        <lastname>Day</lastname>
```

```
            <age>32</age>
            <homeoffice>Alburquerque</homeoffice>
        </employee>
</employees>;
/*
 * Find the <employee> element where the id attribute is 222222222.
 * Return the corresponding last name.
 * This code returns "Day".
 */
var ageXML = xmlEmployees..employee.(thisXML.@id == "222222222").lastname;
```

Related Topics

Accessing Attributes and Their Values With the .@ Operator

Creating and Using XML Variables

Importing Java Classes to ECMAScript with the import Statement

You can use the import statement to specify Java classes that you may use in ECMAScript. The import statement works in a manner very similar to the Java import directive, except that you may not use the * operator to specify all the classes within a given package.

Classes you reference with import may be qualified by their package name or by their path within the current project. In the following examples, the first imports the ArrayList class, a popular Java class used to create lists of items for iterative access. The second imports Contact, an internal class defined in the ConsolidateAddress.jws web service file, a sample web service available with WebLogic Workshop.

```
import java.util.ArrayList;
import xmlmap.ConsolidateAddress.Contact;
```

For more information about the Java import directive, see "Importing Packages" in Introduction to Java.

Related Topics

Introduction to Java

Functions for Manipulating XML

When writing ECMAScript to manipulate XML, you have access to several member functions designed to make the task easier. This topic describes the functions available with the extended ECMAScript available with WebLogic Workshop.

There may be cases when an element name conflicts with the name of a function listed in this topic. When that occurs, use syntax like that shown in the following example. In this example, the conflicting element name is preceding with a :: (double−colon).

```
/*
 * Return the first <parent> element.
 * This avoids a conflict with the parent function.
 */
var firstParent = xmlFamilies.kid.::parent[0];
```

ECMAScript Functions for XML

Function	Description

xmlElement.appendChild(newChild)	Inserts a new child node after the existing children of the XML value.
xmlElement.attribute(attributeName)	Returns the value of the specified attribute.
xmlElement.attributes()	Returns a list of attributes for the specified element.
xmlElement.child(childIndex)	Returns the XML at the 0−based ordinal position specified by childIndex.
xmlElement.childIndex()	Returns the 0−based ordinal position of the XML value within its parent.
xmlElement.children()	Returns a list of the element's children.
xmlElement.copy()	Returns a copy of the specified element.
xmlElement.domNode()	Returns a org.w3c.dom.Node representation of xmlElement.
xmlElement.innerXML(newContent)	Replaces the entire contents of the XML value with new content.
xmlElement.length	Returns the length of a list XML elements.
xmlElement.namespaceURI()	Returns a string representing the namespace URI associated with xmlElement.
xmlElement.parent()	Returns the parent of the element.
xmlElement.prependChild(newChild)	Inserts a new child node before the existing children of the XML value.
xmlElement.tagName()	Returns the name of the element tag.
xmlElement.thisXML	Specifies the current XML, such as XML returned from an expression.
xmlElement.text()	Returns a string containing the value of all XML properties of xmlElement that are of type string.
xmlElement.toString()	Returns the element and its content as a string.
xmlElement.toXMLString()	Returns an XML encoded string representation of xmlElement.
xmlElement.xpath(xPathExpression)	Evaluates the XPath expression using the XML value as the context node.

Each of the functions listed in this topic is illustrated with an example based on the following XML variable.

```
/* Declare an XML variable with a literal XML value. */
var xmlEmployees = <employees>
    <employee id="111111111">
        <firstname>John</firstname>
        <lastname>Walton</lastname>
        <age>25</age>
    </employee>
    <employee id="222222222">
        <firstname>Sue</firstname>
        <lastname>Day</lastname>
        <age>32</age>
    </employee>
</employees>;
```

xmlElement.appendChild(newChild)

Inserts a new child element (specified by newChild) after the existing children of xmlElement.

The following example adds a <hobby> element to the end of the element assigned to xmlSue.

```
var xmlSue = xmlEmployees..employee.(firstname == "Sue");
xmlSue.appendChild(<hobby>snorkeling</hobby>);
```

xmlElement.attribute(attributeName)

Returns the value of attributeName.

The following example returns a string containing Sue's ID number.

```
var xmlEmployeeID = xmlEmployees.employees.employee.(firstname == "Sue").attribute("id");
```

Note that the attribute() function is an alternative to the .@ operator. Because you may pass it a variable rather than a literal value, the attribute() function is useful when the attribute in question is unknown until run time.

For more information about the .@ operator, see Accessing Attributes with the .@ Operator.

xmlElement.attributes()

Returns a list of the attributes for xmlElement.

The following example returns the attributes for the <employee> element where the <name> value is "Sue"; its return value is "id".

```
var suesAttributes = xmlEmployees..employee.(firstname == "Sue").attributes();
```

xmlElement.child(childIndex)

Returns the XML at the 0−based ordinal position specified by childIndex.

The second line of code in the following example returns an XML type containing the information about Sue.

```
var secondEmployeeIndex = xmlEmployees..employee.(firstname == "Sue").childIndex();
var secondEmployee = xmlEmployees.employees.child(secondEmployeeIndex);
```

xmlElement.childIndex()

Returns the ordinal position (within its parent) of xmlElement.

The following example returns the index of the <employee> element where the <name> element value is "John". Returns "0".

```
var johnIndex = xmlEmployees..employee.(firstname == "John").childIndex();
```

Note that when the expression to the left of childIndex returns multiple items, childIndex returns −1, as in the following example:

```
var nameIndex = xmlEmployees..employee.name.childIndex();
```

xmlElement.children()

Returns an XMLList containing all of the children of xmlElement.

The following example returns the child elements of the first employee element in xmlEmployees. It returns "John, Walton, 25".

```
var xmlEmps = xmlEmployees..employee[0].children();
```

xmlElement.copy()

Returns a deep copy of xmlElement.

The following example adds data about Gladys, a new employee, to a list that already includes John and Sue.

Result: The XML held by newEmployee is added to the list of employees.

```
/* Create a new variable from data from the first employee in the list. */
var newEmployee = xmlEmployees..employee[0].copy();
/* Assigned new values to the copy. */
newEmployee.@id = "555555555";
newEmployee.firstname = "Gladys";
newEmployee.lastname = "Cravits";
newEmployee.age = "43";
/* Append the copy to the list of employees. */
xmlEmployees..employees += newEmployee;
```

xmlElement.domNode()

Returns a org.w3c.dom.Node representation of xmlElement.

A node is the primary data type for the W3C Document Object Model (DOM). The DOM defines a logical structure through which to access XML (and HTML) documents. In the DOM, a node represents a single node in the document tree. Xerces is a Java implementation of the DOM available through the Apache Project.

While the ECMAScript extensions offered with WebLogic Workshop are an alternative to accessing XML via the DOM, the domNode function is provided as a convenient way to create a DOM node. This might be useful, for example, if the script should return a DOM node.

For more information about the Node interface, see the Node Interface topic of the Apache Xalan documentation.

xmlElement.innerXML(newContent)

Replaces the entire contents of xmlElement with newContent.

Note that this is not the same as reassigning the variable. In the example below, the root element represented by the variable (<employee>) remains intact, but its contents are replaced.

The following example replaces the entire contents of xmlSue with the specified elements. This effectively replaces Sue with Gladys, while keeping Sue's ID number.

```
var xmlSue = xmlEmployees..employee.(firstname == "Sue");
xmlSue.innerXML(<><firstname>Gladys</firstname><lastname>Cravits</lastname><age>43</age></>);
```

xmlList.length

Returns the number of items in the list represented by xmlList.

For more on the XMLList data type, see Creating and Using XML Variables.

The following example returns the number of <employee> elements in xmlEmployees. Returns the number 2.

```
var numberOfEmployees = xmlEmployees.length;
```

xmlElement.namespaceURI()

Returns a string representing the namespace URI associated with xmlElement.

The following example returns the namespace URI associated with a SOAP message. Returns a string containing "http://schemas.xmlsoap.org/soap/envelope/”.

```
var xmlStockMessage = <Envelope xmlns ="http://schemas.xmlsoap.org/soap/envelope/"
    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <Body>
        <m:GetLastTradePrice xmlns:m="http://mycompany.com/stocks">
            <symbol>DIS</symbol>
        </m:GetLastTradePrice>
    </Body>
</Envelope>;
var ns = xmlStockMessage.Envelope.Body.namespaceURI();
```

xmlElement.parent()

Returns the parent of xmlElement.

The following example find the employees whose age is under 30, then uses the parent function to return the entire element containing that person's data. It returns an XML type the element containing John's information.

```
var ns = xmlEmployees..employee.(age <= "30").age;
var emp = ns.parent();
```

xmlElement.prependChild(newChild)

Inserts newChild before the existing children of xmlElement.

The following example adds a <prefix> element to the beginning of the element assigned to xmlSue. This code adds <prefix>Mr.</prefix> as the first child of the <employee> node containing information about John.

```
var xmlJohn = xmlEmployees..employee.(firstname == "John");


xmlJohn.prependChild(<prefix>Mr.</prefix>);
```

xmlElement.tagName()

Returns the name of the tag for xmlElement.

The following example returns the name of the first child element of the first <employee> element. It returns a string containing "firstname".

```
var childTags = xmlEmployees..employee[0].children();


var firstChildTag = childTags[0].tagName();
```

xmlElement.thisXML

Specifies the current XML, such as XML returned from an expression.

For more information, see Specifying the Current XML with the thisXML Property.

xmlElement.text()

Returns a string containing the value of all XML properties of xmlElement that are of type string.

The following example finds the second child of the second <employee> element and returns its content. It returns a string containing "Sue".

```
var ageXML = xmlEmployees..employee[1][1].text();
```

xmlElement.toString()

Returns the XML and its contents as a string.

The following example returns a string representation of the first <employee> element.

```
var johnText = xmlEmployees..employee[0].toString();
```

xmlElement.toXMLString()

Returns an XML encoded string representation of xmlElement.

The following example returns a string representation of the first <employee> element.

```
var johnText = xmlEmployees..employee[0].toXMLString();
```

xmlElement.xpath(xPathExpression)

Evaluates the XPath expression (specified by xpathExpression) using xmlElement as the context node.

The following example uses an XPath expression to return the <employee> element where the <firstname> element value is "Sue". It returns an XMLList containing the element.

Result: Returns the <employee> element containing information about Sue.

```
var xmlSue = xmlEmployees.xpath("//employee[firstname='Sue']")
```

Related Topics

Handling XML with ECMAScript Extensions

Summary of ECMAScript Language Extensions

Summary of ECMAScript Language Extensions

This topic lists the operators, functions and keywords you can use within ECMAScript files in WebLogic Workshop projects.

Operators and Keywords

Operator	Description
.@	Provides access to attributes belonging to the element on the left side of the operator. See Accessing Attributes With the .@ Operator.
[]	Provides access to an element child using an index corresponding to its position in document order. See Accessing Element Children Through Their Index.
::	Used with a namespace variable, :: can be used to access elements within a specified namespace. See Filtering By Namespace.
{}	Provides a way to specify an ECMAScript expression with which to substitute for values in XML. See Resolving XML Dynamically with Embedded Expressions.
.	Provides access to immediate child elements contained by the element on left side of the operator. See Accessing Element Children With the . Operator.
..	Provides access to any child element contained by the element on the left of the operator. See Accessing Element Descendants With the .. Operator.
<	Specifies that what follows should be interpreted as XML when assigned to an XML variable. See Creating and Using XML Variables.
.()	Filters a list of element children using a specified value. See Filtering Multiple Children With Predicates.
+	Combines XML to create a new XMLList or to add new values to existing XML. See Combining XML With the + Operator.
+=	Inserts an XML element after another element. See Inserting Elements With the += Operator.
delete	Removes elements and attributes from XML. See Removing Elements and Attributes With the delete Operator.
import	Imports Java classes for used in ECMAScript. See Importing Java Classes to ECMAScript with the import Statement.
namespace	Declares a namespace variable, which you can use to access namespaced elements. See Filtering By Namespace.
thisXML	Specifies the current XML, such as XML returned from a nested expression. See Specifying the Current XML with the thisXML Keyword.

Member Functions

Note For more complete descriptions and examples, see Functions for Manipulating XML.

Function	Description
xmlElement.appendChild(newChild)	Inserts a new child node after the existing children of the XML value.
xmlElement.attribute(attributeName)	Returns the value of the specified attribute.
xmlElement.attributes()	Returns a list of attributes for the specified element.
xmlElement.child(childIndex)	Returns the XML at the 0−based ordinal position specified by childIndex.
xmlElement.childIndex()	Returns the 0−based ordinal position of the XML value within its parent.
xmlElement.children()	Returns a list of the element's children.

xmlElement.copy()	Returns a copy of the specified element.
xmlElement.domNode()	Returns a org.w3c.dom.Node representation of xmlElement.
xmlElement.innerXML(newContent)	Replaces the entire contents of the XML value with new content.
xmlElement.length	Returns the length of a list XML elements.
xmlElement.namespaceURI()	Returns a string representing the namespace URI associated with xmlElement.
xmlElement.parent()	Returns the parent of the element.
xmlElement.prependChild(newChild)	Inserts a new child node before the existing children of the XML value.
xmlElement.tagName()	Returns the name of the element tag.
xmlElement.thisXML	Specifies the current XML, such as XML returned from an expression.
xmlElement.text()	Returns a string containing the value of all XML properties of xmlElement that are of type string.
xmlElement.toString()	Returns the element and its content as a string.
xmlElement.toXMLString()	Returns an XML encoded string representation of xmlElement.
xmlElement.xpath(xPathExpression)	Evaluates the XPath expression using the XML value as the context node.

Related Topics

Handling XML with ECMAScript Extensions

Why Use XML Maps?

A Few Things to Remember About XML Maps and Script

As you work with XML maps, you might find it useful to keep in mind these notes about special behavior in XML maps and ECMAScript for mapping.

When Using Script

Reserved Words Require Special Handling

XML allows a great deal of flexibility when naming elements, attributes, and the like. As a result there may be times when a name in XML you are handling with script is the same as an ECMAScript keyword or one of the functions provided for use with XML. When elements have names that conflict with ECMAScript keywords or function names, you can use the following syntax instead. Note that there are two different workarounds adov−−> one is for use with ECMAScript keywords and the other is for use with function names. The following example illustrates the workaround for the . operator; be sure to see the documentation for the specific operator you are trying to use.

```
/*
 * Return all of the <if> elements.
 * This avoids a conflict with the "if" ECMAScript keyword.
 */
var ifs = xmlData.product_description["if"];
/*
 * Return the first <parent> element.
 * This avoids a conflict with the parent function.
 */
var firstParent = xmlFamilies.kid.::parent[0];
```

XML Fragments Must Have an Anonymous Root

The XML−related data types included with Workshop's extended ECMAScript allow you to assign XML fragments to XML variables. However, a fragment must be enclosed in an anonymous element represented by <> and </>, as in the following example:

```
var myXMLList =
<>
    <firstname>John</firstname>
    <lastname>Walton</lastname>
    <age>25</age>
<>
```

When Using Maps

Function References in XML Maps Must Be a Root Element

The function reference (everything between and including the curly braces) must be the only child of its parent element. For example, the following XML map fragment will cause an error:

```
<placeOrder xmlns="http://www.openuri.org/">
    {CustomerServices.OrderScripts.convertOrder(currentOrder, currentCustomer)}
    <customer_info>{currentCustomer}</customer_info>
</placeOrder>
```

But something like the following would work:

```
<placeOrder xmlns="http://www.openuri.org/">
    <order_info>
        {CustomerServices.OrderScripts.convertOrder(currentOrder, currentCustomer)}
    </order_info>
    <customer_id>{currentCustomer.custID}</customer_id>
</placeOrder>
```

For more information, see How Do I: Use Script from a Web Service?

Literal Text Can't Be Combined with Substitutions

Literal text — that is, text that is not part of an XML element or attribute name and is not part of mapping tags and attributes — is allowed in XML maps only under certain circumstances. For example, literal text may not be combined with substitution directives (text in xm tags or between curly braces), as in the following example:

```
<!-- not a legal xml map -->
<item>
    <description>  bad text  {d}  bad text
    </description>
    bad text
    <quantity units="bad text {u} bad text">
        bad text  {q}  bad text
    </quantity>
    bad text
</item>
```

The following example, however, is legal:

```
/**
```

```
 * @jws:parameter-xml xml-map="xml"::
 * <searchData>
 * <query>
 *     <criterion>
 *         <vendorName>Legal Text</vendorName>
 *         <partName><xm:value obj="criterionValue"/></partName>
 *     </criterion>
 * </query>
 * </searchData>
 * ::
 */
public String searchData(String criterionName, String criterionValue)
{...}
```

XML Maps Change a Service's Interface

When you apply an XML map to a method or a callback, that service's interface (as expressed in its WSDL, for example) changes. The interface reflects the data as it is presented through the map. The map in effect "hides" the underlying Java code.

For example, imagine you have a callback that sends an inner class as one of its parameters. If you have an XML map on the callback, perhaps parsing the inner class's data members to XML elements, the inner class itself may not be visible to clients. When you generate a service control from your service's WSDL, the control will declare a kind of anonymous substitute for inner class prepended with "AnonType_". A client web service using the control must, in turn, import this class, giving the control name as a package name. Note that adding the import statement is handled by WebLogic Workshop when you add the control to a client web service.

Map Roots Must Be Unique Within a JWS File

Just as with classes inside of JWS files, the root tags of XML maps must be unique within a JWS file. This is because they result at compile time in distinct classes for the types that support them. In the following two XML maps, for example, the <searchRequest> and <SearchRequest> tags occur as root tags. The result will be that the CLASS files needed to support one will overwrite those needed to support the other.

```
/**
 * This code will not work as expected! The <searchRequest> tags create a situation
 * in which one map will work while types to support the other are overwritten.
 *
 * @jws:operation
 * @jws:parameter-xml xml-map::
 *     <searchRequest  xmlns="http://www.openuri.org/">
 *         <productName serialNumber="{serialNumber}">{productName}</productName>
 *         <quantity>{quantity}</quantity>
 *     </searchRequest>
 *      * ::
 * @jws:return-xml xml-map::
 *     <SearchRequest  xmlns="http://www.openuri.org/">
 *         <return>{return}</return>
 *     </SearchRequest>
 *      * ::
 */
```

Note: If you are working on a computer running Windows, the uniqueness must take case into account. Because Windows does not have a case−sensitive file system, a class called searchRequest.class will overwrite a class called SearchRequest.class.

In other words, it's a good practice to list the names of all classes and XML map root tags within a JWS and make sure they're unique with respect to each other.

Related Topics

How Do XML Maps Work?

Debugging Web Services

You can use the WebLogic Workshop integrated debugger to debug your web service. The debugger allows you to set breakpoints, step through your code line−by−line, view local variables, set watches on variables, and view the call stack and exception information.

Testing with Debugging

To use the debugger while testing your web service, click the Start and Debug button on the toolbar, or press F5. To stop debugging, click the Stop button on the toolbar.

When the debugger is running, you'll see a command window, titled WebLogic Workshop Debugger, for the debugger proxy. This window must remain open in order to use the debugger. If you close this window, your breakpoints will not be hit when you test your service.

While a web service is running in the debugger, it is unavailable to clients.

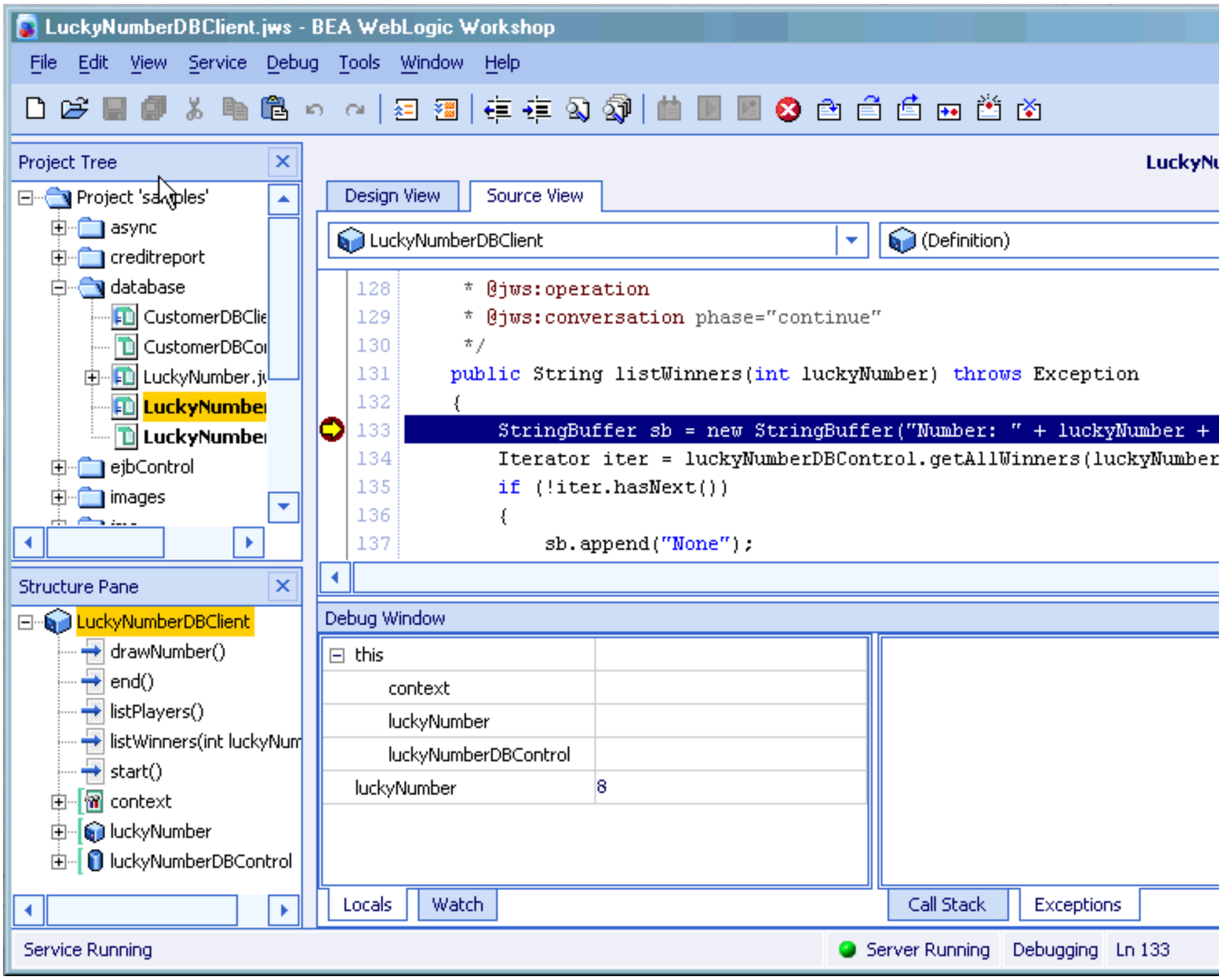
Toggling Breakpoints

You can set a breakpoint in your code to halt execution on that line, immediately before it is executed. You can then examine variable values or the call stack or set other break points.

To set or remove a breakpoint, go to Source View and put the cursor on the line on which you want to break. Click the Toggle Breakpoint button on the toolbar. You can also toggle a breakpoint by pressing F9; by clicking in the "trough" to the left of the line numbers on the left edge of the Source View window; or by right−clicking on the line of source code and selecting Set Breakpoint from the context menu that appears.

To clear all breakpoints in the project at once, click the Clear All Breakpoints button on the toolbar, or press Ctrl−Shift−F9.

The following image shows execution halted on a breakpoint in Source View:



```
     * @jws:operation
     * @jws:conversation phase="continue"
     */
    public String listWinners(int luckyNumber) throws Exception
    {
        StringBuffer sb = new StringBuffer("Number: " + luckyNumber +
        Iterator iter = luckyNumberDBControl.getAllWinners(luckyNumber
        if (!iter.hasNext())
        {
            sb.append("None");
```

⊟ this	
context	
luckyNumber	
luckyNumberDBControl	
luckyNumber	8

If you set a breakpoint on a line that is not executed, such as a variable declaration, WebLogic Workshop will display a warning in the Output window that the breakpoint is invalid.

Note that if you add breakpoints to a file, the breakpoints will be lost if you close the file in the visual development environment.

Using the Debugging Commands

Once you've halted execution on a breakpoint, you can use one of the following commands to continue executing your code with the debugger. All of these commands are available on the toolbar and on the Debug menu.

- Step Into: The Step Into command continues execution line−by−line, beginning with the next line of code. Use this command if you want to debug your code one line at a time.
- Step Over: The Step Over command executes a method call without debugging that method, and halts execution on the next line. Use this command if you know that the code in a method works and you don't need to step into it.
- Step Out: The Step Out command finishes executing a method and returns execution to the procedure that called it, halting on the line immediately following the method call. Use this command if you have stepped into a method and you don't want to continue stepping all the way through it.

- Continue: The Continue command resumes execution until another breakpoint is encountered or the procedure has completed.

Note that if you step into a line that contains more than one statement, all of the statements on that line will be executed when you step to the next line.

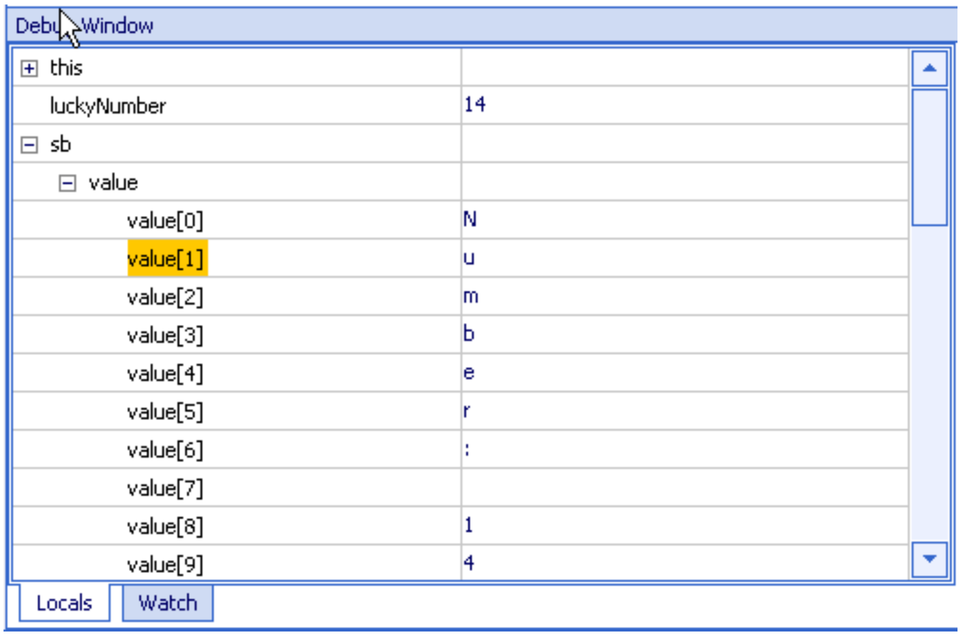
Using the Debug Window

The Debug window provides information about values in your code while you're running it in the debugger.

Viewing Local Variables

The Locals window automatically shows variables that are in scope as you execute your code in the debugger. You can modify the value of a primitive−type variable here while you are debugging.

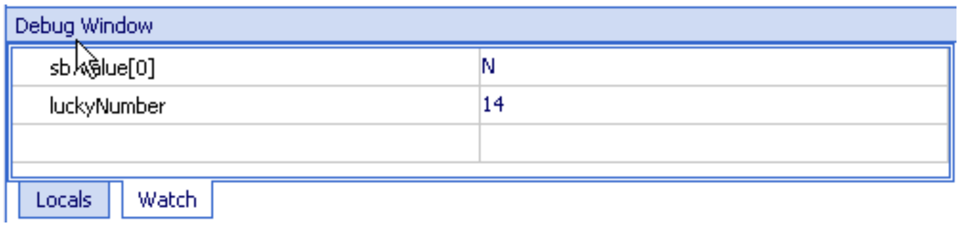
You can expand entries for objects in the Locals window to view their members. The following image shows the individual array elements that make up the value of a StringBuffer object.



this	
luckyNumber	14
sb	
value	
value[0]	N
value[1]	u
value[2]	m
value[3]	b
value[4]	e
value[5]	r
value[6]	:
value[7]	
value[8]	1
value[9]	4

Setting a Watch

You can use the Watch window to observe how the value of a particular variable changes as you're debugging. To set a watch, simply type the name of the variable into the Watch window. The following image shows variables in the Watch window.



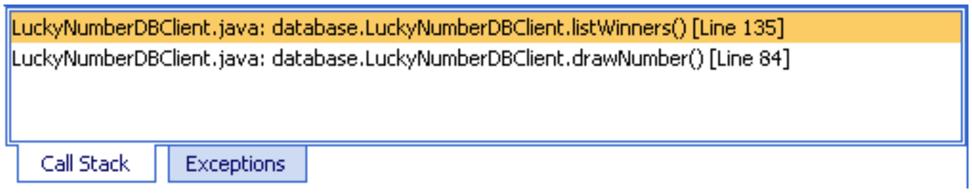
sb.value[0]	N
luckyNumber	14

Viewing the Call Stack

The call stack shows the methods that have been called in order to reach the point at which execution is halted in the debugger. The methods are listed from most recently called to first called. You can also think in terms of each call being nested within the method below it on the call stack.

You can double−click on a method in the call stack to navigate to that method in Source View.

The following image shows two methods on the call stack.



```
LuckyNumberDBClient.java: database.LuckyNumberDBClient.listWinners() [Line 135]
LuckyNumberDBClient.java: database.LuckyNumberDBClient.drawNumber() [Line 84]
```

Remote Debugging

If you're developing against a remote server, you can also debug against that remote server. In the Preferences dialog, on the Paths tab, set the Name option to the name of the remote server and set the Port option to its specified port. Set the Config directory option to point to the directory that contains the server domain, and set the Domain option to the name of that domain.

Note: The remote server must have a valid machine name in order for you to debug against it. On a Linux system, you can use the hostname command to determine the machine name.

For more information on modifying your WebLogic Workshop preferences to point to a remote server, see How Do I: Create a New WebLogic Workshop−enabled WebLogic Server Domain?

Documenting Web Services

The web services you build may include documentation for the web service. The documentation you include is displayed in WebLogic Workshop's Test View during web service testing.

Documentation Content

The sections below describe the format and use of Javadoc comments in web services. The comments you include in your web services are included in the WSDL that you generate for each web service if you want to publish your web service for use by others. The documentation you provide will also be displayed on WebLogic Workshop's Service View, which is a production mode view that presents only the Overview Page you see in Test View.

When you write comments for web services, remember that a core concept of web services is platform− and language−independence. In WebLogic Workshop, you write web services in Java. But clients of you web services do not (and should not) need to know that. The descriptions you provide should explain the semantics of your web service and its methods and, at a high level, the XML and/or SOAP message formats your web service expects (the WSDL file specifies the details of message formats). Your descriptions should not include implementation details of your web service or mention Java data structures, classes or language features.

Javadoc Comments

The documentation for your web service and its methods is optionally placed in Javadoc comments on the web service's main class and on each of its methods. These areas are described in more detail below.

Javadoc comments always begin with /** (not /*) and end with */. If you create a comment that begins with anything other than /**, it will not be considered a Javadoc comment and the information in this section will not apply.

HTML Formatting in Comments

Javadoc comments may include HTML formatting tags. Some tools, including WebLogic Workshop's Test View, will honor the HTML formatting when presenting the documentation. This means you may include bolding (<b> tag), italics (<i> tag), lists (<ul> and <ol> tags with member <li> tags), etc.

Javadoc Tags and Comment Structure

If Javadoc tags (tokens beginning with @) are present in a comment, they must be located at the end of the comment. Any text following a Javadoc tag (and preceding another tag) is considered arguments to the tag. If you include additional commentary after the Javadoc tags in a Javadoc comment, the WebLogic Workshop compiler will generate errors about inappropriate arguments to tags.

For example, the following Javadoc comments will compile successfully:

```
/**
 * Credit reporting service.  This service simulates constructing
 * a credit report from multiple secondary sources of information.
 * It uses two external services, one representing a bank and the
 * other representing the Internal Revenue Service (IRS).


 *
 * The @jws:conversation−lifetime max−idle−time tag controls how
 * long a conversational instance of this service will survive without
 * seeing activity.


 *
 * Conversations represent resources: they shouldn't be left around.


 *
 * @jws:conversation−lifetime max−idle−time="30 minutes"
 */
public class CreditReport
{
    ...
}
```

The following comment will not compile successfully because of the spurious comments following the @jws:conversation−lifetime tag.

```
/**
 * Credit reporting service.  This service simulates constructing
 * a credit report from multiple secondary sources of information.
 * It uses two external services, one representing a bank and the
 * other representing the Internal Revenue Service (IRS).
 *
 * The @jws:conversation-lifetime max-idle-time tag controls how
 * long a conversational instance of this service will survive without
 * seeing activity.
 *
 * Conversations represent resources: they shouldn't be left around.
 *
 * @jws:conversation-lifetime max-idle-time="30 minutes"
 *
 * Here are some more comments that will cause compile problems.
 */
public class CreditReport
{
    ...
}
```

These examples also illustrate the issue of mentioning Javadoc tags within comments. Notice that the comment discusses the @jws:conversation−lifetime tag. If the tag were located at the start of the line, it would be interpreted as a tag (not as a comment) and would generate compile errors because of the text following it. The presence of text before the comment on the line (the word "The") prevents the tag form being interpreted as an active tag.

Class Comment

A Javadoc comment that immediately precedes a class is called the class comment. The class comment is currently not displayed in Test View, but it is included in WSDL files generated from a JWS file.

Method Comment

A Javadoc comment that immediately precedes a class is called the method comment. Test View will display the method comment for each method on both the Overview Page and preceding the method parameters and invocation on the Test Form and Test XML pages.

Method comments are also included in WSDL files generated from a JWS file.

Related Topics

Introduction to Java

Structure of a JWS File

JWS Files: Java Web Services

WSDL Files: Web Service Descriptions

Security

WebLogic Workshop’s security model allows you:

1. To use WebLogic Server’s declarative security model, an implementation of the J2EE servlet security model.

2. To use WebLogic Server’s programmatic security model

The J2EE servlet security model lets you restrict access to any web resource, whether that resource is a web application, a web service within that application, or a particular method in that web service. On the J2EE model, users are assigned to groups, and different groups are granted access to different web resources. For more information on using groups and roles, see Declarative Security.

The servlet security model also lets you expose the web services within a given application on HTTP− or HTTPS−enabled ports as necessary. Https−enabled ports should be used whenever web service communication includes sensitive data that needs to be encrypted. For information on configuring your web service to be exposed on an HTTPS−enabled port see Using HTTPS to Secure a Workshop Web Service.

The programmatic security model lets you make security decisions in the web service code itself to change web service behavior based on the permissions of the user. For more information, see Programmatic Security.

WebLogic Server's 7.0 Security Model

WebLogic Workshop supports two modes of security infrastructure: the new WebLogic Server 7.0 security model and the WebLogic Server 6.1 realm−based model.

If your Workshop applications do not include WebLogic Portal or WebLogic Integration components running in the same server domain, use the new WebLogic Server 7.0 security model. An empty WebLogic Server domain template is provided to support configuration of a domain for such usage.

If your Workshop applications include WebLogic Portal or WebLogic Integration components running in the same domain, you must use the WebLogic Server 6.1 security model. Because security is configured on a domain−basis, this is true even if the components are running in different server instances. For information about how to run an application on a WebLogic Server 7.0 server while using the 6.1 security model, see Compatibility Security in the WebLogic Server 7.0 documentation

Using HTTPS to Secure a Workshop Web Service

The following topic explains how expose a Workshop web service using HTTPS.

WebLogic Server Security

Web services built with Workshop are deployed to WebLogic Server in the form of web applications. Different services within a single application can be exposed using different transport protocols, secure or insecure, as necessary. Secure an individual service by specifying that it be exposed using HTTPS rather than HTTP. HTTPS encrypts the communication between the client and web service, and it also offers a degree of authentication of the server using a digital certificate, which the server presents to the client. The one−way authentication offered by the server to the client, can be supplemented by two−way authentication, where the client offers a digital certificate to the server, in addition to the certificate offered by the server.

Using HTTPS

You specify the web service exposure protocol as you prepare to deploy the web service to WebLogic Server. To expose a web service using the HTTPS protocol rather than the HTTP protocol, you must:

1. Make sure that the WebLogic Server where you deploy the web service is listening on an HTTPS−enabled port. See Configuring the SSL Protocol in the WebLogic Server 7.0 documentation.
2. Make sure that the web service's WSDL file advertises that HTTPS−enabled port.

To ensure that your web service's WSDL directs clients to an HTTPS−enabled port, edit the weblogic−jws−config.xml file, found in the project's WEB−INF directory. In the weblogic−jws−config.xml file you can set a dedicated HTTP port, a dedicated https port, and set which ports individual web services should use.

The weblogic−jws−config.xml file can include multiple <jws> elements. Inside each <jws> element you specify a web service using the <class−name> element, and the exposer protocol, either HTTP or HTTPS, using the <port> element. Specifying that a web service should use the HTTPS−enabled port causes the web service WSDL to advertise the application's HTTPS−enabled port.

The example weblogic−jws−config.xml file below shows how to expose the web service MySecureService on the HTTPS port (specified as port 7002). You should recompile the application EAR and redeploy the application each time you change the weblogic−jws−config.xml file, otherwise changes will have no effect on the deployed application.

Note: the settings on the weblogic−jws−config.xml file can be overridden at compile−time by the parameters you specify in the jwsCompile command. For more information see Deploying Web Services and JwsCompile Command.

```
<config>
 <protocol>http</protocol>
 <hostname>localhost</hostname>
 <http-port>7001</http-port>
 <https-port>7002</https-port>
 <jws>
  <class-name>MySecureService</class-name>
  <protocol>https</protocol>
 </jws>
 ...
</config>
```

In order to expose different services on differently−enabled ports, add a <jws> element with child <class−name> and <protocol> elements. The example weblogic−jws−config.xml file below specifies that the HelloWorld service should use the HTTP−enabled port and the service HelloWorldSecure should use the HTTPS−enabled port.

Note: in the example below, the default <protocol> tag has the value http: this makes it, strictly speaking, redundant to use the jws−specific <protocol> element to specify that HelloWorld should use the http port. HelloWorld's jws−specific <protocol> element is present for the sake of clarity.

```
<config>


    <!--  The global <protocol> element says that any service in the project file should
     be exposed on http, unless otherwise specified.


    -->


 <protocol>http</protocol>
 <hostname>localhost</hostname>
 <http-port>7001</http-port>
 <https-port>7002</https-port>


     <!--It is, strictly speaking, superfluous to use the jws-specific <protocol tag to
     declare that the HelloWorld service should be exposed on the http protocol this was
     accomplished by the global <protocol> element.-->


 <jws>



     <!-- Recall that JWS files are really Java classes, hence, the element name 'class-name'.


     -->


  <class-name>HelloWorld</class-name>
  <protocol>http</protocol>
</jws>
<jws>
  <class-name>HelloWorldSecure</class-name>
  <protocol>https</protocol>
 </jws>


...
</config>
```

Declarative Security

WebLogic Server's implementation of the J2EE servlet security model lets you authenticate users and authorize them to access different URL's within the servlet. Since the servlet controls the security gate, web resources never encounter incoming client requests to the web resource that do not pass the servlet's security gate.

Groups and Roles

Authorization involves granting permissions to a user to access some web resource such as a web application, a web service within the application, or an individual operation within the service. The servlet restricts access by allowing only some users access to the URL where the web resource resides.

Users are assigned roles, and roles are associated with permissions to access web resource URLs. Users and groups of users are assigned roles in the weblogic.xml file. Roles are associated with permissions to access URLs in the web.xml configuation file.

Configuring Web Services with web.xml and weblogic.xml

You declare a web service as a protected web resource in the web.xml file. The web.xml file uses the <url−pattern> element to restrict access to regions of the URL space in the servlet. The following example restricts access to the members role by allowing only users who are members access to the URL /MyWebService.jws and to any URLs in the path of the /MembersFiles directory.

Note: since individual methods are invoked via URLs (http://productionMachine:7001/myApp/myService.jws/myMethod)), you can restrict access on a method−by−method basis by using the appropriate URL pattern.

```
<security-role>
      <description>Members have access to the web service</description>
      <role-name>members</role-name>
    </security-role>
    <security-constraint>
        <display-name>Security Constraints</display-name>
        <web-resource-collection>
            <web-resource-name>MyWebService</web-resource-name>
            <url-pattern>/MyWebService.jws</url-pattern>
            <url-pattern>/MemberFiles/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>members</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
```

Users are granted certain roles in the weblogic.xml configuration file. Individual users and groups can appear in the <principal−name> element. The example below grants two individuals the role of member.

```
    <security-role-assignment>
        <role-name>members</role-name>
        <principal-name>john</principal-name>
        <principal-name>jane</principal-name>
    </security-role-assignment>
```

Related Topics

Overview: Security

Programmatic Security

Workshop supports WebLogic's security API's to allow you to change web service behavior based on a user's group, and other security parameters. The API's appear in the web service code itself and are executed after a user has passed the servlet's security gates. The servlet controlled security model is described in Declarative Security.

Authorizing Clients within Web Service Code

Once a client has passed the servlet security gates, you can access the clients group and security permissions with the web service code in order to authorize individual methods and operations, or to otherwise change web service behavior based on the identity of client. Client discovery is accomplished by methods in the JwsContext interface.

```
public interface JwsContext
        {
        public java.security.Principle getCallerPrinciple();
        public Boolean isCallerInRole( String roleName );
        }
```

You can use getCallerPrinciple() to discover the principle that identifies the caller. Use isCallerInRole() to determine whether a given caller is in a specified role.

Sending Credentials with Outgoing Messages

When you need to send authentication credentials with outgoing method invocations on external web services, you can attach credentials in one of three ways.

1. If your outgoing method invocation is accomplished through a service control, then use the following methods on the ServiceControl interface:

public interface ServiceControl { public void setUsername(String username); public void setPassword(String password); public String getUsername(); public String getPassword(); }

Note: To call a secure service through a service control, you must modify the service CTRL file so that the http−url property of its @jws:location annotation specifies a fully qualified HTTPS url. For example, if you are trying to call the HelloWorldSecure service via a service control, the CTRL file must be modified as below:
@jws:location http−url="https://localhost:7002/samples/HelloWorldSecure.jws"

2. If you outgoing method invocation is accomplished through a callback interface, that is, if your service issues a callback to an external service, then use the following methods on the JwsContext object:

public interface JwsContext { public void setCallbackUsername(String username); public void setCallbackPassword(String password); public String getCallbackUsername(); public String getCallbackPassword(); }

3. You can also specify credentials for outgoing messages by encoding them in the endpoint URL for both service controls and callback interfaces. The format of an URL with credentials is:

<protocol>://<username>:<password>@<host>:<port>/...

Users may call ServiceControl.setEndPoint() and JwsContext.setCallbackURL() with such an URL. When an outgoing method invoking message is packaged, the current username and password will be encoded for the current protocol and sent with the invoking message.

Related Topics

Overview: Security

Protocols and Message Formats

When you're building your web service, you can specify how it sends and receives messages and how those messages are formatted. WebLogic Workshop supports two protocols, HTTP and JMS. Either protocol can exchange messages in a variety of message formats, including SOAP and raw XML. The topics in this section explore the protocols and message formats available to you.

Topics Included in This Section

Supported Protocols

Discusses the HTTP and JMS protocols and the advantages of each.

Supported Message Formats

Explores the message formats available for each protocol.

Building a JMS Client

Describes the required settings for accessing a web service over JMS.

Supported Protocols

When designing your web service in WebLogic Workshop, you can choose which protocols your web service supports for sending and receiving data. A protocol is a low−level network scheme for passing messages between systems. It provides a set of rules for formatting and encoding data in a standard way.

WebLogic Workshop currently supports two protocols: HTTP, or Hypertext Transport Protocol, the basic protocol used by web browsers and web servers; and JMS, or Java Messaging Service, a component of the J2EE specification. WebLogic Server includes WebLogic JMS, and can also support other third−party messaging systems.

Your web service can support one or both protocols. Each method and callback on your service can also support either or both protocols. To specify which protocol a service or one of its methods or callbacks uses, set the attributes of the protocol property of that service, method, or callback. For more information on setting the protocol property, see Supported Message Formats.

HTTP

The most common protocol supported by web services is HTTP (Hypertext Transport Protocol, the basic protocol used by web browsers and web servers). Although HTTP has been most commonly used to transmit World Wide Web documents up until now, it can be used to transmit almost any document. The fact that it is ubiquitous and flexible and that most firewalls are trained to allow HTTP messages to pass through makes it a useful protocol for web services which need to be available to users in geographically dispersed locations, over the Internet.

HTTP is also a useful protocol for web services that require synchronous, two−way communication. When a client calls a method of your web service over HTTP, that method may immediately return a value to the client.

However, HTTP lacks many features that are crucial to the enterprise, most notably reliable asynchronous messaging. In order for two systems to exchange messages via HTTP, both systems must be available. If one is not, there is no guarantee that the message will be delivered; HTTP provides no means for persisting that message. Also, HTTP does not provide transactional support, so complex operations cannot be reliably executed over HTTP. For enterprise applications that require asynchronous messaging and transactions, consider using JMS as the transport protocol for your web service.

JMS

JMS is a useful protocol for distributed enterprise applications that require asynchronous, loosely−coupled, highly reliable communication. When a message is exchanged via JMS, the system posting the message can be completely independent of the system that retrieves the message. JMS serves as the middleman; the participating systems don't have to communicate in any way, or even be simultaneously available, to exchange information asynchronously. The message is guaranteed to be delivered, because JMS persists the message until it is retrieved by its intended recipient.

JMS also supports transactions, so that complex operations are performed in a controlled, coordinated manner. Transactions guarantee that an operation involving a number of resources will either succeed completely or fail completely, so that the system is left in a stable state either way.

When your service uses JMS as a transport protocol, it employs a JMS queue. JMS queues are one−way, so synchronous return messages are not sent to your service when you send data over JMS. Methods using the JMS protocol must have a void return. Your service should return results to the client either by using a callback or by using a JMS control to put another message on the queue.

Related Topics

Supported Message Formats

Supported Message Formats

Once you chosen a protocol – HTTP or JMS – you can choose which formats to support for the messages that you send and receive over it. The message format describes how the message is prepared to be sent over the protocol. For example, your web service can format data as a SOAP message, the most common format for web services, to send over either HTTP or JMS. Or it can format data in raw XML, to send over either HTTP or JMS.

A web service can support multiple protocols and message formats, as can a method or callback on the web service. To specify which protocols and message formats are supported for a service or for an individual method or callback, select the service, method, or callback in Design view and set the attributes of the protocol property. The attributes of the protocol property are described below:

The various combinations of protocols and message formats available in WebLogic Workshop are described below:

- http−soap: Messages are SOAP−formatted and delivered over HTTP. This is the most common protocol and message format for web services. If your web service supports this protocol, you can also specify whether the soap−style is set to document or RPC. By default http−soap is set to true.
- form−post: To receive data from a form in a web browser, your web service can support form POST over HTTP. Form POST encodes the form data as a set of name−value pairs and transports the data in the body of the HTTP POST request. The HTTP POST request is MIME−encoded; its MIME type is "application/x−url−formencoded". No SOAP headers are sent with form−post. By default form−post is set to true.
- form−get: To receive data from a form in a web browser, your web service can support form GET over HTTP. Form GET encodes the form data as a set of name−value pairs that are transmitted on the URL when the form is submitted. No SOAP headers are sent with form−get. By default form−get is set to true.
- http−xml: Messages are sent as raw XML over HTTP, without SOAP headers. The XML is transmitted over HTTP POST, but in this case the encoding type is set to “text/xml”, which is not form−compatible. By default http−xml is set to false.

- jms−soap: Messages are SOAP−formatted and delivered over a JMS queue. You can specify whether the soap−style is set to document or RPC. By default jms−soap is set to false.
- jms−xml: Messages are sent as raw XML over a JMS queue, without SOAP headers. The MIME type is set to “text/xml”. By default jms−xml is set to false.
- soap−style: SOAP−formatted messages can be formatted according to either of two encoding styles, document or rpc. Document−style encoding is the more flexible style, because the developer can determine the shape of the XML data and how it maps to Java data structures. RPC−style encoding is a stricter style and does not permit the developer to configure XML data. By default, soap−style is set to document.

Notes:

- In order to use Test View to test and debug your web service or an individual method or callback, the service, method, or callback must support HTTP GET; that is, the form−get attribute must be set to true.
- If your web service sends and receives messages through either of the raw XML protocols, http−xml and jms−xml, note that WebLogic Workshop cannot manage conversations for you using these protocols, so callbacks and continue/finish methods will not work automatically.
- If a method using the jms−soap protocol participates in a conversation, any subsequent callbacks that participate in that conversation must be explicitly configured to use the same protocol.
- A service or method that supports the ***jms−soap*** protocol can receive a ***TextMessage*** object whose body contains the SOAP message. The ***TextMessage*** object must have an application−defined property named "URI" that specifies the URI of the target service. For example, to send a message to the service that's at http://myserver:portnumber/myapp/myjws.jws, set the URI property to "/myapp/myjws.jws".

Related Topics

Supported Protocols

Building a JMS Client

To build a client that exchanges messages with a WebLogic Workshop web service over a JMS queue, you need to configure the client so that it can connect to the queue. Import the javax.naming class and create a new InitialContext object with the properties shown in the following table:

Property	String Setting
SECURITY_PRINCIPAL	User or principal name; default is "system"
SECURITY_CREDENTIALS	Principal password; default is "password"
INITIAL_CONTEXT_FACTORY	JMS context factory: "weblogic.jndi.WLInitialContextFactory"
PROVIDER_URL	Host URL, using the t3 protocol (e.g., "t3://localhost:7001")

The queue name is "jws.queue".

The message that you send to the web service must be a TextMessage object containing XML. If the web service is asynchronous, then the XML in the message must be SOAP−formatted and must include the SOAP headers to support conversations. For more information on conversational SOAP headers, see Conversing with Non−Workshop Clients.

The message must also have a String property named URI that points to the web service. For example, if the URL to the web service is "t3://localhost:7001/samples/JMSService.jws", set the URI property to "/samples/JMSService.jws".

For more information on building a JMS client, see Developing a WebLogic JMS Application on e−docs.bea.com.

Related Topics

Conversing with Non−Workshop Clients

How Do I: Tell Developers of Non−WebLogic Workshop Clients How to Participate in Conversations?

Implicit Transactions in WebLogic Workshop

When a method or callback in your web service is executed, the method or callback is encapsulated in a transaction. Other resources called by the method or callback may also be part of the transaction. A transaction coordinates a set of operations so that if one operation fails, none of the operations happen, and the state of the application is returned to the way it was before the method or callback executed.

Topics Included in This Section

Transaction Basics

Provides a basic overview of transactions in enterprise applications.

Default Transactional Behavior in WebLogic Workshop

Explains the default behavior of implicit transactions in WebLogic Workshop.

Transaction Basics

Transactions make it possible to build robust, reliable enterprise applications by controlling the way in which changes are made to the application state. The application state is the state of all of the application's resources at a given moment in time – what data is stored on disk, in memory, in a database. In a sophisticated application, these resources may be in different processes or on different machines, and a single operation may involve updates to multiple resources. If there is a failure at one point, and the other changes are allowed to succeed, the application's integrity may be compromised. Transactions solve this problem by ensuring that no updates occur unless they all succeed. If they do, the transaction is committed – that is, updates are made to all participating resources. If one operation fails, then all changes are rolled back to the initial state, and the failure is logged or returned to the user as an error.

A transaction has the properties of atomicity, consistency, isolation, and durability, collectively referred to as ACID properties, described here:

- Atomicity: A transaction is an indivisible unit of work. All changes that a transaction makes to the state of an application are made as one unit; otherwise, all changes are rolled back.
- Consistency: A successful transaction transforms the state of an application from a previous valid state to a new valid state, without violating any integrity constraints.
- Isolation: Any changes that a transaction makes to an application's state are not visible to other operations until the transaction completes its work.
- Durability: Any changes that a transaction makes to an application's state will survive future system or media failures.

Transaction management is a fundamental feature of WebLogic Server. For the most part, transactions are managed behind the scenes, but it's a good idea to be aware of how transactions will happen in your application. For more information on transaction management, see Managing Transactions on e−docs.bea.com (http://e−docs.bea.com/wls/docs70/adminguide/managetx.html).

Related Topics

Default Transactional Behavior in WebLogic Workshop

Default Transactional Behavior in WebLogic Workshop

Each time a method or callback in your web service is called, WebLogic Workshop begins an implicit transaction that is associated with that method or callback. If the method or callback returns successfully, without throwing an exception, the transaction is committed. If the method or callback fails, the transaction is rolled back, and none of the changes that have happened within the transactional context are committed.

Included within the transactional context are changes to the state of an ongoing conversation, and calls to methods of the Database, Timer, EJB, and JMS controls. If a method fails at any point, operations performed by the method, changes to the conversation state, and operations performed by any of these controls will be rolled back to their original state. For example, if the web service method calls a method on a Database control to create a table, and the web service method fails subsequently on a different call, the database operation will be rolled back.

Any operations performed by a Service control or an Application View control are not included in the transactional context of the web service method or callback. If the web service method fails following a successful call to a method or callback on a Service control, the operation performed by the Service control will not be rolled back.

There is one case in which a Service control is included in the web service's transactional context to a limited extent. If a method on a Service control has a message buffer, then the queue on that method is included in the transactional context of the web service method. That is, if a failure occurs after a call to a buffered method on a Service control, the method will never be called.

If an exception occurs in a web service method that participates in a conversation, the transaction is rolled back as expected, and the effect on the conversation state depends on whether the the exception is handled. If the exception is handled, the conversation state is updated; if not, the conversation state is not updated. Specifically, if an unhandled exception occurs in a method that starts a conversation, the conversation is never started; if an unhandled exception occurs in a method that continues or finishes a conversation, the conversation state is not updated.

To ensure that conversation state is updated whether or not the method succeeds, handle the exception without rethrowing it, as shown in the following example:

```
/**
 * @operation


 * @conversation finish


 */


public void finish()


{
try
    {
        bank.cancelAnalysis()
    }


catch (ProxyException pe)
    {
        // Log the error, but don't rethrow.
    }


}
```

Note: Whether a particular control used by your web service actually participates in the service's transaction depends on how the resource exposed by the control is configured. For example, a database exposed by a Database control must be configured to use a transactional connection pool in order to participate in the web service's transaction. Enterprise JavaBeans can support an existing transaction or be configured to create their own transaction rather than to participate in the web service's transaction. When you're building your service, you should take into account how the resources that you're using are configured to participate in distributed transactions.

For more information, see the following topics on e−docs.bea.com:

Managing Transactions

Programming Weblogic JTA

Related Topics

Transaction Basics

Overview: Deployment and Clustering

Workshop web services are deployed to WebLogic Server in the form of EAR (Enterprise Application archive) files. You can deploy a Workshop EAR either to a single instance of WebLogic Server or to a cluster of WebLogic Servers. The topics in this section explain how to generate an EAR file for a Workshop project and deploy it, either to a single instance of WebLogic Server, or to a cluster of WebLogic Servers.

Topics Included in this Section

Deploying Web Services

Explains how to build an EAR file for a Workshop project and deploy it to a single instance of WebLogic Server.

Overview: Clustering

Explains the basic concepts of clustering WebLogic Servers.

Clustering Workshop Web Services

Explains the basic concepts of deploying Workshop applications to a cluster of WebLogic Servers.

Cluster Deployment

Provides a step−by−step guide to setting up a WebLogic Server cluster and deploying a Workshop application to that cluster.

Production Mode Deployment Overview

WebLogic Server can be run in two modes with respect to web services: design mode and production mode.

The default mode of the server is design mode, which provides support for iterative development. Your web service application source files are stored on disk in exploded directory format. Changes to those source files are dynamically noted by the server and affect the running service. Developers of web services should run the server in design mode.

At some point you may want to deploy a Web Service onto a different machine than the one you developed it on. The most common reason for this is to make the service available on a publicly accessible machine, which is usually not a machine that is used for development. Such a machine should run the server in production mode, which provides support for applications packaged in a single−file archive (Enterprise Application archive, or EAR). Workshop web services must be packaged as EAR files when deployed on a WebLogic Server running in production mode. It is not possible to modify an application that is running on a production mode server. However, a application can be updated and then replaced on a production mode server.

WebLogic Server and Workshop provide command line tools to perform tasks related to taking a developed application on one machine and deploying it to another. These command line tools are:

- JwsCompile −− used to generate an EAR that contains all the code for a Web Service application and that can be deployed on another machine.
- weblogic.Deployer −− used to deploy an EAR onto a production mode server.

To Generate an EAR for a Web Service Application

Let's say you've finished developing a Web Service application and are ready to deploy it to a production mode server. Use the JwsCompile command to generate an EAR for that application. For a full description of the jwsCompile tool see JwsCompile Command in the Reference section.

If the root of your project is C:\bea\weblogic700\samples\workshop\applications\foo, and you want to create an EAR for that application called foo.ear and place that EAR in the directory C:/myEARS/, run the following:

```
JwsCompile –p C:\bea\weblogic700\samples\workshop\applications\foo –a –ear C:\myEARS\foo.ear
```

JwsCompile produces an EAR that will work for a server running on a particular hostname and port. The default hostname is whatever machine JwsCompile is run on, and the default port is 7001 (WebLogic Server standard). In order to specify that the EAR is to run on a different machine and port, place a configuration file named weblogic−jws−config.xml in the project's WEB−INF directory or you can use the −hostname and −http−port parameters of jwsCompile at compile−time. Note that the values in weblogic−jws−config.xml will be overridden by the −hostname and −http−port parameters when running JwsCompile. The following sample shows a weblogic−jws−config.xml file for deploying the web service HelloWorld to a machine named ProductionServer:

```
<config>
```

```
 <protocol>http</protocol>
 <hostname>ProductionServer</hostname>
 <http-port>7001</http-port>
 <https-port>7002</https-port>
 <jws>
  <class-name>HelloWorld</class-name>
  <protocol>http</protocol>
 </jws>
</config>
```

For a complete description of the syntax for weblogic−jws−config.xml see weblogic−jws−config.xml Configuration File in the Reference section.

To Start WebLogic Server in Production Mode for Web Services

The mode that the server is running in is determined at startup time. By default, the server runs in design mode. To make the server run in production mode, include a production parameter when calling the startWebLogic script. It is also recommended that you include a nodebug parameter when starting in production mode. For example, if you have installed WebLogic Server on the C drive, the following command will start WebLogic Server in production mode:

C:\bea\weblogic700\samples\workshop\startWebLogic.cmd production nodebug

Note: running Workshop while WebLogic Server is running in production mode is not recommended.

To Deploy a Web Service Application EAR to a Production Mode Server

A server must be running in production mode in order for you to be able to deploy applications to it. Use the weblogic.Deployer command line tool to deploy an EAR to a production mode server. Alternatively, you can deploy through the WebLogic Server console. For instructions on deploying an EAR by either of these methods, see How Do I: Deploy WebLogic Workshop Web Services to a Production Server .

For a complete description of weblogic.Deployer syntax, see weblogic.Deployer in the WebLogic Server 7.0 documentation.

Related Topics

How Do I: Deploy WebLogic Workshop Web Services to a Production Server?

Overview: Clustering

A WebLogic Server cluster is a deployment in which multiple copies, or instances, of an application work together to provide increased performance, especially in high traffic contexts. In cases where an application receives a high volume of requests, the different instances of WebLogic Server in the cluster share the work of processing the requests. From the client's point of view, there appears to only one instance of WebLogic Server servicing the requests.

Clusters also provide failover support. Should one instance of the application fail for some reason, for example, because of a hardware outage, another copy of the application in the cluster can pick up and complete the tasks left incomplete by the failed server.

The server instances that make up the cluster can run a single machine, or they can run on different machines. Each server instance in a cluster must run the same version of WebLogic Server.

To learn more about deploying applications to WebLogic Server clusters see WebLogic Server Clusters.

Related Topics

Clustering Workshop Web Services

Clustering Workshop Web Services

Introduction

Clusters are used to provide scalability and support failover for web resources. The basic clustering model consists of the following elements:

1. One administration server that manages state and configures the other servers in the cluster
2. One HTTP proxy serveradov−−>either a hardware or a software proxy serveradov−−>which receives requests from clients and distributes jobs to the other servers in the cluster
3. Any number of managed servers that actually do the work of servicing requests from clients

All configuration of the cluster takes place on the administration server. All servers in the cluster use the copy of config.xml on the administration server. There may be local copies of config.xml on the managed servers in the cluster, but, these copies of config.xml are ignored in favor of the copy on the administration server.

When Workshop EARs are deployed to a cluster, the managed servers must be homogeneously configured. That is, all managed servers must run the same set of JWS resources: it is not possible to have one set of servers handle some JWSs in an EAR and another set of servers handle other JWSs in an EAR. The following required Workshop resources must also be deployed homogeneously across all servers in a cluster: JDBCConnectionPool, JDBCTxDatasource, JMSQueueConnectionFactory. A JMS Server is also a required Workshop resource, however, it can only be deployed to one server in the cluster.

Configuring Clusters in config.xml

Complete syntax for the config.xml file can be found at WebLogic Server Configuration Reference in the WebLogic Server 7.0 documentation.

The following section hightlights some of the most import elements within a cluster−defining config.xml file (located on the administration server).

The <Cluster> Element

The ClusterAddress attribute specifies a DNS name that maps to the list of IPs of the servers. ClusterAddress does not give the DNS name of the multicast address, which does not require a DNS name. The cluster as a whole can be used as a deployment target. To deploy a J2EE resource to the entire cluster, use the value of the Name attribute as the target of the deployment. See the −targets parameter of the deployment tool, weblogic.Deployer, in the WebLogic Server 7.0 documentation.

Resource Deployment

Resources in the clusteradov−−>such as database connection pools, data sources, and JMS serversadov−−>are defined in the administration server's config.xml file. Resources are not by default universally available across the cluster. The servers they are deployed to are specified by their Targets attributes.

Workshop resources must deployed homogenously across the servers in the cluster. That is, the same Workshop resources must be distributed to each server on the cluster. The connection pool and conversational datasource that Workshop relies on is defined on each managed server in the cluster. For example, the Targets attribute on the JDBCConnectionPool and JDBCDataSource elements specifies each managed server, and each managed server has its own pool of connections.

The queue connection factory that Workshop relies on is defined on each managed server.

However, the JMSServers can only be targeted at one WLS server. Currently Workshop only uses one JMSServer that is targeted, by convention, at the first managed server in the cluster.

Database Support

Workshop clusters currently only support Oracle 8.1.7. To change the supported database edit the JDBC elements within config.xml.

Proxy Server Setup

Clusters can use a software proxy server to distribute HTTP requests across the cluster. The proxy server is also called the sprayer or load balancer. This software proxy is implemented as a web application deployed to the proxy server. You configure the proxy by editing the web.xml descriptor in the proxy application's WAR file. There are entries in the descriptor for specifying what IP addresses and ports the proxy should distribute requests to. For more information about configuring the proxy server see Configure Proxy Plug−Ins in the WebLogic Server 7.0 documentation.

Related Topics

Cluster Deployment

How Do I: Deploy a Workshop Application to a Cluster?

This topic provides step by step instructions for configuring a domain for a cluster of WebLogic Servers and for deploying a Workshop application to that cluster. These instructions applies only to Windows installations of Workshop. For information about cluster hardware configuration see Setting up WebLogic Clusters in the WebLogic 7.0 documentation.

The five basic tasks in setting up and deploying to a cluster are:

- To configure the domain and administration server
- To configure the managed servers
- To configure the software proxy server
- To start the clustered servers
- To deploy the application EAR file to the cluster

To Configure the Domain and Administration Server

1. Select Start−−>BEA WebLogic Platform 7.0−−>Configuration Wizard

2. In the Select a Template dialog, select WebLogic Workshop.
Under Name, type the name you wish to give to your domain. Click Next.

3. In the Choose a Sever Type dialog, select Admin Server with Clustered Managed Server(s). Click Next.

4. In the Choose Domain Location dialog, enter the location of the domain on the local machine.
5. In the Configure Clustered Severs dialog, click Add to enter information about each managed server you wish to included in your cluster. For each managed server, enter:

- Name
  The Name must consist of alphanumeric characters with no spaces.
- Listen Address
  For the Listen Address field see DNS Names or IP Addresses in the WebLogic Server 7.0 documentation.
- Listen Port and SSL Listen Port
  Any port numbers between 1 and 65535 are valid.

Repeat step 5 for each managed server you wish to add to the cluster.

6. In the Configure Cluster dialog, enter

- Cluster Name
  This gives the name of the cluster as a whole.
- Cluster Multicast Address
  Allowable values are in the range 224.0.0.0 − 239.255.255.255
- Cluster Multicast Port
  Any pumber between 1 and 65535 is valid.
- Cluster Address
  The Cluster Address defaults to the address:port combinations for each server instance in the cluster.

7. In the Configure Standalone/Administration Server dialog, enter

- Name
  The Name must consist of alphanumeric characters with no spaces.
- Server Listen Address
  For the Listen Address field see DNS Names or IP Addresses in the WebLogic Server 7.0 documentation. The listen address you enter here should be the same as was entered in step 5 above.
- Server Listen Port and SSL Listen Port
  Any port number between 1 and 65535 are valid. The ports you enter here should be the same as you entered in step 5 above.

8. In the Create Administrative User dialog, enter a username and password. The username and password are required to start the server instances in the cluster. The username, such as "system" must belong to a role that is permitted to start server instances. For information about allowable usernames see the Protecting System Administration Operations in BEA WebLogic Server Administration Guide.
9. In the Create Start Menu Entry For Server dialog, choose whether you want to place a link to the administration server on the Start Menu.
10. In the Configuration Summary dialog, click Create. It is recommended that you cut and paste the configuration summary to a text editor for future reference.
11. In the Configuration Wizard Complete dialog, select End Configuration Wizard, and click Done.
The Domain Configuration Wizard will configure your domain and generate the domain's config.xml file according to the values specified.

To Configure the Managed Servers

This task must be repeated for each managed server that you specified in step 5 above.

12. Select Start−−>BEA WebLogic Platform 7.0−−>Configuration Wizard

13. Under Select a template, select WebLogic Workshop. Under Name, enter the domain name you entered when configuring the administration server.

14. In the Choose Server Type dialog, select Managed Server (with owning Admin Server configuration)

15. In the Choose Domain Location dialog, enter the location of the domain on the local machine.

16. In the Configure Administrative Server Connection dialog, enter:

- Admin Server Name or IP
  See DNS Names or IP Addresses in the WebLogic Server 7.0 documentation.
- Admin Server Listen Port
  Any port number between 1 and 65535 are valid. The ports you enter here should be the same as you entered in step 5 above.
- Admin Server SSL Listen Port
  Any port number between 1 and 65535 are valid. The ports you enter here should be the same as you entered in step 5 above.
- Managed Server Name
  Enter the DNS name of the managed server.

17. In the Configure Standalone/Administration Server dialog, enter:

- Name
  The Name must consist of alphanumeric characters with no spaces.
- Server Listen Address
  For the Listen Address field see DNS Names or IP Addresses in the WebLogic Server 7.0 documentation. The listen address you enter here should be the same as was entered in step 5 above.
- Server Listen Port and SSL Listen Port
  Any port number between 1 and 65535 are valid. The ports you enter here should be the same as you entered in step 5 above.

18. In the Create Administrative User dialog, enter a username and password. The username and password are required to start the server instances in the cluster. The username, such as "system" must belong to a role that is permitted to start server instances. For information about allowable usernames see the Protecting System Administration Operations in BEA WebLogic Server Administration Guide.

19. In the Create Start Menu Entry For Server dialog, choose whether you want to place a link to the administration server on the Start Menu.

20. In the Configuration Summary dialog, click Create. It is recommended that you cut and paste the configuration summary to a text editor for future reference.

21. In the Configuration Wizard Complete dialog, select End Configuration Wizard, and click Done.

To Configure the Software Proxy Server

If you are using a software proxy server, you must build and configure the proxy application that will service requests from clients. For information about software proxy server see Configure Proxy Plug−Ins in the WebLogic Server 7.0 documentation.

To Start the Cluster

To start a cluster, you must first start the administration server and then each of the managed servers. Do not start the managed servers until the administration server has been started.

To start the administration server:

22. Open a command shell, and cd to the domain directory that you created in step 4 above.

23. Enter the following command:
```
startWebLogic production nodebug.
```

To start a managed server:

24. Open a command shell, and cd to the domain directory you created in step 4 above.

25. Enter the command

```
startManagedWeblogic [managed server DNS name or IP address] [URL of the administration server:port]
```

For example,
```
startManagedWeblogic managedServer1 http://adminServer:7133
```

Repeat steps 24 and 25 for each managed server in the cluster.

To Deploy the Application EAR File to the Cluster

Before you deploy a Workshop application to a server cluster, you must first generate an EAR file for the application. For instructions on generating an EAR file, see How To Generate an EAR for a Web Service Application.

When you make an EAR using jwsCompile, make sure to specify the −http−port, −https−port and −hostname parameters. The −http−port and −https−port should give the ports that clients will use to call the proxy server. The −hostname parameter should specify the proxy server's DNS name. Once you have generated an EAR, you must deploy it to the servers in your cluster. To deploy an EAR file to a cluster, you can either use the command line tool weblogic.Deployer or the WebLogic Server console. To learn more about using weblogic.Deployer to deploy an EAR file to cluster, see weblogic.Deployer Utility and weblogic.Deployer in the WebLogic Server 7.0 documentation. To learn more about using the WebLogic Server console to deploy an EAR file see WebLogic Administrative Console.

Related Topics

Deploying Web Services

How Do I: Deploy WebLogic Workshop Web Services to a Production Server?

How Do I...

Getting Started

Learn what I need to know about Java?

Learn what I need to know about XML?

Get started creating WebLogic Workshop web services?

Start a new project?

Start or stop WebLogic Server?

Create a new web service?

Test a web service using WebLogic Workshop's Test View?

Debug a web service?

Find samples of WebLogic Workshop web service?

Using Databases

Use a database from a web service?

Administer the default PointBase database?

Use a "WHERE... LIKE" Clause in a Database Control?

Connect a database control to a different database (SQL, Oracle)?

XML Messaging and Maps

Add or Edit an XML Map with the Edit Maps and Interface Dialog?

Handle XML Messages of a Particular Shape?

Ensure That Outgoing XML Messages Conform to a Particular Shape?

Use ECMAScript (JavaScript) to process XML messages?

Work with XML Documents Directly?

Begin a Reusable XML Map?

Web Service Clients

Publish a WSDL file for my web service?

Use the Java proxy for a web service?

Control the protocol and message formats a web service supports?

Generate a WSDL File?

Tell developers of non−WebLogic Workshop clients how to participate in conversations?

Communicate with a web service from a JSP file?

Communicate with a web service from a servlet?

Exchange messages with a JMS queue or topic from a web service?

Use Java code from a web service?

Deployment and Administration

How Do I: Reset the State of WebLogic Server?

How Do I: Configure WebLogic Workshop to Use a Different Database for Internal State?

Deploy a web service to a production server?

Publish a Web Service to a UDDI Directory?

Move or copy a web service from one project to another?

Use a web service that is in another project?

Create a New WebLogic Workshop−enabled WebLogic Server Domain?

WebLogic Workshop−Enable an Existing WebLogic Server Domain?

How Do I: Start WebLogic Workshop?

WebLogic Workshop is a visual development environment for building enterprise−class web services. Once you have installed the WebLogic platform, you can start WebLogic Server using the following steps.

To Start WebLogic Workshop on Microsoft Windows

- Click Start−−>Programs−−>BEA WebLogic Platform 7.0−−>WebLogic Workshop−−>WebLogic Workshop.

To Start WebLogic Workshop on Linux

1. Using a file system browser or shell, locate the Workshop.sh file at:

$HOME/bea/weblogic700/workshop/Workshop.sh

2. Run Workshop.sh as follows:

- From the command line, use the following command:

```
sh Workshop.sh
```

- From a file system browser, double−click Workshop.sh.

Related Topics

Tutorial: Your First Web Service

How Do I: Start A New Project?

A WebLogic Workshop project may contain one or more web services. Typically the web services in a project are related in some way. The project name you provide will be visible to clients accessing your web service, so it should describe the collective purpose of the services in the project.

When you create a project in WebLogic Workshop, you are also creating a WebLogic Server web application. For more information about projects and web applications, see WebLogic Workshop Projects.

To Create a Project from WebLogic Workshop

1. In the WebLogic Workshop visual development environment, select the File menu, then New Project. The New Project dialog appears.
2. In the Project Name field, type the name you would like to give your new project.

This name will become part of the URL used to access web services in the project. For example, for a deployed web service, the name you enter in the Project Name box corresponds to MyProject in the following URL:

http://server:port/MyProject/MyService.jws

For services you are testing on your development computer, you will use a URL similar to the following:

http://localhost:7001/MyProject/MyService.jws

In addition to being part of the URL of all web services in the project, the project name is also used as the Java package name for all Java classes declared in the project. Therefore, the project name must be a valid Java identifier.

3. Click OK. This creates a new folder with the name you entered in step 1.

4. In the Create New File dialog, under Type of file to create, confirm that Web service is selected.

After you create a new project, WebLogic Workshop automatically prompts you to create the first file in the project. You can create a new web service or other file, click Cancel to create an empty project.

Related Topics

WebLogic Workshop Projects

How Do I: Move or Copy a Web Service from One Project to Another?

How Do I: Create a New Web Service?

In WebLogic Workshop, the implementation of a web service is contained in a JWS (Java Web Service) file. JWS files are standard Java class files with special annotations that provide powerful web service functionality. Creating a new web service is as simple as creating a new file.

For more information about JWS files, see JWS Files: Jave Web Services.

To Create a New Web Service in the Root of an Existing Project

1. Choose File−−>New−−>New Web Service. The Create New File dialog appears.

2. In the Type of file to create menu, confirm that Web service is selected.
3. In the File name field, enter the name for your new service.

This name will become part of the URL used to access your web service. The name will also be used as Java class name in your web service's JWS file, so it must be a valid Java identifier.

A default web service template will be generated for you to begin editing and adding methods.

To Create Your Web Service in a Directory Other Than the Project Root

1. Expand the Project tree.
2. Right−click the folder in which you would like to create the service, then click New File.
3. Follow steps 2 and 3 in the preceding procedure.

Related Topics

How Do I: Start a New Project?

Tutorial: Your First Web Service

How Do I: Start or Stop WebLogic Server?

The web services you develop in WebLogic Workshop are deployed to a running instance of WebLogic Server. Before you can compile, run or debug a web service, WebLogic Server must be running. If you attempt to perform these operations when the server is not running, WebLogic Workshop will offer to start the server for you.

If you wish to start the server yourself, you can start or stop WebLogic Server as follows:

To Start WebLogic Server

- In WebLogic Workshop, click the Tools menu, then click Start WebLogic Server.

or

- From a command line, change directory to the root directory of the WebLogic Server domain in which you'd like to start WebLogic Server. The root directory of the WebLogic Workshop sample server in the workshop domain is BEA_HOME/weblogic700/samples/workshop. Then type startWebLogic.

To Stop WebLogic Server

- In WebLogic Workshop, click the Tools menu, then click Stop WebLogic Server.

or

- Shut down the server from the WebLogic Server console. In the default configuration, the console can be reached at http://localhost:7001/console. In the tree pane on the left hand side, navigate to workshop/Servers/cgServer (select cgServer). In the tabbed pane on the right hand side, select the Control tab. Then select "Shutdown this server..."

Related Topics

How Do I: Debug a Web Service?

How Do I: Debug a Web Service?

If you are having trouble figuring out why the web service you have written is not working, you can use the debugger to step through your source code line−by−line and locate the problem. The debugger allows you to set breakpoints, step through your code line−by−line, view local variables, set watches on variables and expressions, and view the call stack and exception information. Note that when you debug a service in WebLogic Workshop, you are debugging the service as it is deployed on your development server.

To Start the Debugger

1. Position the cursor on a line where you wish to break, click the Debug menu, then click Toggle Breakpoint.

Note: When you toggle a breakpoint you should see a red circle appear on the left edge of the Source View. You may quickly set and clear breakpoints by clicking where you see this circle.

2. Click the Debug menu, then click Start and Debug.

This will launch a browser as if you had clicked Start, except that the service will be running in debug mode. You should see the Debug Window appear at the bottom of the Source View in WebLogic Workshop.

3. Switch to the browser and invoke a method.

When the line on which you have set a breakpoint is run, the web service will be paused and you will be able to step through the service's code one line at a time. You may inspect and change the values of variables that are currently defined and see the call stack by using the Debug Window.

To learn more about debugging with WebLogic Workshop, see Debugging Web Services.

Related Topics

Source View

How Do I: Test A Web Service Using WebLogic Workshop's Test View?

WebLogic Workshop provides a browser−based interface through which you can test the functionality of your web service. With this interface, Test View, you play the role of the client, invoking the service's methods and viewing the service's responses. For more complex web services, you can follow the path of processing through multiple requests and responses, even multiple services.

The tutorial included with this documentation provides an introduction to Test View, as well as other aspects of WebLogic Workshop. For more information, see Tutorial: Your First Web Service.

To Test Your Web Service with Test View

1. In WebLogic Workshop, open the JWS file of the web service you would like to test.
2. Ensure that WebLogic Server is running. If it is running, the following indicator will be visible in the status bar at the bottom of the WebLogic Workshop visual development environment:

- To start WebLogic Server if it is not running, click the Tools menu, then click Start WebLogic Server.

3. Click the Start button, shown here:

Clicking the Start button prompts WebLogic Workshop to build your project, checking for errors along the way. It then launches a web browser to display Test View, through which you can test the service with sample input values.

The Test Form and Test XML tabs provide alternative ways to specify the content of the call to your service's methods. The others—Overview and Console—include additional information about the service.

4. On the Test Form page, in the boxes provided, enter data that you web service might receive as part of a client request.
5. Click the button labeled with your service's method name to invoke the method with the values you entered.

The Test Form page will refresh to display a summary of your request parameters and your service's response.

- Under Service Request, the summary displays what was sent by the client (you) when the method was called, including the values of method parameters.
- Under Service Response, the summary displays the XML response returned by the service.

For services that involve multiple communications with clients, or communications with resources such as other web services, the Message Log at the left of the Test Form page displays an entry for each call to a method or callback so that you can view the data for each. Select any log entry to see the details of that interaction.

If your web service participates in a conversation with clients, the Test Form page will display the conversation ID in the Message Log. Select the conversation ID to access continue and finish methods in that conversation.

Related Topics

Test View

How Do I: Administer WebLogic Server Through the Administration Console?

In some cases you may need to configure WebLogic Server while you're building or testing your web service. For example, if you're connecting to a database, you may need to configure the server to add a new connection pool and data source. You can use the WebLogic Server administration console to administer WebLogic Server.

To Launch the WebLogic Server Administration Console

1. Make sure that WebLogic Server is running.
2. If you are building a web service against your development server, navigate to http://localhost:7001/console/ in your web browser. If you are building a web service against a remote server, use the remote server's name.
3. Enter your user name and password. If you are administering your development server, the default user name is installadministrator and the default password is installadministrator.

How Do I: Delete a WebLogic Workshop Project?

To delete a WebLogic Workshop project, you must remove all of the components and files associated with the project.

To Delete a WebLogic Workshop Project

1. Stop WebLogic Server.
2. Open the config.xml file that's located under the Workshop directory in your installation, in the \weblogic700\samples\workshop\config.xml directory.
3. Find the Application node that's associated with the project you are deleting, and delete this Application node. It will have a WebAppComponent node whose Name attribute is the name of your project, as shown in the following example:

```
<Application Name="_appsdir_testProject_dir" Path="C:\bea\weblogic7\samples\workshop\applications"
StagingMode="nostage" TwoPhase="true">
<WebAppComponent Name="testProject" Targets="cgServer" URI="testProject"/>
</Application>
```

4. Find the Application node that's associated with the EJBs in your project and delete it. Each JWS file in your project has an associated EJB whose name corresponds to the JWS file in your project, which appears in the Name attribute of the EJBComponent node, as shown in the following example:

```
<Application Deployed="true" Name="testProject.testJWS_EJB"
Path="C:\bea\weblogic7\samples\workshop\cgServer\.jwscompile\_jwsdir_testProject\EJB"
TwoPhase="true">
<EJBComponent Name="testJWSEJB" Targets="cgServer" URI="testJWSEJB.jar"/>
</Application>
```

5. Save the config.xml file.
6. Delete the project directory and files from your hard drive.
7. Restart WebLogic Server.

Related Topics

How Do I: Administer WebLogic Server Through the Administration Console?

How Do I: Get Started Creating WebLogic Workshop Web Services?

The WebLogic Workshop visual development environment simplifies web service development by letting you focus on application logic rather than application infrastructure. In particular, WebLogic Workshop offers the Design View (through which you can "draw" your service's design), properties as Javadoc annotations (through which you configure your service to support powerful server features), and controls that provide uniform access to resources such as databases.

To Get Started with WebLogic Workshop

1. Build a web service using the tutorial, Tutorial: Your First Web Service.

The best way to learn is by doing. The tutorial provides an introduction to WebLogic Workshop and to web services. In a few hours, you can build an web service that uses many of the features available in WebLogic Workshop.

2. Take a look at Building Web Services with WebLogic Workshop for an overview of how WebLogic Workshop simplifies the task of building web services.
3. Scan How Do I...? for tasks you may be interested in accomplishing with WebLogic Workshop.

This topic provides links to detailed instructions for many of the things you can accomplish with WebLogic Workshop.

Related Topics

How Do I: Find Samples of WebLogic Workshop Web Services?

There are numerous fully documented samples provided with WebLogic Workshop. These samples can be copied or examined to find a coding pattern that would help you to solve a problem you may have.

By default, WebLogic Workshop starts in the samples project, which is where the sample web services are located. If WebLogic Workshop is not currently open in the samples project:

1. Click the File menu, then click Open Project. This displays a dialog that lists the projects stored on your server.

2. Click samples, then click Open.

Related Topics

Samples

How Do I: Learn What I Need to Know About Java?

If you are already familiar with another procedural programming language, there are only a handful of things you need to learn about Java in order to use WebLogic Workshop successfully. These are discussed in Introduction to Java.

Related Topics

Introduction to Java

How Do I: Learn What I Need to Know About XML?

The Introduction to XML section describes all that you will need to know to get started using XML in WebLogic Workshop.

Related Topics

Introduction to XML

How Do I: Use a Database from a Web Service?

You can access a database from a web service by using a Database control.

To Add an Existing Database Control to Your Web Service

1. In Design View, from the Add Control drop−down list, select Add Database Control. The Add Database Control dialog appears.
2. In step 1, enter the name you will use to refer to the new Database control. This name must be a valid Java identifier.
3. In step 2 of the dialog, click Use a Database control already defined by a CTRL file, then enter the name of the CTRL file or click Browse to find the file in your project.

4. Click Create.

To Create a New Database Control for Your Web Service

1. In Design View, from the Add Control drop−down list, select Add Database Control. The Add Database Control dialog appears.
2. In step 1, enter the name you will use to refer to the new Database control. This name must be a valid Java identifier.
3. In step 2 of the dialog, click Create a new Database control to use with this service.
4. In the New CTRL name field, type a name for the CTRL file that you wish to create. This name will be used as the name for the Java interface that defines the Database control, so it must be a valid Java identifier.
5. In step 3 of the dialog, in the Data source field, type the name for the data source you wish to use for this Database control or click Browse to select from a list of available JDBC data sources. The cgSampleDataSource data source is intended for experimentation.
6. Click Create.

Once you have added the Database control to your web service, you may invoke methods on the control as you would with any other Java object.

Related Topics

Database Control: Using a Database from Your Web Service

Controls: Using Resources from a Web Service

How Do I: Connect a Database Control to a Database Such as SQL Server or Oracle?

How Do I: Administer the Default Pointbase Database?

You can administer the default database installed with WebLogic Server (PointBase) using the PointBase administrative console, or a number of third party database visualization and management tools.

To Launch PointBase Console from the Windows Start Menu

1. From the Start menu, choose PointBase Console under BEA WebLogic Platform 7.0−−>WebLogic Workshop | WebLogic Workshop Examples.
2. When the console starts up, you will be prompted to enter connection parameters to properly connect to the database. This connection information is also what you will need to use a third−party product.

- Driver: com.pointbase.jdbc.jdbcUniversalDriver
- URL: jdbc:pointbase:server://localhost:9093/cajun
- User: cajun
- Password: abc

To Launch PointBase Console from the Command Line

Run the following command:

```
java −cp
WL_HOME\samples\server\eval\pointbase\lib\pbclient42ECF172.jar;WL_HOME\samples\server\eval\pointbase\lib\p
com.pointbase.tools.toolsConsole com.pointbase.jdbc.jdbcUniversalDriver
```

as all one line and with WL_HOME replaced with the root directory of the WebLogic Platform installation (e.g. c:\bea\weblogic700).

How Do I: Use a "WHERE... LIKE" Clause in a Database Control?

Curly braces "{}" within literals (strings within quotes) are ignored. This means statements like the following will not work as you might expect:

```
/**
 * @jws:sql statement::
 *     SELECT name
 *     FROM employees
 *     WHERE content LIKE '%{partialName}%'
 * ::
 */
public String[] partialNameSearch(String partialName);
```

Since the curly braces are ignored inside the literal string, the expected substitution of the partialName Java String into the SELECT statement does not occur. To avoid this problem, pre−format the match string in the JWS file before invoking the Database control method, as shown below:

```
String partialNameToMatch = "%" + matchString + "%"
String [] names = myDBControl.partialNameSeach(partialNameToMatch);
```

For more information about parameter substitution in Database control methods, see Parameter Substitution in @jws:sql Statements.

Related Topics

Database Control: Using a Database from Your Web Service

How Do I: Connect a Database Control to a Database Such as SQL Server or Oracle?

WebLogic Server manages the databases you can use and allows you to access configured databases through JDBC data sources. If you wish to use a new database, there must be a JDBC data source set up that allows you to access that database.

Each WebLogic Workshop Database control determines which data source it will use from its connection property (@jws:connection tag).

To Change the Data Source Used by a Database Control

1. In Design View, click the Database control your service will use to access the database.
2. In the Properties pane, expand the connection property.
3. Click the data−source−jndi−name attribute, then select the value corresponding to the data source you want to use.

To Configure a New Connection Pool and Data Source

To access an Oracle, SQL Server, or Informix database, you first need to install the appropriate JDBC driver. For evaluation versions and installation information, see:

Installing WebLogic jDriver for Microsoft SQL Server

Installing WebLogic jDriver for Oracle

Installing WebLogic jDriver for Informix

Once you have installed the driver, you must configure a new connection pool and data source in WebLogic Server, using the WebLogic Server Console application. If you are running WebLogic Workshop and WebLogic Console on the same computer, you can access the console as follows. Otherwise, check with your system administrator to configure WebLogic Server.

1. Launch the console by navigating to http://localhost:7001/console/.
2. Provide your user name and password; by default, both are set to installadministrator.

Next, create a new connection pool:

1. Navigate to Connection Pools in the JDBC section.
2. Click Configure a new JDBC Connection Pool.
3. On the Configuration tab, click General.
4. Fill in the fields on the General tab as shown in the following table:

Field	Description	Example (using WebLogic jDriver for Microsoft SQL Server)
Name	The JNDI name for the connection pool.	sqlPool
URL	The URL for this database. The URL is passed to the driver to create the physical database connections.	jdbc:weblogic:mssqlserver4v70:sqldbname:1433
Driver Classname	The name of the WebLogic jDriver class	weblogic.jdbc.mssqlserver4v70.Driver

Properties	Name−value pairs that provide connection information to the driver	user=sa
		
		password=
		
		db=pubs
		
		server=sqldbname
		
		port=1433

ACL	Specify the ACL used to control access to this connection pool. By default no ACL is specified, which allows any user open access, provided that user passes other WebLogic Server security controls.	
Password	Encrypted password for driver to use in accessing database.	

5. Apply your changes.
6. On the Targets tab, choose the target server on which you want to deploy this connection pool.
7. Apply your changes and restart WebLogic Server.

Once you set up a connection pool, you must set up a data source based on that connection pool. The name you provide for the data source is the name that you will use to set the data−source−jndi−name attribute of the connection tag for the Database control.

WebLogic Workshop provides default transaction semantics for web service operations. The default transaction semantics include wrapping database operations that occur within a web service operation in a transaction. The default transaction semantics require use of a TxDataSource (transacted data source), instead of a DataSource. If you use a Tx Data Source, you may not attempt to control transaction semantics directly (e.g. via calls to java.sql.Connection.setAutoCommit). The sample data source cgSampleDataSource is a Tx Data Source.

To learn more about WebLogic Workshop's default transaction semantics, see Transactions in WebLogic Workshop.

1. Depending on whether you want default transaction semantics or not, navigate to Data Sources or Tx Data Sources in the WebLogic Server Console.
2. Choose Configure a new JDBC data source.
3. Specify a friendly name in the Name field for this data source.
4. Specify the JNDI name by which you will refer to this data source.
5. Specify the name of the pool you created previously in the Pool Name field.
6. Click the Targets tab and choose the target server on which you wish to deploy this data source.
7. Restart WebLogic Server.

For more information on configuring connection pools and data sources, see Configuring WebLogic JDBC Features.

Related Topics

Database Control: Using a Database from Your Web Service

How Do I: Use a Database from a Web Service?

How Do I: WebLogic Workshop−Enable an Existing WebLogic Server Domain?

@jws:connection Tag

How Do I: Use ECMAScript (JavaScript) to Process XML Messages?

Sometimes the transformations needed to process incoming or outgoing XML messages are more complex than you can accomplish with static XML maps. In these cases, you can use WebLogic Workshop's extended version of ECMAScript to perform XML mappings. To do so, you must first create a script file (with a JSX extension) in your project to hold the script.

To Create a Script File

1. Click the File menu, point to New, then click New JavaScript File. The Create New File dialog appears.
2. Confirm that JavaScript is selected.
3. In the File name field, type a name for the new script file.
4. Click OK.

The newly created JSX file contains two functions, named myScriptFromXML and myScriptToXML. These are used to map from XML to Java types and to XML from Java types. The newly created JSX file also contains some rudimentary instructions about how to use the script from a map.

From within a map, script may be called by using the following syntax:

```
{ScriptFileName.myScript(parameters)}
```

Notice that the function name in the call does not include "ToXML" or "FromXML". The appropriate method in the JavaScript file is determined automatically based on the direction of the mapping being performed.

parameters will depend on whether you are mapping the parameters or the return value of your Java declaration. If you are placing the call to script in a parameter−xml map, parameters should be the arguments of the method or callback that you would like this script method to populate. If you are placing the call to script in a return−xml map, you should simply enter "type return" where type is the method or callback's declared return type.

Related Topics

Handling XML with ECMAScript Extensions
Handling and Shaping XML Messages with XML Maps
How Do I: Create and Invoke a Script for Use with XML Messages?

How Do I: Work with XML Documents Directly?

XML maps allow you to control how XML messages are mapped to the Java arguments and return value of a method. However, sometimes you want to get at the XML directly, without using a map of any kind.

To Receive the Input of a Method as Raw XML

1. In Source View, at the top of your web service, add an import for the XML Document Object Model (DOM) classes:

import org.w3c.dom.*;

2. Edit the method that will be receiving XML so that the parameter holding the XML is a org.w3c.dom.Node object.

This gives you direct access to the XML sent to your web service. You can access the data using the DOM programming API. For more information about the API, see Xerces API Documentation.

To Generate Raw XML for the Return value of a Method

1. In Source View, at the top of your web service, add imports for the DOM classes and for the Xerces Document implementation.

import org.w3c.dom.*; import org.apache.xerces.dom.DocumentImpl;

2. In your method, you first must create a Document object. This object will be used as a factory for creating nodes in the XML tree.

Document doc = new DocumentImpl();

3. Create the root tag of the XML to return by calling createElement.

```
Element root = doc.createElement("myTag");
```

4. You can set the attributes on the tag by calling setAttribute.

```
root.setAttribute("myAttr", "value");
```

5. You can add child tags by calling appendChild.

```
Element child = // ... root.appendChild(child);
```

6. You can add text into the body of the tag by appending a child node that is a Text object.

```
child.appendChild(doc.createTextNode("some text"));
```

7. At the end of your method, return the root tag of the XML.

```
return root;
```

Related Topics

Handling XML with ECMAScript Extensions

Handling and Shaping XML Messages with XML Maps

How Do I: Begin a Reusable XML Map?

You can store XML map code in a file separate from your JWS or CTRL file. This enables you to call a map from code in multiple services and use a map in multiple projects. When creating an XML map file, you add a new file to your project, give it an .xmlmap extension, and add code to make it a self−contained map file. To use a map in a XMLMAP file, you refer to it from within a parameter−xml or return−xml map using the <xm:use> map tag.

Note: Whenever possible, it is a good practice to create and debug an XML map with the Edit Maps and Interface dialog because it provides code completion and error checking. You can then remove the map to a separate file and enclose it with the tags needed to make it self−contained. For more information on the Edit Maps and Interface dialog, see How Do I: Add or Edit an XML Map with the Edit Maps and Interface Dialog?

For more detailed information on what makes up an XML map file, see Creating Reusable Maps.

To Create an XMLMAP file

1. In Design View, choose File−−>New File. The Create New File dialog appears.
2. Click Text.
3. In the File name field, type the name of the XMLMAP file.

Note: If this map file will contain multiple maps, you may want to give it a name that conveys a sense of the maps as a group, such as POMaps for maps that handle purchase orders.

4. In the File extension field, type xmlmap, which is the extension all XMLMAP files use.

5. Click OK. The new map file opens in Source View.

6. Enter the following at the top of the newly created empty map file:

<xm:map−file xmlns:xm="http://www.bea.com/2002/04/xmlmap/">

This identifies the file as a map file.

7. After this line, begin a new map with code similar to the following, replacing text as described below:

<xm:xml−map signature="mapName(datatype parameter)"> ...text of the XML map... </xm:xml−map> </xm:map−file>

- mapName adov−−> Enter a name for this map that is unique in the context of this file. There may be multiple maps in the file, each with different names.
- datatype parameter adov−−> Enter parameters for this map's signature. Note that the parameter names specified in the map's signature attribute do not need to match the parameter names of the method to which the map is being associated. However, the types of the parameters in the map signature and in the invocations of the map must agree in both type and order adov−−> in other words, the same rules apply as when making a method call.

For more information on constructing the text of an XML map, see How Do XML Maps Work? and Matching XML Shapes.

8. Enter additional maps as needed, enclosing each map between <xm:xml−map> tags as in the preceding steps.
9. End the map file with an </xm:map−file> tag.

Here is an example of code you might create with this procedure:

```
<xm:map−file xmlns:xm="http://www.bea.com/2002/04/xmlmap/">
    <xm:xml−map signature="placeOrder(String partID, int quantity)">
        ... text of the XML map...
    </xm:xml−map>
</xm:map−file>
```

To Refer to a Map in a XMLMAP File from Within an XML Map

1. Locate the source code for the method or callback that will use an XML map that is stored in the map file.
2. Immediately preceding the method's declaration, in the Javadoc comment containing attributes, enter the following, replacing sample text as described below:

/* * @jws:mode−xml xml−map:: * <methodName> * <xm:use call="MapFileName.mapName(datatype parameter)"/> * </methodName> * :: */

- mode adov−−> This should be "parameter" or "return," depending on which sort of map this is.
- methodName adov−−> The name of the method or callback to which this map call applies. For example, if the method declaration following this annotation is public String reportRequest(String applicantSSN), methodName would be reportRequest.
- MapFileName adov−−> The name of the XMLMAP file you created. Note that it may be necessary to prepend path information, as described in Creating Reusable XML Maps.

- mapName adov−−> The name of the map to be used. This name corresponds to the mapName in the preceding procedure.
- datatype parameter adov−−> Parameters from the declaration of your method or callback if this is a parameter−xml map; if this is a return−xml map, the parameter here takes form: datatype return, where "return" is literal.

Here is an example of code to invoke a parameter−xml map:

```
    /*
     * @jws:parameter-xml xml-map::
     *   <methodName>
     *     <xm:use call="CustomerRequests.placeOrder(String)"/>
     *   </methodName>
     * ::
     */
```

Related Topics

[Creating Reusable XML Maps]()

[How Do I: Add or Edit an XML Map with the Edit Maps and Interface Dialog?]()

How Do I: Handle Incoming XML Messages of a Particular Shape?

When your service must handle a particular incoming XML message shape, you can create an XML map as an translation layer between the incoming message and the Java parameters that your method expects. XML maps are particularly useful in cases where a web service you develop must respond to messages whose shape is beyond your control. Maps enable you to handle those messages without changing your code.

In the simplest example, you specify this mapping with substitution directive such as {} for the XML element value you want to map. In the following example, parameters in the declaration are mapped to element and attribute values.

```
/**
 * @jws:operation
 * @jws:parameter-xml xml-map::
 *   <searchRequest>
 *     <partName partID="{serialNumber}">{productName}</partName>
 *     <number>{quantity}</number>
 *   </searchRequest>
 * ::
 */
public String searchRequest (String productName, String serialNumber, int quantity)
```

Note: You will find an introduction to XML maps at [Why Use XML Maps?]()

Depending on how the incoming message arrives at your service, you will create a map that converts an XML message into the parameters of your Java method. The following lists the four circumstances under which an incoming XML message can arrive at your service, and describes which kind of map you should use for each.

Note: For each of the incoming message source described in this topic, there is a related outgoing message source. For more information on mapping to shape outgoing messages, see [How Do I: Ensure that Outgoing XML Messages Conform to a Particular Shape]().

An incoming message can arrive:

- From a client as a call to a method of your service.

Use a parameter−xml map on your web service's method. You may want to map data in the incoming XML message to the Java types of your method's parameters.

- As a callback received from a control.

Use a parameter−xml map on the callback from your control. When your service implements a callback handler you need to map the XML input from that control into the parameters of the callback handler Java function.

Note: When you edit a map associated with a control, you are modifying the CTRL file that defines the control. Remember that the CTRL file may be used by other web services. If you wish to edit maps on a control without affecting other web services, you should create a new control that is not shared.

For more information about callbacks, see Using Callbacks to Notify Clients of Events.

- The return value of a method call to a control.

Use a return−xml map on the control method. When your service calls a method of a control, you may need to map the returned XML message to the Java return type.

Note: When you edit a map associated with a control, you are modifying the CTRL file that defines the control. Remember that the CTRL file may be used by other web services. If you wish to edit maps on a control without affecting other web services, you should create a new control that is not shared.

- As the return value of a callback on the client.

Use a return−xml map on a callback from your service to a client to map the XML return message from the call into the Java return value of the callback.

In practice, you apply an XML map in roughly the same way (using the same syntax and tags) regardless of the message's source. The Edit Maps and Interface dialog provides an easy way to begin an XML map.

For more information, see How Do I: Add or Edit an XML Map with the Edit Maps and Interface Dialog?

Related Topics

Why Use XML Maps?

How Do I: Ensure That Outgoing XML Conforms to a Particular Shape?

How Do I: Add or Edit an XML Map with the Edit Maps and Interface Dialog?

Using Callbacks to Notify Clients of Events

How Do I: Ensure that Outgoing XML Messages Conform to a Particular Shape?

When your service must generate a particular outgoing XML message shape, you can create an XML map as an translation layer between the specific Java types of your service's implementation and the corresponding outgoing message. XML maps are particularly useful in cases where a web service you develop must generate messages whose shape is beyond your control. Maps enable you to control the shape of those messages without changing the implementation of your class.

Note: You will find an introduction to XML maps at Why Use XML Maps?

Depending on how the outgoing message leaves your service, you will create either a parameter−xml map or a return−xml map. The following lists the four circumstances under which an outgoing XML message can leave at your service, and describes which kind of map you should use for each.

Note: For each of the outgoing message sources described in this topic, there is a related incoming message source. For more information on mapping to handle incoming messages, see How Do I: Handle XML Messages of a Certain Shape?

An outgoing message can be sent:

- To a client as the returned response to a call to a method of your service.

Use a return−xml map on your web service's method. Here, you need to map data from your method's return value to the desired outgoing message shape.

- As a return value of a callback sent to your service by a control.

Use a return−xml map on the callback handler for the control's callback. When your service implements a callback handler to respond to a callback sent by a control (including another service), that handler may return a value you must map to the outgoing message.

Note: When you edit a map associated with a control, you are modifying the CTRL file that defines the control. Remember that the CTRL file may be used by other web services. If you wish to edit maps on a control without affecting other web services, you should create a new control that is not shared.

For more information about callbacks, see Using Callbacks to Notify Clients of Events.

- As the call from your web service to a control's method.

Some control types allow you to specify XML maps to be applied to method calls to that control. This is most typically used with Service controls, which represent another web service.

Use a parameter−xml map on the control's method. When your service calls a method of another service, you must pass any needed parameter values with the method call. You can map the Java types of the parameters to the desired message shape.

Note: When you edit a map associated with a control, you are modifying the CTRL file that defines the control. Remember that the CTRL file may be used by other web services. If you wish to edit maps on a control without affecting other web services, you should create a new control that is not shared.

- As a callback sent to a client.

Use a parameter−xml map on the callback. Given that this sort of callback (the kind sent from your service to clients) is an aspect of the interface your service defines, it is likely that you will control the message format, thus eliminating the need for an XML map. However, when the need arises you can map the Java types of your callback's parameters to a desired outgoing message shape.

The Edit Maps and Interface dialog provides a convenient way to begin an XML map.

For more information, see How Do I: Add or Edit an XML Map with the Edit Maps and Interface Dialog?

Related Topics

Why Use XML Maps?

How Do I: Handle XML Messages of a Certain Shape?

How Do I: Add or Edit an XML Map with the Edit Maps and Interface Dialog?

Using Callbacks to Notify Clients of Events

How Do I: Publish A WSDL File For My Web Service?

At any time in the development process, the Web Service Description Language (WSDL) file describing your web service is available from WebLogic Server. WSDL is a standard XML document type controlled by the World Wide Web Consortium (W3C, see www.w3.org for more information).

WSDL files describe all the methods a web service exposes (in the form of XML messages it can accept and send), as well as the protocols over which the web service is available. The WSDL file provides all the information a client application needs to use the web service.

There are two ways to obtain the WSDL file corresponding to a JWS file. One is through a URL to a page that others (including clients) can also use to get information about your service, including its WSDL file. The other is through a command in WebLogic Workshop that you can use when you merely wish to generate a WSDL for your own use. For more information about the second, see How Do I: Generate a WSDL File?

To Generate a WSDL File from a JWS File in WebLogic Workshop

1. In the Project tree, browse to the JWS file for which you would like to generate a WSDL file.
2. Right−click on the JWS file in the Project tree and select Generate WSDL from JWS.

If the name of the JWS file is MyService.jws, a file with the name MyServiceContract.wsdl will be created in the same directory. By default, the WSDL file is linked to the JWS file from which it was generated, meaning it will be regenerated whenever the JWS file is changed.

To Obtain a WSDL File from WebLogic Server

The WSDL file for a web service is available to any potential client that can reach the web service's URL. To obtain a web service's WSDL file from WebLogic Server:

1. In a browser, browse to the URL of the web service with ?WSDL appended. For example

   http://myServer:7001/MyProject/MyWebService.jws?WSDL
2. Use your browser's File−>SaveAs function to save the WSDL file to your local machine. Note that some browsers will include HTML tags at the top and bottom of the saved file. You must remove these tags to produce a valid WSDL file.

Related Topics

Test View: Overview Tab

How Do I: Use the Java Proxy for a Web Service?

Using the Java proxy for a web service requires different steps depending on whether you use it from within WebLogic Server (as in a JavaServer Page or servlet) or from outside WebLogic Server (as in a standalone Java application).

To Use the Java Proxy from a JSP

1. Open your web service in WebLogic Workshop, then click the Start button to run the service.
2. In Test View, click the Overview tab.
3. Under Web Service Clients, click Java Proxy.
4. When prompted, save the file to disk. Save the file to the WEB−INF/lib directory of the web application from which you wish to use the proxy. The default name of the file is <web service name>.jar; accept the default name unless it conflicts with an existing JAR file in WEB−INF/lib.
5. In your JSP file, add an import of the web service proxy package as shown here:

```
<%@ page import="weblogic.jws.proxies.*" %>
```

6. Create an instance of the proxy class as shown below. The generic proxy class is the name of the web service with "_Impl" appended to the end:

```
<% HelloWorld_Impl proxy = new HelloWorld_Impl(); %>
```

7. The generic proxy returns protocol−specific proxies that in turn contain the actual interface of the web service. The following example assumes you wish to communicate with the web service using SOAP. Use other getHelloWorldXXX methods to get proxies for other protocols:

```
<% HelloWorldSoap soapProxy = proxy.getHelloWorldSoap(); %>
```

8. Call the appropriate methods on the protocol−specific proxy, as in the following example:

```
<%= soapProxy.Hello(); %>
```

Note that you will need to download a new copy of the proxy if you change any of the signatures of the web service methods or callbacks.

To Use the Java Proxy in a Separate Java Application

1. Follow steps 1 through 3 of the procedure above to obtain the Java proxy JAR file.
2. Save the JAR file to a location that is convenient for your Java application.
3. From the Overview tab of Test View, click Proxy Support Jar and save the webserviceclient.jar file to the same location as the proxy JAR file.
4. Use the proxy classes as described in the procedure for JSPs above. The following is an example of using the HelloWorld sample JWS:

```
            import weblogic.jws.proxies.*;
            public class Main
            {
public static void main(String[] args)
{
    try
    {
        HelloWorld_Impl proxy = new HelloWorld_Impl();
        HelloWorldSoap soapProxy = proxy.getHelloWorldSoap();
        System.out.println(soapProxy.Hello());
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
```

```
        }
                    }
```

5. Compile your source, including the JAR files saved in steps 2 and 3 on your class path.
6. Run your Java application, including the two JAR files on your class path.

Related Topics

Test View

How Do I: Control the Protocol and Message Formats a Web Service Supports?

Web services you build with WebLogic Workshop can receive messages using several different protocols and message formats. These can be set on a per−method or per−web service basis. If the protocol and message format is not set on a particular method, that method inherits the protocol and message format used by the web service.

To Set the Protocol and Message Format on a Method

1. In Design View, click the method for which you want to set protocol and message format.
2. In the Properties pane, expand the protocol property.
3. Enable and disable the desired protocols and select the desired message format.

To Set the Protocol and Message Format on a Web Service

1. In Design View, click the web service for which you want to set protocol and message format.
2. In the Properties pane, expand the protocol property.
3. Enable and disable the desired protocols and select the desired message format.

Related Topics

@jws:protocol Tag

How Do I: Tell Developers of Non−WebLogic Workshop Clients How to Participate in Conversations?

For the most part, making your web service accessible to other client applications is as simple as providing a WSDL or Java proxy file. However, if your web service uses the conversation or callback capabilities of WebLogic Workshop, some additional work may be required.

Information about conversations (the conversation instance identifier) and callback URL are transmitted through SOAP headers on the messages that are passed between the client and the web service. If the client is not build with WebLogic Workshop, it will need to set the required SOAP headers manually.

Calling a Start Method

In order to call a start method, the client must propose a conversation ID. The conversation ID is a string with no format restrictions. The conversation ID must be unique on the server hosting the target web service, however. Clients may wish to use a combination of the client machine's hostname, IP address, process ID, clock time or other data that in combination is likely to be unique to that client. If a client proposes a conversation ID that is not unique on the server, a SOAP fault will be returned. The maximum length of the conversation ID is configured in the jws−config.properties Configuration File.

If the client wishes to receive callbacks from the web service, it must send a second SOAP header specifying the URL to which callback messages should be sent.

The general form of the SOAP headers for a start method is as follows:

```
<SOAP:Header>
    <StartHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
        <callbackLocation>Callback_URL</callbackLocation>
        <conversationID>Conversation_ID</conversationID>
    </StartHeader>
</SOAP:Header>
```

Most web service tools include an API through which SOAP headers can be set. Please see your vendor's documentation for more detailed information.

Calling a Continue or Finish Method

When you call a continue or finish method, you pass only the conversation ID. If the specified conversation ID does not identify an existing conversation on the target server, a SOAP fault will be returned.

```
<SOAP:Header>
   <ContinueHeader xmlns="http://openuri.org/2002/04/soap/conversation/">
       <conversationID>Conversation_ID</conversationID>
   </ContinueHeader>
</SOAP:Header>
```

Samples

For an example of a .NET client that participates in a conversation with a WebLogic Workshop web service, and optionally accepts results via callback, see .NET Client Sample.

Related Topics

Maintaining State with Conversations

Overview: Conversations

How Do I: Communicate with a Web Service from a JSP or Servlet?

WebLogic Workshop will generate a Java proxy that you may use to communicate with a WebLogic Workshop web service from any Java code, including a JSP page or servlet. The Java proxy is provided as a JAR file containing the web service−specific proxy classes.

For detailed instructions on how to use a Java proxy , see How Do I: Use the Java Proxy for a Web Service?

Related Topics

How Do I: Use the Java Proxy for a Web Service?

How Do I: Communicate with a Web Service from Another Java Application?

WebLogic Workshop will generate a Java proxy that you may use to communicate with a WebLogic Workshop web service from any Java code. The Java proxy is provided as a JAR file containing the web service−specific proxy classes.

For detailed instructions on how to use a Java proxy from a Java application, see How do I: Use the Java Proxy for a Web Service?

Related Topics

How do I: Use the Java Proxy for a Web Service?

How Do I: Call One Web Service From Another?

You can call another web service from a WebLogic Workshop web service using a Service control. A Service control provides an interface to another web service, allowing the service you are creating to invoke the methods and handle the callbacks of the other service. In addition, through a Service control you can customize how you interact with the web service the control calls, using XML maps to shape XML message, buffers to support asynchrony, and so on. The other web service can be one developed with WebLogic Workshop or any web service for which a WSDL file is available.

To learn more about Service controls, see Service Control: Using Another Web Service.

To learn more about controls, see Controls: Using Resources from a Web Service.

To Add a Service Control to Access to Another Web Service

1. In Design View, from the Add Control drop−down list, select Add Service Control. The Add a Service dialog appears.
2. In Step 1, type the variable name for your Service control. This must be a valid Java identifier.
3. In Step 2, choose the source of your control as follows:

- Click Use a Service control already defined by a CTRL file if there is already a CTRL file for the service you will be calling. You can click Browse to select the file from among those in your project.
- Click Create a Service control from a WSDL if you have a WSDL file representing the service you will be calling. You can click Browse to select the file from among those in your project. You can also enter a URL specifying the location of the WSDL file or click UDDI to browse for a WSDL stored in a UDDI registry.

4. Click Create to finish the process and close the Add a Service Control dialog.

The control you have added is displayed in Design View on the right side of your web service. To learn how to use the control once it has been added to your web service, see Using a Control.

Related Topics

Service Control: Using Another Web Service

Using a Control

How Do I: Use a WebLogic Server Web Service?

How Do I: Use an Apache SOAP Web Service?

How Do I: Use a .NET Web Service?

How Do I: Use a .NET Web Service?

You can use a .NET web service from a WebLogic Workshop web service by using a Service control. A Service control allows your web service to use another web service as though it were a regular Java class, regardless of the language in which the other web service is implemented.

In order to create a Service control for a non−WebLogic Workshop web service, you must first obtain the target web service's WSDL (Web Service Description Language) file. The WSDL file can typically be obtained by accessing the web service's URL with ?WSDL appended.

To Build a Service Control for a .NET Web Service

1. In a web browser, access the web service from the browser using the normal URL with ?WSDL appended to the end.

For instance, if the service was at the URL http://host/service.asmx, you would enter the following URL:

```
http://host/service.asmx?WSDL
```

2. Save the WSDL you receive to a WebLogic Workshop project. Note that some browsers will save the file with HTML tags at the top and bottom that must be removed in a text editor.
3. In WebLogic Workshop, browse to the project and directory to which you saved the WSDL file.
4. Right−click on the WSDL file and select Generate CTRL from WSDL.
5. The resulting CTRL file is a Service control that you may use from any web service in the project. To learn more about Service controls, see Service Control: Using Another Web Service.

To learn how to use the resulting Service control, see Using a Control.

This topic describes how a WebLogic Workshop web service may act as the client to a .NET web service. For an example of a .NET web service acting as a client to a WebLogic Workshop web service, see .NET Client Sample.

Related Topics

Service Control: Using Another Web Service

WSDL Files: Web Service Descriptions

CTRL Files: Implementing Controls

How Do I: Use a WebLogic Server Web Service?

WebLogic Server provides a way to create web services outside of WebLogic Workshop, by using the servicegen facility. You can use such a web service from WebLogic Workshop in exactly the same way you use a web service built with any tool: via a Service control. A Service control can be created from the WSDL file for the target web service.

In order to create a Service control for a non−WebLogic Workshop web service, you must first obtain the target web service's WSDL (Web Service Description Language) file. The WSDL file can typically be obtained by accessing the web service's URL with ?WSDL appended.

To Build a Service Control for a WebLogic Server Web Service

1. In a web browser, access the web service from the browser using the normal URL with ?WSDL appended to the end.

For instance, if the service was at the URL http://host/TraderService, you would enter the following URL:

```
http://host/TraderService?WSDL
```

2. Save the WSDL you receive to a WebLogic Workshop project. Note that some browsers will save the file with HTML tags at the top and bottom that must be removed in a text editor.
3. In WebLogic Workshop, browse to the project and directory to which you saved the WSDL file.
4. Right−click on the WSDL file and select Generate CTRL from WSDL.
5. The resulting CTRL file is a Service control that you may use from any web service in the project. To learn more about Service controls, see Service Control: Using Another Web Service.

To learn how to use the resulting Service control, see Using a Control.

Note: WSDL files generated by WebLogic Server 6.1 use an earlier version of XML Schema which may make them incompatible with WebLogic Workshop 7.0. To learn how to modify a WebLogic Server 6.1 WSDL file so that it may be used by WebLogic Workshop, see How Do I: Create a Service Control from a WSDL File Generated by WebLogic Server 6.1?.

Related Topics

Service Control: Using Another Web Service

How Do I: Generate a CTRL File?

How Do I: Use an Apache SOAP Web Service?

You can build a web service acts as the client of an Apache SOAP web service. You can also call a WebLogic Workshop service from an Apache SOAP web service. This topic describes how to do both.

To Call Your Web Service from an Apache web service

In order to call your web service from an Apache web service, you must change your service to use the SOAP−RPC message format. By default, WebLogic Workshop web services use the Document−literal SOAP message format. To configure your web service to use SOAP−RPC format:

1. In Design View, select your web service.
2. In the Properties pane, expand the protocol property.
3. Click the drop−down list for the soap−style attribute and select "RPC".

   This will make all methods of your web service use the SOAP −RPC message format instead of Document−literal. You may configure the message format on a per−method basis instead of for the service as a whole. To do this, modify the soap−style attribute or the protocol property for each method individually.
4. Obtain a new WSDL file for you web service for use by the Apache SOAP client. You may obtain the WSDL file in any browser by browsing to the URL of the web service with ?WSDL appended to the URL, as follows:

http://host:port/application/service.jws?WSDL

To Call an Apache Web Service From Your Service

You can ordinarily use another web service from your WebLogic Workshop service by creating a Service control from that service's WSDL file. To learn how to use another web service's WSDL file to create a Service control, see How Do I: Call One Web Service from Another?.

Some older Apache WSDL files do not conform to the current WSDL specification. For these files, modifications must be made before the file can be used by WebLogic Workshop. Specifically, the problem is that the children of the root tag <definitions> in the file are sometimes in the wrong order.

To use the Apache SOAP WSDL file, alter it so that the children of the root tag appear in the following order:

1. <types>
2. <message>
3. <portType>
4. <binding>
5. <service>

Note that some of these may appear more than once. If the Apache WSDL file contains these elements in an illegal order, then reordering them by hand may fix the problem.

Related Topics

WSDL Files

How Do I: Use a WebLogic Server Web Service?

How Do I: Use a .NET Web Service?

How Do I: Create a Web Service that Implements a WSDL File?

A Web Services Description Language (WSDL) file is an XML file that contains a description of the interface of a web service. WebLogic Workshop can generate a new JWS file (a web service) that implements the interface described in a WSDL file.

A WSDL file describes only the interface (or public contract) of a web service. After following the instructions below to generate a JWS file from a WSDL file, you must still add code to the JWS file to implement the necessary logic of the web service.

You can generate the JWS file in WebLogic Workshop by following the steps below:

1. Copy the target WSDL to a directory in a WebLogic Workshop project.
2. In WebLogic Workshop, open the project in which you placed the WSDL file.
3. In the Project tree, browse to the directory containing the target WSDL file.
4. Right−click the target WSDL file and select Generate JWS from WSDL.

Related Topics

WSDL Files: Web Service Descriptions

JWS Files: Java Web Services

Structure of a JWS File

How Do I: Create a Service Control from a WSDL File Generated by WebLogic Server 6.1?

WebLogic Server 6.1 generates WSDL files using the version of XML Schema that was valid at the time WebLogic 6.1 was released (referred to as "schema 99"). WebLogic Workshop expects WSDL files that comply with the current XML Schema specification: schema 2001.

To Convert a Schema 99 WSDL File to Schema 2001

Follow the steps below:

1. In all <schema> definitions, any <attribute> elements in <complexType> definitions must appear after the <sequence> tag (if present), not intermixed. For example:

```
<schema>  <complexType>    <attribute>    <attribute>    <sequence>      <element>      <element>
   </sequence>  </complexType> </schema>
```

must be changed to:

```
<schema>  <complexType>    <sequence>      <element>      <element>    </sequence>    <attribute>
   <attribute>  </complexType> </schema>
```

2. Declare the XML namespace tns to refer to the target namespace of the WSDL. So a WSDL that begins with the following:

```
<definitions targetNamespace="java:com.mycompany.webservices.vehicle" ... >
```

should also include a definition of the tns namespace as shown below:

<definitions targetNamespace="java:com.mycompany.webservices.vehicle" ...
xmlns:tns="java:com.mycompany.webservices.vehicle" >

3. Convert all <date> types in the WebLogic Server 6.1 WSDL to <datetime>

4. Update the WSDL file to use the current schema. Change:

```
xmlns="http://www.w3.org/1999/XMLSchema"
```

to:

```
xmlns="http://www.w3.org/2001/XMLSchema"
```

Once you have made these changes to the WSDL file, you should be able to use it to create a Service control in WebLogic Workshop.

Related Topics

How Do I: Use a WebLogic Server Web Service?

Service Control: Using Another Web Service

How Do I: Handle DataSets Returned from .NET Web Services?

DataSets are a data type used by .NET to store information in a format that is similar to a relational database. Each DataSet may contain a number of tables. .NET web services return DataSets by serializing them into

XML.

You have two choices when deciding how to write code to accept the XML−encoded DataSet: you can write code that processes the XML DOM (Document Object Model) Node directly, or you can write an XML map that will process the incoming XML message automatically.

Handling the DOM Node

You might choose to handle the XML DOM node directly if the structure of the DataSet is subject to change or is otherwise unpredictable. If the structure does not change, even though the quantity of data may, you should consider using the XML map approach described in the next section.

If you create a Service control for a .NET web service that returns a DataSet, the corresponding method on the Service control will return an org.w3c.Node. This class is part of the W3C Document Object Model (DOM), which is an API that may be used to manipulate XML documents directly.

To learn more about the DOM, see Document Object Model (DOM).

Using an XML Map

If the structure of the DataSet being returned is constant, you can construct an XML map that will enable WebLogic Server to process the incoming XML automatically, converting it to Java objects of your choice. To do this:

1. Test the .NET service, capturing the returned XML−encoded DataSet.
2. Examine the structure of the XML. Since it represents query returns, it consists of a hierarchy of XML elements that match the structure of the query result.
3. Produce an XML map that matches the structure of the DataSet XML.
4. Apply this XML map as the value of a @return−xml tag on the method you are using to call the .NET service.
5. Make use of the <xm:multiple> attribute to handle repeating structures.
6. WebLogic Server will now extract the data from the incoming XML automatically and transfer it to the Java object the Service control method returns.

Related Topics

How Do I: Communicate with .NET Web Services?

Service Control: Using Another Web Service

Handling and Shaping XML Messages with XML Maps

Why Use XML Maps

<xm:multiple> Attribute

ConsolidateAddress.jws Sample

InputMapMultiple.jws Sample

How Do I: Get a WSDL from a UDDI Registry and Turn It into a Service Control?

WebLogic Workshop provides a way to get a WSDL from a public or private UDDI registry and bring it in as a Service control in your JWS file.

To Take a WSDL from a UDDI Registry and Add it as a Control

1. In Design View, from the Add Control drop−down list, select Add Service Control. The Add Service Control dialog appears.
2. In step 1, enter a name that your web service will use to reference the control to be created from the WSDL. The name must be a valid Java identifier.
3. In step 2 of the dialog, select Create a Service Control from a WSDL, then click UDDI. This button opens a browser and the UDDIExplorer, a web application that allows you to browse both public and private registries for web services stored as WSDL files.
4. Use the Search a Public Registry and Search a Private Registry links to locate a WSDL. There is also a Help link on the web application page if you need assistance.
5. When you find the WSDL file you wish to use as a Service control, copy the URL for the WSDL and return to the Add Service Control dialog.
6. Paste the URL for the WSDL into the File or URL box.
7. Click Create.

A CTRL file will be created in the same directory as the web service to which it was being added, and a reference to that control will be added to your web service. This control can now be used to access the web service represented by the WSDL.

Related Topics

Service Control: Using another Web Service

WSDL Files: Web Service Descriptions

How Do I: Find a Web Service in a UDDI Directory?

Each UDDI directory has a different user interface. The UDDI directory included with WebLogic Server is called the UDDI Directory Explorer. You can access the UDDI Directory Explorer with the following URL:

http://localhost:7001/uddiexplorer/index.jsp

To learn how to use a web service whose WSDL file you find in a UDDI directory, see How Do I: Get a WSDL from a UDDI Registry and Turn It into a Service Control?

Related Topics

Service Control: Using Another Web Service

WSDL Files: Web Service Descriptions

How Do I: Use a Web Service Found in a UDDI Directory?

To learn how to use a web service found in a UDDI directory, see How Do I: Get a WSDL from a UDDI Registry and Turn It into a Service Control?.

Related Topics

Service Control: Using Another Web Service

WSDL Files: Web Service Descriptions

How Do I: Design a Web Service that Involves Long−Running Operations?

WebLogic Workshop introduces the notion of conversations to manage the state of your web service (stored in class member variables) over a long−running sequence of operations and correlate messages between your web service and its clients and controls. To use conversations, you simply set the conversation property on each method or callback, marking it as starting, continuing, or finishing a conversation. When a start method is called, WebLogic Server creates a new record (a conversation instance) for the state of the web service. When continue methods or callbacks are called, the state of the conversation is accessible. After a finish method or callback completes, the conversation state is deleted to release server resources.

To Specify the Conversational Behavior of a Web Service

1. In Design View, select a method you wish to start a conversation.
2. In the Properties pane, expand the conversation property.
3. On the phase attribute, from the drop−down list, select start.
4. Repeat steps 1 through 3 for each method that can start a conversation.
5. Select a method you wish to continue the conversation.
6. In the Properties pane, expand the conversation property.
7. On the phase attribute, from the drop−down list, select continue.
8. Repeat from steps 5 through 7 for each method that can continue a conversation.
9. Select a method you wish to finish the conversation.
10. In the Properties pane, expand the conversation property.
11. On the phase attribute, from the drop−down list, select finish.
12. Repeat steps 9 through 11 for each method that can finish a conversation.

To Specify Conversation Lifetime

Conversations represent server resources and should be released whenever possible. In addition to marking methods of a conversation to control how they affect conversation phase, you can also configure the maximum lifetime or idle time of a conversation. These properties will cause the conversation to be released after a specified period even if (or because) none of the conversational methods have been called.

To set the maximum lifetime of a conversation, regardless of whether or when continue or finish methods are called, set the max−age attribute of the conversation−lifetime property on the web service.

To set the maximum idle time of a conversation (the maximum time that may pass between method invocations), set the max−idle−time attribute of the conversation−lifetime property on the web service.

Related Topics

Maintaining State with Conversations

@jws:conversation−lifetime Tag

How Do I: Return Information to a Client Using a Callback?

You can use a callback to return information to a client asynchronously. When you invoke the callback, your web service sends a message back to the client.

Using callbacks assumes that the client software implements its own code for receiving the callback message you send. If the client does not support receiving callbacks, an alternative is to use polling, as described in Using Polling as an Alternative to Callbacks.

To Add a Callback

1. In Design View, select your web service.
2. From the Add Operation drop−down list, select Add Callback.
3. Double−click the arrow corresponding to the callback. The Edit Maps and Interface dialog appears.
4. Change the name of your callback and its arguments to include the information that you wish to send to the client.
5. Click OK.

To invoke the callback in your web service, you can write code such as the following:

```
callback.name(arg1, arg2);
```

Replace name with the name of your callback, passing it the arguments that match the types you entered in the Edit Maps and Interface dialog.

Related Topics

Edit Maps and Interface Dialog

Using Callbacks to Notify Clients of Events

Using Polling as an Alternative to Callbacks

How Do I: Poll Another Service for Information?

When you want to communicate with a web service that does not support asynchronous callbacks, you will have to poll the web service until the result is available. With polling, you call a function at regular intervals to check if the result is ready.

This is easy to accomplish using a Timer control.

To Add a Timer Control to Your Web Service for Polling

1. In Design View, select your web service.
2. From the Add Control drop−down list, select Add Timer Control.
3. In the dialog that appears, in step 1 of the dialog, enter a name for the variable that will represent your control (for example, you might enter timer). The name you enter must be a valid Java identifier.

4. In step 2 of the dialog, in the timeout−in box, enter the amount of time until the first poll should occur (for example, "0 sec").
5. In step 2 of the dialog, in the repeats−every box, enter the amount of time between polls (for example, "5 sec").

Call the timer's start method when you want to begin polling. Your code to start the timer might look something like this:

```
service.startOperation();
timer.start();
```

Each time the timer goes off, it will call your handler for the "onTimeout" callback. In this handler, you should call the web service to see if the information is available. Your code might look something like this:

```
private void timer_onTimeout(long time)
{
    if (service.isOperationDone())
   {
       timer.stop();
        // process operation result...
    }
}
```

Be sure to call stop method on the timer when you want to stop polling.

Related Topics

Timer Control: Using Timers in Your Web Service

How Do I: Control the Lifetime of a Web Service?

If several messages between your web service and a client are related to each other, they messages can be associated with one another by using conversations. To create a conversation, identify messages from the client as messages that start a conversation, continue a conversation, finish a conversation, or do not participate in a conversation at all.

Methods that start a conversation will create a unique identifier that is returned to the client.

Continue methods and callbacks will use this identifier to associate their message with the information that was saved from the start method. There can be as many continue methods necessary to accumulate the information you web service needs.

Once all of the processing is complete, finish methods will remove the unique identifier from the server's conversation persistence cache.

To Set How a Method Participates in a Conversation

1. In Design View, click the method you identified as starting the conversation.
2. In the Properties pane, expand the conversation property.
3. On the phase property, from the drop−down list, select start.
4. Repeat steps 1 through 3 for each method that can start a conversation.
5. Click the method you identified as continuing the conversation.
6. In the Properties pane, expand the conversation property.
7. On the phase property, from the drop−down list, select continue.
8. Repeat from steps 5 through 7 for each method that can continue a conversation.
9. Click the method you identified as finishing the conversation.
10. In the Properties pane, expand the conversation property.
11. On the phase property, from the drop−down list, select finish.
12. Repeat steps 9 through 11 for each method that can finish a conversation.

A conversation can also be finished at run time by a call to the JwsContext interface's finishConversation() method. This interface was added to your web service when the service was created with a default variable name of context. To end a conversation from the server, add this line in the source view where you would like to end the conversation.

```
context.finishConversation();
```

A conversation can also expire due to the max−age attribute of the conversation−lifetime property. The default duration for a conversation is one day.

Note that conversation termination never occurs during the execution of a web service method. Whether the conversation is finished due to the method being marked as a "finish" method; because JwsContext.finishConversation is called; or whether the conversation lifetime expires, the conversation does not actually finish until immediately after execution of any current method invocation.

To Change the Default Conversation Lifetime

1. In Design View, select your web service.
2. In the Properties pane, expand the conversation−lifetime property.
3. For the max−age attribute, enter the duration you would like to preserve a conversation.

You can also give a conversation a max−idle time, or how long should the server wait between messages.

To Change the max−idle−time Property

1. In the Properties pane, expand the conversation−lifetime property.
2. For the max−idle−time attribute, enter the longest allowed duration between messages. Use the default of 0 if you do not want to use the max−idle time.

If a conversation exists for a duration of more than the max−age, or there are not messages for longer than the max−idle−time, the conversation will be removed from the server's cache.

Related Topics

Managing Conversation Lifetime

@jws:conversation−lifetime Tag

JwsContext.finishConversation() Method

How Do I: Associate Code With the End of a Conversation?

When a conversation ends, the onFinish callback of the JwsContext interface is invoked.. You can write code to be executed at the end of a conversation by adding a handler for this callback to your web service.

To Add a Handler for the onFinish Callback

1. In Source View, from the left−hand drop−down list at the top of the pane, select context.
2. From the right−hand drop−down list at the top of the pane, select onFinish.

This adds a handler called context_onFinish.

3. Add your code to the body of this method.

The argument to onFinish is a boolean value that indicates whether the conversation ended because it timed out (true) or because a finish method was invoked (false).

Related Topics

Managing Conversation Lifetime

How Do I: Avoid Deadlock Situations in a Web Service?

There are many situations in which deadlocks can be created between communicating web services. However, following this rule will keep you out of trouble most of the time:

Never invoke a synchronous method on the caller to a synchronous method.

When a client invokes a synchronous method on a web service, the client will block waiting for the method to complete. If you attempt to call the client (invoke a callback) while the client is waiting for the original method to complete, then each party will waiting for the other. This is deadlock. The same situation could occur if a callback handler attempted to call a synchronous method on it's caller.

If you have a situation where you want to invoke a callback from a method, you can avoid deadlock by making either the method or the callback asynchronous. One simple way to make the method or callback asynchronous is to use message buffering.

To Buffer a Method or Callback

1. In Design View, select the method or callback.
2. In the Properties pane, expand the message−buffer property.
3. From the drop−down list next to the enable attribute, select true.

Related Topics

Using Asynchrony to Enable Long−Running Operations

How Do I: Handle Errors In a Web Service?

Some web service errors cannot be detected at compile time. For example, your code may attempt to call a method on a null reference or it may invoke a Database control that attempts to execute some invalid SQL. In most cases, an error at run time will cause an exception to be thrown. Unless you add code to handle the exception, this will cause a SOAP fault to be sent back to the client. Unhandled exceptions also cause rollback of a web service's implicit transaction.

You may want to override the default exception handling and perform a custom action, such as logging an error so that an administrator will know the problem occurred.

There are two ways that you can write code to handle exceptions. The first way is to add try/catch blocks around the code in your method. For example, to catch all exceptions that are thrown in your method "myMethod," you would write this:

```
/**
 * @jws:operation
 */
public void myMethod()
{
    try
    {
        // [ Normal code for the method. ]
    }
    catch (Exception e)
    {
        // [ Code to handle error cases. ]
    }
}
```

If an exception is thrown at any point during the execution of the normal code of the method, the code in the "catch" block will be executed.

One disadvantage of this is that it only works for "myMethod." So if you want to implement exception handling for all of your methods, you have to add try/catch blocks into every method.

The second way to handle exceptions is to add a handler for the "onException" callback of the JwsContext interface. Unlike try/catch blocks, this code will be executed when an exception is thrown in any of the methods of your web service.

To Add a General Handler for Run−Time Exceptions

1. In Source View, just beneath the Source View tab, click the left−hand class drop−down list, then click context.
2. Click the right−hand drop−down list at the top of the pane, then click onException.

This will insert a handler named context_onException into your web service.

3. Write your custom exception handling code into the body of this method. It will be executed whenever an exception occurs in one of your methods.

For further information on Java exceptions, see the Java language tutorial at java.sun.com/docs/books/tutorial/essential/exceptions/index.html.

Writing to Log Files

WebLogic Workshop has pre−configured log files to which you can write messages. To learn about the pre−configured log files and how to write to them, see workshopLogCfg.xml Configuration File.

Related Topics

Introduction to Java

Default Transactional Behavior in WebLogic Workshop

How Do I: Understand How Transactions Are Managed in a Web Service?

Whenever a method of your web service is invoked, all of the actions performed in that method will be part of one transaction. If the method completes normally, the transaction will commit. However, if an exception occurs, then the transaction will be rolled back.

For a description of WebLogic Workshop's implicit transactions, see Default Transactional Behavior in WebLogic Workshop.

Related Topics

Implicit Transactions in WebLogic Workshop

How Do I: Use an Existing Enterprise Java Bean?

WebLogic Workshop provides the EJB control to enable convenient access to an existing Enterprise Java Bean (EJB) from web services. To use an EJB, the client interfaces must be present in a JAR file in the WEB−INF\lib directory of the appropriate WebLogic Workshop project.

To Use an Existing Enterprise Java Bean

1. In WebLogic Workshop, open the project that will use the EJB.
2. If the WEB−INF\lib folder exists in the project, skip to step 5. If not, proceed to step 3.
3. In the Project tree, right−click the WEB−INF folder and select New Folder.

4. In the Enter a new folder name field, type lib, then click OK.
5. Copy the EJB JAR file from the to the WEB−INF\lib folder.

Once the EJB is included in your project, you can create a control to access to the EJB.

6. In WebLogic Workshop, open a web service that will use the EJB.
7. In Design View, from the Add Control drop−down list, select Add EJB Control. The Add EJB dialog appears.
8. Enter a variable name that your web service will use to reference the control member. It is customary for this name to start with an lower−case character. The name you enter must be a valid Java identifier.
9. Select Create a new EJB control to use with this service.
10. In the New CTRL name box, enter a name for the CTRL file. It is customary for this name to start with an upper−case character. The name you enter must be a valid Java identifier.
11. In the jndi−name field, type the Java Naming and Directory Interface (JNDI) name for your EJB, or click Browse to select one. Once you have selected an EJB by JNDI name, the home and bean interface text boxes should contain the names of the EJB's interfaces.
12. Click Create.

A CTRL file is created in the same directory as the web service to which it is being added, and a reference to that control is added to your web service. This control can now be used to access methods in the EJB. This CTRL file can also be used by other web services.

Related Topics

EJB Control: Using Enterprise Java Beans from a Web Service

How Do I: Exchange Messages with a JMS Queue or Topic from a Web Service?

WebLogic Workshop provides the JMS control to enable convenient access to an existing Java Message Service (JMS) queue or topic from web services.

To Create a JMS Control

1. In Design View, find the web service from which you would like to access a JMS queue or topic.
2. From the associated Add Control drop−down list, select Add JMS Control. The Add JMS Control dialog appears.
3. In step 1, enter a name that your web service will use to reference the control member. It is customary for this name to start with a lower−case character. The name you enter must be a valid Java identifier.
4. Select Create a new JMS control to use with this service.
5. In the New CTRL name field, enter a name for the CTRL file. It is customary for this name to start with an upper−case character. The name you enter must be a valid Java identifier.
6. In the first part of step 3 in the dialog, select Queue or Topic.
7. If this JMS control will allow web services to send or publish messages, in the next part of step 3 corresponding to the send−jndi−name box, click browse to select the JNDI name for the queue or topic you will be sending to.

Java Naming and Directory Interface (JNDI) is a registry for resources such as Enterprise Java Beans, JMS queues, and so on. The provider of the resource is responsible for registering it and advertising the name to people or code who need the resource.

7. If this JMS control will allow web services to receive or subscribe to messages, in the next part of the step 3 corresponding to the receive−jndi−name box, enter the JNDI name for the queue or topic you will be receiving from or subscribing to, or click Browse to select one.

8. In the connection factory box, enter the connection factory that you will be using to connect to the queue or topic, or click Browse to select one.
9. Click Create.

A CTRL file is created in the same directory as the web service to which it was being added, and a reference to that control is added to your web service. This control can now be used to access the specified JMS queue or topic. This CTRL file can also be used by other web services.

Related Topics

JMS Control: Using Java Message Service Queues and Topics from Your Web Service

How Do I: Use Java Code From a Web Service?

Not all the classes you define with WebLogic Workshop need to be web services. You may create a regular Java class that you can use from your web services. If you wish to access a Java class from within a web service you may need to place an import statement at the top of your web service.

To Create a Java file

1. Click the File menu, point to New, then click New Java Class. The Create New File dialog appears.
2. Confirm that Java is selected.
3. In the File name field, type a name for the Java class you are creating.
4. Click OK.

Once you have created the Java file, you may create a Java class that performs whatever task you need. This Java class will not be a web service and the methods of this Java class will not be accessible to users of your web service.

To Use Existing Java Code

If you have existing Java code that you would like to call from your web service code, you merely have to place the Java code in the project. If you place source files (files with the .java extension) in the project, they will be compiled automatically when the web service that references the Java classes is built. You may also place compiled Java class files (files with the .class extension) in the WEB−INF\classes directory of the folder (in an appropriate directory hierarchy); or you may place a JAR file containing the classes in the WEB−INF\lib directory of the project.

Related Topics

Introduction to Java

How Do I: Create a New WebLogic Workshop−enabled WebLogic Server Domain?

This topic describes how to create a new WebLogic Server domain that is configured for building and testing web services with WebLogic Workshop.

To modify an existing domain, see How Do I: WebLogic Workshop−Enable an Existing WebLogic Server Domain?

To Create a New WebLogic Workshop−Enabled Domain

1. Run the Domain Configuration Wizard by selecting Start | BEA WebLogic Platform 7.0 | Domain Configuration Wizard.

2. On the Choose Domain Type and Name page, select WebLogic Workshop as the domain template, and enter a name for the new domain.
3. Select the server type. If you are creating a new domain on your development server, select the Single Server (Standalone Server) option.
4. Specify a directory for the domain.
5. Specify the server name, the listen address if it differs from the server name, the listen port, and the secure listen port.

6. Enter an administrative user name and password and finish the wizard.

To Run WebLogic Workshop in the New Domain

1. Launch WebLogic Workshop.
2. Choose Preferences from the Tools menu.
3. On the Paths tab, change the Config directory option to point to the directory that contains the new domain, which you specified in Step 4 above. For example, if you've created the domain C:\bea\user_projects\mydomain, you would then set the Config directory to C:\bea\user_projects, then choose mydomain from the Domain list. When you modify the Config directory setting, the values in the Domain list will be updated to show the domains available in that directory.
4. Close the Preferences dialog.
5. From the File menu, create a new project in this domain or open the DefaultWebApp project.

Note: The project that is open in WebLogic Workshop must be part of the domain specified in the Config directory setting in order to design and test it. If the project that is open in WebLogic Workshop is not part of the domain that you've specified in the Config directory setting, the server indicator will show that WebLogic Server is disabled. If you try to test a service in the project, you'll get an error message stating that you need to change to a different project.

To Verify that a Domain is WebLogic Workshop−Enabled

1. Run WebLogic Server in the domain you want to test.

2. In your web browser, type http://<server name>:<port number>/<project name>/jwsdir. For example, if you are testing a domain on your local server, and you have configured it to listen to port 7001, and you have a project in that domain named DefaultWebApp, you would type the URL as follows: http://localhost:7001/DefaultWebApp/jwsdir. If the server is running and the domain is WebLogic Workshop−enabled, you'll see the WebLogic Workshop directory for that project. If the server is running but the domain is not WebLogic Workshop−enabled, you'll get a "Page not found" error.

Related Topics

How Do I: WebLogic Workshop−Enable an Existing WebLogic Server Domain?

How Do I: Deploy WebLogic Workshop Web Services to a Production Server?

You can use the following procedures to deploy web services built with WebLogic Workshop to production computers installed with WebLogic Server. This task is divided into two steps to compile your web application as an EAR file and steps to deploy the EAR file on a production server.

To Compile a Web Application as an EAR File

1. Modify the weblogic−jws−config.xml file in your project's WEB−INF directory. The contents of this file should specify the <hostname>, which should match the name of the production machine to which the service will be deployed.   The global <protocol> tag, which is the immediate child of the

<config> tag, specifies the protocol over which the application as whole should be exposed. The <protocol> tag which is the immediate child of the <jws> tag specifies the protocol over which individual web services should be exposed. Valid values for the global and jws−specific <protocol> tags are http or https.
The following example shows the configuration of a web application containing two web services. One web service, HelloWorld, is to be exposed on the http protocol; the other web service, HelloWorldSecure, is to be exposed on the https protocol:

```
<config> <protocol>http</protocol> <hostname>myProductionMachine</hostname>
<http−port>7001</http−port> <https−port>7002</https−port> <jws>
<class−name>HelloWorld</class−name>   <protocol>http</protocol> </jws> <jws>
<class−name>HelloWorldSecure</class−name>   <protocol>https</class−name> </jws> </config>
```

2. Compile your web application as an EAR file. To compile an application as an EAR use the command line tool jwsCompile, supplied by Workshop. For example, suppose you have a project called Hello, which contains one web service called HelloWorld. The following command will compile the project as an EAR file, assuming that you run jwsCompile from the Hello project's project directory C:\bea\weblogic700\samples\workshop\applications\Hello\

```
jwsCompile −a −p . −ear HelloWorld.ear
```

To Deploy the EAR File on a Production Server

1. Confirm that there is no application already deployed on the production machine with the same name as the application from which you generated the EAR file. To confirm that there is no such application, visit the WebLogic Server console on the production machine. Undeploy any applications with the same name as the application you are planning to deploy.
   If you are developing and deploying on the same machine, verify that the EJBs generated while running Workshop in development mode are removed as well. To remove the EJB's, manually edit config.xml (found at BEA_HOME/weblogic700/samples/workshop/) to remove any references to the appropriate EJB's. If the EJB's are already present, deployment will generate an instance already exists exception

2. Start WebLogic Server in production mode. To start WebLogic Server in production mode, run the following
   BEA_HOME\weblogic700\samples\workshop\startWeblogic.cmd production nodebug
3. On the development machine, deploy the EAR file to the production machine by using the WebLogic Server tool weblogic.Deployer. To use the weblogic.Deployer tool make sure that weblogic.jar is in your classpath. For example, the following command deploys the EAR file HelloWorld.ear to the production server myProductionServer:
   C:\>java weblogic.Deployer −adminurl http://myProductionServer:7001 −password installadministrator −source C:\EARS\HelloWorld.ear −targets cgServer −name HelloWorld −activate −upload

Alternatively, you can use the console of the production machine to deploy the EAR file.

1. Start WebLogic Server in production mode. To start WebLogic Server in production mode, run the following BEA_HOME\weblogic700\samples\workshop\startWeblogic.cmd production nodebug

2. In a web browser visit the console page: http://localhost:7001/console
3. When prompted for a password, enter installadministrator.
4. In the directories displayed on the left side of the screen, navigate to Deployments/Applications, and right click on the Applications directory. Select "Configure a new Application...".
5. In Step 1, click the link "upload it through your browser".
6. Click the Browse... button and navigate to the EAR file you wish to deploy, then click Upload.
7. In Step 2, select the application you would like to deploy.
8. In Step 3, select the target servers from the available servers.
9. In Step 4, confirm the name of the application.
10. In Step 5, click "Configure and Deploy".
11. Finally, click "Deploy Application".

How Do I: Move or Copy a Web Service from One Project to Another?

You can use a web service you have created in one project in another WebLogic Workshop project by moving or copying related files.

Moving or copying WebLogic Workshop web service files is generally straightforward. However, be sure to obey the following rules:

1. Create the destination using WebLogic Workshop's File menu. This ensures that the project is properly configured.
2. Do not copy the WEB−INF or EJB subdirectories from one project to another. These folders contain compilation results that are location−specific. If you copy them from one project to another the destination project will not operate properly. To fix the problem, delete these directories and rebuild the web services in the project.
3. If you move a Java file (including JWS, CTRL or JAVA files), the directory hierarchy and package hierarchy must match when you are finished.

Related Topics

WebLogic Workshop Projects

How Do I: Use a Web Service that is in Another Project?

You can create a Service control for a web service in a different project in the same way that you create Service control for any other web service. You must obtain either a WSDL or a CTRL file for the web service you want to call.

To learn more about how to obtain CTRL files and WSDL files, see CTRL Files: Implementing Controls and WSDL Files.

Once you have obtained a CTRL or WSDL file, you must move it into the project from which you will be using it. For more information, see How Do I: Move or Copy a Web Service from One Project to Another?

For a WSDL file you add to the project directory, a Service control may be generated in the normal way.

To learn more about creating and using Service controls, see Creating a New Service Control.

Related Topics

Service Control: Using Another Web Service

How Do I: WebLogic Workshop−Enable an Existing WebLogic Server Domain?

This topic describes how to modify the configuration of an existing WebLogic Server 7.0 domain such that it can host WebLogic Workshop web services.

If you want to create new domain, see How Do I: Create a New WebLogic Workshop−enabled WebLogic Server Domain?

Required Resources

In order for a WebLogic Server 7.0 domain to be able to host WebLogic Workshop web service, the domain must be configured with the resources described in the following sections.

These resources may be configured using the WebLogic Server console or by editing the domain's config.xml file directly. You can learn more about configuring WebLogic Server on e−docs.bea.com at Creating and Configuring WebLogic Server Domains.

JMS Server

A JMS server must be configured to support message buffers, timer controls, JMS controls and JMS as a web service message transport. You can learn how to configure WebLogic JMS servers on e−docs.bea.com at Configuring WebLogic JMS. The name of the JMS server must be specified as the value of the weblogic.jws.InternalJMSServer property in the jws−config.properties file, as described below in WebLogic Server Configuration.

After configuring a JMS server, the resulting entry in config.xml should resemble the following:

```
<JMSServer Name="cgJMSServer" Store="cgJDBCStore" Targets="cgServer">
```

JMS Connection Factory

A JMS connection factory must be configured to provide JMS connections. You can learn how to configure WebLogic JMS connection factories on e−docs.bea.com at Configuring WebLogic JMS. The name of the JMS connection factory must be specified as the value of the weblogic.jws.InternalJMSConnFactory property in the jws−config.properties file, as described below in WebLogic Server Configuration.

After configuring a JMS connection factory, the resulting entry in config.xml should resemble the following:

```
<JMSConnectionFactory JNDIName="weblogic.jws.jms.QueueConnectionFactory" Name="cgQueue" Targets
```

Conversation Data Source and Connection Pool

A JDBC data source must be configured to support persistence of web service conversation state. The data source must be a JDBCTxDataSource (a transacted data source). You can learn how to configure a data source on e−docs.bea.com at Configuring and Managing JDBC Connection Pools, MultiPools, and DataSources Using the Administration Console.

After configuring a data source, the resulting entry in config.xml should resemble the following:

```
<JDBCConnectionPool Name="cgPool"
    DriverName="com.pointbase.jdbc.jdbcUniversalDriver"
    InitialCapacity="5" MaxCapacity="20" CapacityIncrement="1"
    Properties="user=cajun;password=abc" RefreshMinutes="0"
    ShrinkPeriodMinutes="15" ShrinkingEnabled="true"
    SupportsLocalTransaction="true" Targets="cgServer"
    TestConnectionsOnRelease="false"
    TestConnectionsOnReserve="false" URL="jdbc:pointbase:server://localhost/cajun"/>
<JDBCTxDataSource EnableTwoPhaseCommit="true"
    JNDIName="cgDataSource" Name="cgDataSource" PoolName="cgPool" Targets="cgServer"/>
```

While this example uses the Pointbase database, other databases may be used.

JMS Control Data Source

A JDBC data source must be configured to support state persistence by JMS controls. This is typically the same data source configured in the previous section.

WebLogic Server Configuration

The resources define in the previous sections are referred to from the WebLogic Workshop configuration file jws−config.properties. jws−config.properties contains definitions for the following properties:

```
weblogic.jws.InternalJMSServer=<name of JMS server>
weblogic.jws.InternalJMSConnFactory=<JNDI name of JMS connection factory>
weblogic.jws.ConversationDataSource=<JNDI name of Conversation JDBC data source>
weblogic.jws.JMSControlDataSource=<JNDI name of JMS store data JDBC data source>
```

For a complete description of the jws−config.properties file, see jws−config.properties Configuration File.

Related Topics

jws−config.properties Configuration File

How Do I: Create a New WebLogic Workshop−enabled WebLogic Server Domain?

How Do I: Publish a Web Service to a UDDI Directory?

Each UDDI directory has a different user interface. The UDDI directory included with WebLogic Server is called the UDDI Directory Explorer. You can access the UDDI Directory Explorer with the following URL:

http://localhost:7001/uddiexplorer/index.jsp

WebLogic Server is configured with a private registry to which you may publish web services. To publish a web service:

1. In a browser, go to http://localhost:7001/uddiexplorer/index.jsp
2. Select Publish to Private Registry.
3. Log into the UDDI Directory Explorer. The default username and password are installadminstrator.
4. Complete the Service Information.
5. In the WSDL field, type the URL of your web service. This URL directs users who look up your web service to the Overview Page of Test View for your web service, from which they may obtain the WSDL file. For example, the URL of the HelloWorld sample web service is:

   http://hostname:port/samples/HelloWorld.jws
6. Click Add Service.

Related Topics

How Do I: Get a WSDL from a UDDI Registry and Turn It into a Service Control?

How Do I: Configure WebLogic Workshop to Use a Different Database for Internal State?

WebLogic Workshop uses a JDBC data source to store internal state. By default, this data source is defined on the PointBase database that is installed with WebLogic Server. You may wish to configure WebLogic Workshop to use a different database for higher performance or scalability.

If you wish to configure WebLogic Workshop to use a database other than PointBase, you must configure the connection pool and data source you would like WebLogic Workshop to use, then configure WebLogic Workshop to use the new connection pool and data source.

To see how the default connection pool and data source are configured (as an example), see Conversation Data Source in How Do I: WebLogic Workshop−Enable an Existing WebLogic Server Domain?

To see how to configure WebLogic Workshop to use the data source, see WebLogic Server Configuration in How Do I: WebLogic Workshop−Enable an Existing WebLogic Server Domain?

Related Topics

jws−config.properties Configuration File

How Do I: Reset the State of WebLogic Server?

Occasionally while developing web services, you may compile a web service that causes undesirable behavior in the server. WebLogic Workshop provides a tool that may help you clear the state of the server in such situations.

To reset the state of a development WebLogic Server hosting WebLogic Workshop web services:

1. Browse to the URL of the WebLogic Workshop project containing the web service that caused the error, with "/jwsdir" appended to the URL. For example:

   http://localhost:7001/samples/jwsdir
2. Click Clean All on the WebLogic Workshop Directory page that is displayed.

Clean All attempts to undeploy EJBs and other resources that were produced by JWS compilation. In some cases, this is not possible due to the state of the server. If Clean All does not solve the problem, stop and restart WebLogic Server.

Note that the problems Clean All solves do not occur on production servers. Clean All addresses problems that occur due to cyclic compilation of JWS files at development time.

Related Topics

WebLogic Workshop Projects

Samples

WebLogic Workshop includes numerous fully documented samples to help you become familiar with WebLogic Workshop web service design patterns, features and programming techniques. Samples are provided that illustrate web services and web service clients.

Topics Included in this Section

Sample Web Services

Describes a large collection of web service samples. Each sample focuses on a small set of WebLogic Workshop features and techniques.

Widgets Sample Application

Widgets represents a complete business process implemented as web services. The sample application makes heavy use of XML maps to demonstrate loose coupling between web service implementation and message format.

Java Client Samples

Describes three examples of using the Java proxy that WebLogic Workshop provides for each web service. The same web service is used from a JSP page; from a Java console application and from a Java Swing application.

.NET Client Sample

Illustrates how to use a WebLogic Workshop web service from a .NET client, including full participation in conversations and receipt of a callback. A fully developed and documented ASP.NET web service that is a client of a WebLogic Workshop web service is provided.

Related Topics

Building Web Services with WebLogic Workshop

Tutorial: Your First Web Service

Sample Web Services

The following sample web services are provided with WebLogic Workshop. They are located in the samples project, which is the default project when you start WebLogic Workshop for the first time.

Follow the links below for more detailed information on each sample, including instructions for running the sample.

Topics Included in This Section

AccountEJBClient.jws

A web service that demonstrates use of an EJB control (AccountEJBControl.ctrl), which represents the AccountEJB Entity Bean and exposes its business interface to web services. This sample is not related to AccountPublish.jws and AccountSubscribe.jws.

AccountPublish.jws

A web service that demonstrates use of a JMScontrol (AccountPublishJMSControl.ctrl) to publish a message to a JMS topic. This sample is paired with AccountSubscribe.jws, which subscribes to the JMS topic to which AccountPublish.jws publishes. This sample is not related to AccountEJBClient.jws.

AccountSubscribe.jws

A web service that demonstrates use of a JMScontrol (AccountPublishJMSControl.ctrl) to subscribe to a JMS topic. This sample is paired with AccountPublish.jws, which publishes to the JMS topic to which AccountSubscribe.jws subscribes. This sample is not related to AccountEJBClient.jws.

AdvancedTimer.jws

A web service that demonstrates an asynchronous interface to a legacy system (LegacySystem.jws) that does not support asynchrony. It does so by "polling" the legacy system: calling it repeatedly to see if it's done. AdvancedTimer does the polling for the client, invoking a client callback if and when the legacy system responds.

AdvancedTimer uses the @jws:timer timeout= repeats−every= tag.

Buffer.jws

A web service that demonstrates the @jws:message−buffer tag to queue high−traffic requests. Uses a Timer control to delay sending a response back to the client, simulating waiting for a slow back−end service to respond to requests from this web service. The start method is buffered, allowing clients to continue processing immediately after submitting a request.

ConsolidateAddress.jws

A web service that demonstrates use of the ECMAScript language extensions for XML to perform XML mapping using procedural code. An incoming XML document is converted to a different internal Java data structure, then converted back when returned to the client by a subsequent method call. The ECMAScript is in the ConsolidateAddressScript.jsx file. ConsolidateTest.xml is a test document that may be submitted.

ConsolidateAddress uses the <xm:use> XML map tag

Conversation.jws

A web service that demonstrates use of the @jws:conversation tag to control the lifetime of a conversational instance of the web service and provide data persistence.

Conversation.jws implements a synchronous polling interface for the asynchronous HelloWorldAsync web service. A polling interface is necessary if you wish to serve clients that cannot accept asynchronous callbacks due to security arrangements such as firewalls.

Conversations also provide correlation, whereby requests from multiple simultaneous clients are tracked and responses are directed to the right client.

CreditReport.jws

Similar to the Investigate web service developed during Tutorial: Your First Web Service, but more fully developed. Demonstrates use of conversations and Service controls.

CustomerDBClient.jws

A sample that demonstrates use of a database control by managing customer records. CustomerDBClient is a client of CustomerDBControl.ctrl. Together they demonstrate construction of a Database control and use of a Database control by a web service. CustomerDBControl.ctrl demonstrates use of SQL's CREATE, DROP, INSERT, UPDATE, and SELECT statements.

CustomJMSClient.jws

A web service that demonstrates use of a JMS control defined in a CTRL file. CustomJMSClient is a client of the CustomJMSControl.ctrl JMS control.

ExplicitTypes.jws Sample

A web service that demonstrates use of the include−java−types attribute of @jws:paramater−xml and @jws:return−xml to explicitly specify types used in ambiguous situations.

HelloWorld.jws

A very simple web service with a single synchronous method. Illustrates a WebLogic Workshop web service in its simplest form.

HelloWorldAsync.jws

A simple asynchronous web service that illustrates the use of a callback. Uses a Timer control to provide a delay that simulates waiting for a slow back−end service to do some work, then notifies the client of the result via a callback. HelloWorldAsync is used by the Conversation.jws sample, but is also a standalone web service.

InputMapMultiple.jws

Demonstrates the use of an @jws:parameter−xml tag defining an XML map to convert XML input to Java objects, including an indeterminate length list of elements using the <xm:multiple> attribute.

Investigate.jws

This sample represents the state of the Investigate web service after completion of Step 8: Access the Bank Web Service of Tutorial: Your First Web Service.

LuckyNumberDBClient.jws

A sample that uses a database to store players and their lucky numbers. LuckyNumberDBClient uses the LuckyNumber.jws web service to generate random numbers in the range [1,20] inclusive, then checks for winners in a database and returns the number drawn and the list of winners, if any. The database is managed using the LuckyNumberDBControl.ctrl Database control. LuckyNumberDBControl.ctrl creates and manages a PLAYERS table in the database and implements queries against it. Demonstrates using SQL's CREATE , INSERT, SELECT and DROP statements in a database control.

OutputMap.jws

Demonstrates use of the @jws:return−xml tag and implicit <xm:value> tags to convert Java objects to elements in an XML document that is returned to the web service's caller.

OutputScriptMap.jws

A web service that demonstrates use of the ECMAScript language extensions for XML to perform XML mapping using procedural code. A Java object returned by a web service method is converted to a different XML structure in the outgoing message. The ECMAScript is in the PersonScript.jsx file.

QuoteClient.jws

Demonstrates the ability to modify a Service control's CTRL file to change the Java signature of one or more methods while still adhering to the public contract of the called web service. QuoteServiceControl.ctrl was originally generated from the QuoteService.jws web service. But one method in QuoteServiceControl.ctrl has had a parameter removed and replaced with a hard−coded value in the accompanying XML map. Thus, the outgoing XML message is the same, but the Java signature visible to clients of the Service control is simplified.

ServiceFactoryClient.jws

A web service that demonstrates use of a control factory. A control factory provides a way to dynamically manage a collection of instances of a control from a web service.

SimpleJMS.jws

A web service that demonstrates use of a JMS control declared locally in a web service's JWS file.

SimpleMap.jws

A sample showing simple use of a @jws:parameter−xml XML map to convert XML input to Java objects. Uses curly brace "{}" notation as implicit <xm:value> tags.

SimpleTimer.jws

A simple web service that demonstrates use of the Timer control. Uses the @jws:timer timeout= tag.

TraderEJBClient.jws

A web service that demonstrates use of the EJB control TraderEJBControl.ctrl, which represents the TraderEJB Stateless Session Bean and exposes its business interface to web services.

Related Topics

Tutorial: Your First Web Service

AccountEJBClient.jws Sample

A web service that demonstrates use of an EJB control AccountEJBControl.ctrl, which represents the AccountEJB Entity Bean and exposes its business interface to web services.

Concepts Demonstrated by this Sample

- Use of an EJB control

Location of Sample Files

This sample is located in the ejbControl folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\ejbControl\AccountEJBClient.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.

   ♦ On Microsoft Windows systems, from the Start menu navigate to:
   BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.

   ♦ On Linux or Solaris systems, run:
   BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/ejbControl/AccountEJBClient.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test Form tab of Test View, if necessary.
4. Create one or more accounts by entering values for key, openingBalance and type and invoking the createNewAccount method.
5. Click the Message Log title to return to the methods page. Experiment with the other methods in the interface.
6. Examine the AccountEJBClient.jws and AccountEJBControl.ctrl files to explore the relationship between the control and its client. The AccountEJBControl.ctrl file was created using the Add EJB Control dialog.
7. Select log entries in the Message Log to see the message traffic involved in each interaction.

Related Topics

Controls: Using Resources from a Web Service

EJB Control: Using Enterprise Java Beans from a Web Service

TraderEJBClient.jws Sample

Test View

AccountPublish.jws Sample

A web service that demonstrates use of a JMS control to publish messages to a JMS topic. AccountSubscribe.jws is a companion to this sample.

Concepts Demonstrated by this Sample

- Use of a JMS control with a topic
- Use of custom JMS control methods
- Use of JMS message properties

Location of Sample Files

This sample is located in the jms folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\jms\AccountPublish.jws

How to Run the Sample

The AccountPublish.jws and AccountSubscribe.jws samples work together. The instructions below describe how to use both services:

1. Start WebLogic Server in the appropriate domain.

    ♦ On Microsoft Windows systems, From the Start menu navigate to:
    BEA WebLogic E−business Platform−>BEA WebLogic Platform 7.0−>WebLogic Workshop Samples−>Start Samples Server.

    ♦ On Linux or Solaris systems, run:
    BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. In a browser (not from the WebLogic Workshop visual development environment), navigate to http://localhost:7001/samples/jms/AccountSubscribe.jws. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run AccountSubscribe.jws.
3. Navigate to the Test Formtab of Test View, if necessary.
4. Invoke the startListening method. The AccountSubscribe.jws web service is now listening for messages that are published to a JMS topic named jms.AccountUpdate.
5. Launch the AccountPublish.jws service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/jms/AccountPublish.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run AccountPublish.jws.
6. Navigate to the Test Form tab of Test View, if necessary.
7. Enter a string value for accountID and numeric value for amount and invoke the deposit method. At this point a message is published to the jms.AccountUpdate JMS topic.
8. In the browser that is testing AccountSubscribe.jws, select "Refresh".
9. You should see that the accountUpdateReceived callback has been sent to the client.
10. Select the accountUpdateReceived log entry to see the payload of the callback. It should contain the same information you entered for the deposit method in Step 7.

The message containing the account transaction was published to the topic by the AccountPublishJMSControl.ctrl JMS control used by AccountPublish.jws. The JMS server then sent the message to all active subscribers. Since AccountSubscribe.jws is subscribed to the topic via the AccountSubscribeJMSControl.ctrl JMS control, it receives the message. If you examine the two CTRL files, you will see that the information you entered was encoded in both the message properties and the message body using the @jws:jms−property and @jws:jms−header properties of the JMS controls. When using JMS messaging, the senders and receivers of messages must agree on the message format at design time.

Related Topics

Controls: Using Resources from a Web Service

JMS Control: Using Java Message Service Queues and Topics from Your Web Service

Maintaining State with Conversations

AccountSubscribe.jws Sample

CustomJMSClient.jws Sample

SimpleJMS.jws Sample

Test View

AccountSubscribe.jws Sample

A web service that demonstrates use of a JMS control to subscribe to a JMS topic. AccountPublish.jws is a companion to this sample.

Concepts Demonstrated by this Sample

- Use of a JMS control with a topic
- Use of a custom JMS callback
- Use of JMS message properties

Location of Sample Files

This sample is located in the jms folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\jms\AccountSubscribe.jws

How to Run the Sample

The AccountPublish.jws and AccountSubscribe.jws samples work together. The instructions below describe how to use both services:

1. Start WebLogic Server in the appropriate domain.

   ♦ On Microsoft Windows systems, From the Start menu navigate to:
   BEA WebLogic E−business Platform−>BEA WebLogic Platform 7.0−>WebLogic Workshop Samples−>Start Samples Server.

   ♦ On Linux or Solaris systems, run:
   BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. In a browser (not from the WebLogic Workshop visual development environment), navigate to http://localhost:7001/samples/jms/AccountSubscribe.jws. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run AccountSubscribe.jws.
3. Navigate to the Test Formtab of Test View, if necessary.
4. Invoke the startListening method. The AccountSubscribe.jws web service is now listening for messages that are published to a JMS topic named jms.AccountUpdate.
5. Launch the AccountPublish.jws service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/jms/AccountPublish.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run AccountPublish.jws.
6. Navigate to the Test Form tab of Test View, if necessary.

7. Enter a string value for accountID and numeric value for amount and invoke the deposit method. At this point a message is published to the jms.AccountUpdate JMS topic.
8. In the browser that is testing AccountSubscribe.jws, select "Refresh".
9. You should see that the accountUpdateReceived callback has been sent to the client.
10. Select the accountUpdateReceived log entry to see the payload of the callback. It should contain the same information you entered for the deposit method in Step 7.

The message containing the account transaction was published to the topic by the AccountPublishJMSControl.ctrl JMS control used by AccountPublish.jws. The JMS server then sent the message to all active subscribers. Since AccountSubscribe.jws is subscribed to the topic via the AccountSubscribeJMSControl.ctrl JMS control, it receives the message. If you examine the two CTRL files, you will see that the information you entered was encoded in both the message properties and the message body using the @jws:jms−property and @jws:jms−header properties of the JMS controls. When using JMS messaging, the senders and receivers of messages must agree on the message format at design time.

Related Topics

JMS Control: Using Java Message Service Queues and Topics from Your Web Service

AccountPublish.jws Sample

CustomJMSClient.jws Sample

SimpleJMS.jws Sample

Test View

AdvancedTimer.jws Sample

A web service that demonstrates an asynchronous interface to a simulated legacy system (LegacySystem.jws) that does not support asynchrony. It does so by "polling" the legacy system: calling it at intervals to see if it is finished. AdvancedTimer does the polling for the client, invoking a client callback if and when the legacy system responds.

Concepts Demonstrated by this Sample

- Use of a Timer control
- Use of a Service control
- Declaration and use of a client callback
- Use of the JwsContext interface
- Use of Conversations
- Polling

Location of Sample Files

This sample is located in the timer folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\timer\AdvancedTimer.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.
   - ♦ On Microsoft Windows systems, from the Start menu navigate to:
   BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.
   - ♦ On Linux or Solaris systems, run:
   BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/timer/AdvancedTimer.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test Form tab of Test View, if necessary.
4. Invoke the start method to create a new conversational instance and invoke the operation on the simulated legacy system.
5. After the legacy system completes it's operation, AdvancedTimer will invoke the onDone callback on the client. If the legacy system does not complete in 15 seconds, AdvancedTimer will invoke the onDone callback anyway, but with a failure indication.
6. Refresh the browser periodically until the callback.onDone callback entry appears in the Message Log.
7. Select log entries in the Message Log to see the message traffic involved in each interaction.

Related Topics

Controls: Using Resources from a Web Service

Timer Control: Using Timers in Your Web Service

Service Control: Using Another Web Service

Maintaining State with Conversations

Using Callbacks to Notify Clients of Events

Using Polling as an Alternative to Callbacks

JwsContext Interface

SimpleTimer.jws Sample

Test View

Buffer.jws Sample

A web service that demonstrates the @jws:message−buffer tag to queue high−traffic requests. Uses a Timer control to delay sending a response back to the client, simulating waiting for a slow back−end service to respond to requests from this web service. The start method is buffered, allowing clients to continue processing immediately after submitting a request.

Concepts Demonstrated by this Sample

- Use of a Timer control
- Declaration and use of a client callback
- Use of the JwsContext interface
- Use of Conversations

Location of Sample Files

This sample is located in the async folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\async\Buffer.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.

   ♦ On Microsoft Windows systems, from the Start menu navigate to:
   BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.

   ♦ On Linux or Solaris systems, run:
   BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/async/Buffer.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test Form tab of Test View, if necessary.
4. Invoke the startBufferAsync method to create a new conversational instance and invoke the operation on the simulated legacy system.
5. After the artificial delay, Buffer will invoke the BufferResult callback.
6. Refresh the browser periodically until the callback.BufferResult entry appears in the Message Log.
7. Select log entries in the Message Log to see the message traffic involved in each interaction.

Related Topics

Controls: Using Resources from a Web Service

Timer Control: Using Timers in Your Web Service

Maintaining State with Conversations

JwsContext Interface

Conversation.jws Sample

Test View

ConsolidateAddress.jws Sample

A web service that demonstrates use of the ECMAScript language extensions for XML to perform XML mapping using procedural code. An incoming XML document is converted to a different internal Java data structure, then converted back when returned to the client by a subsequent method call. The ECMAScript is in the ConsolidateAddressScript.jsx file. ConsolidateTest.xml is a test document that may be submitted.

Concepts Demonstrated by this Sample

- Use of the JwsContext interface
- Use of Conversations
- XML maps
- XML maps with ECMAScript

Location of Sample Files

This sample is located in the xmlmap folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\xmlmap\ConsolidateAddress.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.

   ♦ On Microsoft Windows systems, from the Start menu navigate to:
   BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.

   ♦ On Linux or Solaris systems, run:
   BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/xmlmap/ConsolidateAddress.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test Form tab of Test View, if necessary.
4. Open ConsolidateTest.xml in WebLogic Workshop or a text editor. Copy the contents and paste them into the text box for the setEmployees method between the <employees> and </employees> XML tags.
5. Invoke setEmployees. The XML document submitted is processed by the XML parameter map on the setEmployees method. The paramater map in turn invokes the ECMAScript function ConsolidateAddressScriptFromXML via the <xm:use> tag in the parameter map. The ConsolidateAddressScriptFromXML processes the incoming XML document and transfers data from it to the Java types that are native to the web service.
6. Click on the conversation ID (the large number at the top each conversation in the Message Log) to access that conversation's continue and finish methods.
7. Invoke getEmployeesNatural. This method returns the internal data structure directly, without an XML map. WebLogic Workshop provides a natural mapping that is used by default when compound

objects require conversion to XML. The response to the method invocation is the natural mapping of the internal Java data structure in which the data is stored. Notice that the internal structure is different from the structure of the document you passed in to setEmployees.
8. Click on the conversation ID again to return to the conversation's continue and finish methods.
9. Invoke getEmployees. This method, like setEmployees, uses an XML map with ECMAScript to convert the Java objects returned by the method to XML. The ECMAScript method ConsolidateAddressScriptToXML is used in this case. It converts the Java structure back into the format that the input document had.
10. Examine ConsolidateAddressScript.ps and the relationship between the ECMAScript it contains and the methods in ConsolidateAddress.jws.

Related Topics

Maintaining State with Conversations

JwsContext Interface

Handling and Shaping XML Messages with XML Maps

Handling XML with ECMAScript Extensions

<xm:use> Tag

@jws:parameter−xml Tag

@jws:return−xml Tag

OutputScriptMap.jws Sample

Test View

Conversation.jws Sample

A web service that demonstrates use of the @jws:conversation tag to control the lifecycle of a conversational instance of the web service and provide data persistence.

Conversation.jws implements a synchronous polling interface for the asynchronous HelloWorldAsync web service. A polling interface is necessary if you wish to serve clients that cannot accept asynchronous callbacks due to security arrangements such as firewalls.

Conversations also provide correlation, whereby requests from multiple simultaneous clients are tracked and responses are directed to the right client.

Concepts Demonstrated by this Sample

- Use of a Service control
- Use of the JwsContext interface
- Use of Conversations
- Polling

Location of Sample Files

This sample is located in the async folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\async\Conversation.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.

   ♦ On Microsoft Windows systems, from the Start menu navigate to:
   BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.

   ♦ On Linux or Solaris systems, run:
   BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/async/Conversation.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test Form tab of Test View, if necessary.
4. Invoke the startRequest method to create a new conversational instance and invoke the operation on the simulated legacy system.
5. Click on the conversation ID (the large number at the top of each section in the Message Log) to access that conversation's continue and finish methods.
6. Notice the call from Conversation.jws to helloAsync.HelloAsync. helloAsync is an instance of a Service control and HelloAsync is one of its methods.
7. Click on the conversation ID again to access the conversation's continue and finish methods.
8. Invoke the getRequestStatus method to see if the request is complete. If you do this before the helloAsync_onHelloResult callback arrives from the Service control, you will get a response telling you the back end system hasn't yet replied.
9. If you invoke getRequestStatus after the helloAsync_onHelloResult callback has arrives, getRequestStatus will return the result.
10. Click on the conversation ID again to access the conversation's continue and finish methods.
11. Invoke terminateRequest to finish the conversation. This releases the resources associated with this conversational instance of the web service.
12. Select log entries in the Message Log to see the message traffic involved in each interaction.

Related Topics

Controls: Using Resources from a Web Service

Service Control: Using Another Web Service

Maintaining State with Conversations

Using Polling as an Alternative to Callbacks

JwsContext Interface

Buffer.jws Sample

Test View

CreditReport.jws Sample

Similar to the Investigate web service developed during Tutorial: Your First Web Service, but more fully developed. Demonstrates use of conversations and Service controls.

Concepts Demonstrated by this Sample

- Use of a Timer control
- Use of a Service control
- Declaration and use of a client callback
- Use of the JwsContext interface
- Use of Conversations
- Use of XML maps
- Polling

Location of Sample Files

This sample is located in the creditreport folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\creditreport\CreditReport.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.

   ♦ On Microsoft Windows systems, from the Start menu navigate to:
   BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.

   ♦ On Linux or Solaris systems, run:
   BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/creditreport/CreditReport.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test Form tab of Test View, if necessary.
4. Enter any numeric value for ssn and invoke the requestReport method.
5. If you click Refresh, log entries for calls to the two external services will appear, to bank.startCustomerAnalysis and irs.requestTaxReport.
6. After 10 seconds, bank will report back by sending the onDeliverAnalysis callback. The partial results will be forwarded to the client with an invocation of CreditReport's onProgressNotify callback. You must continually click Refresh to see method invocations on controls and callbacks because there is no way to push these events to a browser.

7. At any time, you may navigate to the continue methods for CreditReport by clicking on the conversation ID in the Message Log.
8. After 20 seconds, the irs service control will respond. Another onProgressNotify callback is sent to the client, then finally a onReportDone callback ends the conversation.
9. At any time before the conversation ends you may invoke getCurrentStatus.
10. At any time before the conversation ends you may invoke cancelReport, which will in turn invoke cancel methods on the external services if they are still pending.
11. Select log entries in the log to see the message traffic involved in each interaction.

Related Topics

Controls: Using Resources from a Web Service

Timer Control: Using Timers in Your Web Service

Service Control: Using Another Web Service

Maintaining State with Conversations

Using Polling as an Alternative to Callbacks

JwsContext Interface

CustomerDBClient.jws Sample

A sample that demonstrates use of a database control by managing customer records. CustomerDBClient is a client of CustomerDBControl.ctrl. Together they demonstrate construction of a Database control and use of a Database control by a web service. CustomerDBControl.ctrl demonstrates use of SQL's CREATE, DROP, INSERT, UPDATE, and SELECT statements.

Concepts Demonstrated by this Sample

- Use of a Database control

Location of Sample Files

This sample is located in the database folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\database\CustomerDBClient.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.

    ♦ On Microsoft Windows systems, from the Start menu navigate to:
    BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.

    - On Linux or Solaris systems, run:
  BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/database/CustomerDBClient.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test Form tab of Test View, if necessary.
4. Invoke the createCustomerTable method to create the database table and populate it with test data.
5. Use other methods in the interface to query the database in various ways. Until you insert additional records, valid customer IDs are 1, 2 or 3.
6. Select log entries in the Message Log to see the message traffic involved in each interaction.
7. Examine the source code for CustomerDBClient.jws and CustomerDBControl.ctrl to see how the CTRL file defines database operations and method shape and the web service uses the methods and data structures provided by the Database control.

Related Topics

LuckyNumberDBClient.jws Sample

Controls: Using Resources from a Web Service

Database Control Design Issues

Test View

CustomJMSClient.jws Sample

A web service that demonstrates use of a JMS control defined in a CTRL file. CustomJMSClient is a client of the CustomJMSControl.ctrl JMS control.

Concepts Demonstrated by this Sample

- Use of a JMS control with a queue
- Declaration and use of a client callback
- Use of Conversations

Location of Sample Files

This sample is located in the jms folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\jms\CustomJMSClient.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.

    - On Microsoft Windows systems, from the Start menu navigate to:

BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.

  - ♦ On Linux or Solaris systems, run: BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh

2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/jms/CustomJMSClient.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test Form tab of Test View, if necessary.
4. Enter values for firstname and lastname and invoke the sendPerson method. The values are packaged in a message and sent to a JMS queue via the JMS control myCustomQ.
5. Since the JMS control in this simple example is configured to listen to the same queue it sends to, the message immediately arrives and is forwarded to the web service via the onResponse callback. Refresh the browser to see the callback entry appear in the Message Log.
6. Select log entries in the Message Log to see the message traffic involved in each interaction.

Related Topics

SimpleJMS.jws Sample

Controls: Using Resources from a Web Service

JMS Control: Using Java Message Service Queues and Topics from Your Web Service

Maintaining State with Conversations

Test View

ExplicitTypes.jws Sample

Demonstrates use of the include−java−types attribute of the @jws:parameter−xml and @jws:return−xml tag to enable correct conversion of Java types contained in Collections.

Concepts Demonstrated by this Sample

- XML maps
- Java<−>XML conversion of Collections
- The include−java−types attribute of @jws:parameter−map and @jws:return−map

Location of Sample Files

This sample is located in the xmlmap folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\xmlmap\ExplicitTypes.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.

   ♦ On Microsoft Windows systems, from the Start menu navigate to:
   BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.

   ♦ On Linux or Solaris systems, run:
   BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/xmlmap/ExplicitTypes.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test XML tab of Test View. This sample expects complex types that cannot be encoded in a HTTP−GET request, so the Test Form page cannot be used to invoke the in method.
4. Invoke the out method. The out method demonstrates use of the include−java−types to control serialization of objects of otherwise unknown type. The object being serialized is an ArrayList, which is declared to contain Objects. Without help, the serialization code would not know how to serialize a generic Object. The include−java−types attribute provides a list of types being serialized, allowing the serialization code to recognize and properly convert the objects. In this case, the ArrayList being serialized contains an array of int and an array of String as its two elements.
5. Notice that the message returned contains <int> and <String> elements that represent the contents of the arrays.
6. From the out method's response, copy the XML between (but not including) the <outResult> tags to the clipboard.
7. Select "Test operations" to return to the page containing all of the web service's methods.
8. Paste the contents of the clipboard into the text area for the in method as the contents of the <arrayList> tag. The <in> and <arrayList> tags should remain − you should be replacing everything between the <arrayList> and </arrayList> tags.
9. Invoke the in method. The response is a formatted string that verifies the input data.

Related Topics

Handling and Shaping XML Messages with XML Maps

@jws:parameter−xml Tag

@jws:return−xml Tag

Test View

HelloWorld.jws Sample

A very simple web service with a single synchronous method. Illustrates a WebLogic Workshop web service in its simplest form.

Concepts Demonstrated by this Sample

- Basic WebLogic Workshop web service structure

Location of Sample Files

This sample is located in the root folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\HelloWorld.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.
   - On Microsoft Windows systems, from the Start menu navigate to:
   BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.
   - On Linux or Solaris systems, run:
   BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/HelloWorld.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test Form tab of Test View, if necessary.
4. Invoke the Hello method. The return value is the string "Hello, World!", of course.
5. Select log entries in the Message Log to see the message traffic involved in each interaction.

Related Topics

Development with WebLogic Workshop

Structure of a JWS File

@jws:operation Tag

Test View

HelloWorldAsync.jws Sample

A simple asynchronous web service that illustrates the use of a callback. Uses a Timer control to provide a delay that simulates waiting for a slow back−end service to do some work, then notifies the client of the result via a callback. HelloWorldAsync.jws is used by the Conversation.jws sample, but is also a standalone web service.

Concepts Demonstrated by this Sample

- Use of a Timer control
- Declaration and use of a client callback
- Use of Conversations

Location of Sample Files

This sample is located in the async folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\async\HelloWorldAsync.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.

   ♦ On Microsoft Windows systems, from the Start menu navigate to:
   BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.

   ♦ On Linux or Solaris systems, run:
   BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/async/HelloWorld.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test Form tab of Test View, if necessary.
4. Invoke the HelloAsync method to create a new conversational instance and invoke the simulated legacy system operation.
5. After 5 seconds, HelloWorldAsync will invoke the onHelloResult callback on the client.
6. Refresh the browser periodically until the callback.onHelloResult callback entry appears in the Message Log.
7. Select log entries in the Message Log to see the message traffic involved in each interaction.

Related Topics

Controls: Using Resources from a Web Service

Timer Control: Using Timers in Your Web Service

Maintaining State with Conversations

Test View

InputMapMultiple.jws Sample

Demonstrates the use of a @jws:parameter−xml tag defining an XML map to convert XML input to Java objects, including an indeterminate length list of elements using the <xm:multiple> attribute.

Concepts Demonstrated by this Sample

- XML maps
- Specification of repeating elements in an XML<−>Java conversion using the <xm:multiple> attribute

Location of Sample Files

This sample is located in the xmlmap folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\xmlmap\InputMapMultiple.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.

   ♦ On Microsoft Windows systems, from the Start menu navigate to:
   BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.

   ♦ On Linux or Solaris systems, run:
   BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/xmlmap/InputMapMultiple.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test Form tab of Test View, if necessary.
4. The single method of the web service, getTotalPrice, specifies an XML map for incoming data. The Test Form displays a default map conforming to the expected schema.
5. Enter values for each of the XML tag values that start with Value_ (replace the entire text string with a new value).
6. Invoke the getTotalPrice method.
7. getTotalPrice extracts the data from the XML map into Java parameters to the method, computes the total price of the submitted order, then returns a single floating point value.
8. Note that there are multiple sibling <item> elements in the expected schema. The <xm:multiple> attribute specified in getTotalPrice's XML map allows it to accept an indeterminate number of sibling <item> elements as input.

Related Topics

OutputScriptMap.jws Sample

Handling and Shaping XML Messages with XML Maps

<xm:multiple> Attribute

@jws:parameter−xml Tag

@jws:return−xml Tag

Test View

This sample represents the state of the Investigate web service after completion of Step 8: Client Application of Tutorial: Your First Web Service.

Note: if you are planning on using the tutorial, you should not run the Investigate.jws service before you begin building with the tutorial.

Concepts Demonstrated by this Sample

- Use of a database control
- Use of a Service control
- Use of a JMS control
- Use of a EJB control
- Use of a timer control
- Declaration and use of a client callback
- Use of Conversations

Location of Sample Files

This sample is located in the tutorials folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\tutorials\Investigate.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.

   ♦ On Microsoft Windows systems, from the Start menu navigate to:
   BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.

   ♦ On Linux or Solaris systems, run:
   BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/tutorials/Investigate.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test Form tab of Test View, if necessary.
4. Enter a value for the first taxID and invoke the requestCreditReportAsynch method. A message is returned asynchronously (through a callback to the client). For taxID, you should enter one of the following 9 digit numbers supported by the database: 123456789, 111111111, 222222222, 333333333, 444444444, 555555555.
5. The Investigate service invokes the Bankruptcies database through a database control. The database tells Investigate whether an applicant is bankrupt.
6. The Investigate service then invokes the CreditCardReport service via a service control. The CreditCardReport service responds through the callback creditCardDataResult. The CreditCardReport service provides a report on an applicant's credit cards.
7. Next Investigate invokes the JMS through the CreditScoreControl control. The JMS responds with a callback. The JMS provides a credit score based on the applicant data retrieved from the database and the credit card report.

8. Next Investigate invokes a EJB, which gives a final credit evaluation for the applicant.
9. Finally, the Investigate service issues a callback, onCreditReportDone, to the client.
10. Select log entries in the Message Log to see the message traffic involved in each interaction.

Related Topics

Controls: Using Resources from a Web Service

Service Control: Using Another Web Service

Maintaining State with Conversations

Using Callbacks to Notify Clients of Events

Asynchronous Web Services

Test View

LuckyNumberDBClient.jws Sample

A sample that uses a database to store players and their lucky numbers. LuckyNumberDBClient uses the LuckyNumber.jws web service to generate random numbers in the range [1,20] inclusive, then checks for winners in a database and returns the number drawn and the list of winners, if any. The database is managed using the LuckyNumberDBControl.ctrl Database control. LuckyNumberDBControl.ctrl creates and manages a PLAYERS table in the database and implements queries against it. Demonstrates using SQL's CREATE , INSERT, SELECT and DROP statements in a database control.

Concepts Demonstrated by this Sample

- Use of a Database control

Location of Sample Files

This sample is located in the database folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\database\LuckyNumberDBClient.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.

   ♦ On Microsoft Windows systems, from the Start menu navigate to:
   BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.

   ♦ On Linux or Solaris systems, run:
   BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh

2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/database/LuckyNumberDBClient.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test Form tab of Test View, if necessary.
4. Invoke the start method to create the database table and populate it with test data.
5. Click on the conversation ID (the large number in the Message Log) to access the continue and finish methods for that conversation.
6. Invoke the drawNumber method. The web service invokes the getLuckyNumber method of the LuckyNumber web service to obtain a random number. LuckyNumberDBClient then queries the database, via the LuckyNumberDBControl Database control, to see if there are any players holding that number.
7. Any winners found are returned in a string, as the return value of drawNumber.
8. Test View always show the details of a Message Log entry when it arrives. Since the invocation of getLuckyNumber occurred after the invocation of drawNumber, Test View is now displaying the details of the getLuckyNumber invocation. Select the drawNumber entry in the Message Log to see its formatted return string.
9. Select log entries in the Message Log to see the message traffic involved in each interaction.
10. Examine the source code for LuckyNumberDBClient.jws and LuckyNumberDBControl.ctrl to see how the CTRL file defines database operations and method shape and the web service uses the methods and data structures provided by the Database control.

Related Topics

CustomerDBClient.jws Sample

Controls: Using Resources from a Web Service

Database Control: Using a Database from Your Web Service

@jws:sql Tag

Test View

OutputMap.jws Sample

Demonstrates use of the @jws:return−xml tag and implicit <xm:value> tags to convert Java objects to elements in an XML document that is returned to the web service's caller.

Concepts Demonstrated by this Sample

- XML maps

Location of Sample Files

This sample is located in the xmlmap folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\xmlmap\OutputMap.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.

   ♦ On Microsoft Windows systems, from the Start menu navigate to:
   BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.

   ♦ On Linux or Solaris systems, run:
   BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/xmlmap/OutputMap.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test Form tab of Test View, if necessary.
4. Enter values for fname and lname and invoke the Hello method.
5. The Hello method receives the values using the default mapping (no custom XML map), but returns its result using an XML map specified with the @jws:return−xml tag.
6. The XML map formats the fields of the Person object into the schema specified in the map, then returns that XML to the client.

Related Topics

InputMapMultiple.jws Sample

Handling and Shaping XML Messages with XML Maps

@jws:return−xml Tag

Test View

OutputScriptMap.jws Sample

A web service that demonstrates use of the ECMAScript language extensions for XML to perform XML mapping using procedural code. A Java object returned by a web service method is converted to a different XML structure in the outgoing message. The ECMAScript is in the PersonScript.jsx file.

Concepts Demonstrated by this Sample

- XML maps
- XML maps with ECMAScript

Location of Sample Files

This sample is located in the xmlmap folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\xmlmap\OutputScriptMap.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.

    ♦ On Microsoft Windows systems, from the Start menu navigate to:
    BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.

    ♦ On Linux or Solaris systems, run:
    BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/xmlmap/OutputScriptMap.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test Form tab of Test View, if necessary.
4. Enter values for fname and lname and invoke the Hello method.
5. The Hello method receives the values using the default mapping (no custom XML map), but returns its result using an XML map specified with the @jws:return−xml tag. The @jws:return−xml tag includes a <xm:use> tag that calls the ConvertPersonToXML ECMAScript function in the PersonScript.jsx file.
6. The ECMAScript function formats the fields of the Person object into a schema it determines, then returns that XML to the client.
7. Examine PersonScript.ps and the relationship between the ECMAScript it contains and the method in OutputScriptMap.jws.

Related Topics

ConsolidateAddress.jws Sample

OutputMap.jws Sample

Handling and Shaping XML Messages with XML Maps

Handling XML with ECMAScript Extensions

<xm:use> Tag

@jws:return−xml Tag

Test View

QuoteClient.jws Sample

Demonstrates the ability to modify a Service control's CTRL file to change the Java signature of one or more methods while still adhering to the public contract of the called web service. QuoteServiceControl.ctrl was originally generated form the QuoteService.jws web service. But one method in QuoteServiceControl.ctrl has had a parameter removed and replaced with a hard−coded value in the accompanying XML map. Thus, the outgoing XML message is the same, but the Java signature visible to clients of the Service control is simplified.

Concepts Demonstrated by this Sample

- Use of a Service control
- Customization of a Service control
- Declaration and use of a client callback
- Use of Conversations

Location of Sample Files

This sample is located in the service folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\service\QuoteClient.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.

   - On Microsoft Windows systems, from the Start menu navigate to:
   BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.

   - On Linux or Solaris systems, run:
   BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/service/QuoteClient.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test Form tab of Test View, if necessary.
4. Enter a value for tickerSymbol (BEAS, of course) and invoke the getQuote method.
5. QuoteClient then calls the getQuote method of QuoteService, via the QuoteServiceControl Service control, which will return a price quote for the specified ticker symbol.

Running this sample isn't the interesting aspect of it. If you examine QuoteService.jws, you will see that its getQuote method expects two parameters. But QuoteClient is calling the method with a single parameter. This is accomplished via an XML map on the QuoteServiceControl Service control.

QuoteServiceControl.ctrl was originally generated from QuoteService.jws. It was then modified: the first parameter of getQuote was removed from the Java signature, meaning callers are no longer expected to pass it. An XML map was then added to getQuote, with the value of the <customerID> element hard−coded. This leaves the message shape as QuoteService expects it, with two parameters. The calling service passes one parameter, which the Service control combines with the hard−coded parameter to produce the two−parameter message the target service expects.

Related Topics

Controls: Using Resources from a Web Service

Service Control: Using Another Web Service

Customizing a Service Control: Defining Java to XML Translation with XML Maps

Maintaining State with Conversations

Using Callbacks to Notify Clients of Events

JwsContext Interface

Test View

SimpleJMS.jws Sample

A web service that demonstrates use of a JMS control that sends to and receives from a queue.

Concepts Demonstrated by this Sample

- Use of a JMS control with a queue
- Declaration and use of a client callback
- Use of Conversations

Location of Sample Files

This sample is located in the jms folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\jms\SimpleJMS.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.

   ♦ On Microsoft Windows systems, from the Start menu navigate to:
   BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.

   ♦ On Linux or Solaris systems, run:
   BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/jms/SimpleJMS.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test Form tab of Test View, if necessary.
4. Enter a value for name and invoke the start method to start a conversation.
5. Navigate to the continue and finish methods for the conversation by clicking on the conversation ID (the large number) at the top of the section in the Message Log.
6. Enter values for msg and invoke the sendString method. The values are packaged in a message and sent to a JMS queue via the JMS control myCustomQ, which is an instance of CustomJMSControl.

7. Since the JMS control in this simple example is configured to listen to the same queue it sends to, the message immediately arrives and is forwarded to the web service via the onMessageReceived callback. Refresh the browser to see the callback entry appear in the Message Log.
8. Select log entries in the Message Log to see the message traffic involved in each interaction.

Related Topics

CustomJMSClient.jws Sample

Controls: Using Resources from a Web Service

JMS Control: Using Java Message Service Queues and Topics from Your Web Service

Maintaining State with Conversations

Test View

SimpleMap.jws Sample

A sample showing simple use of a @jws:parameter−xml XML map to convert XML input to Java objects. Uses curly brace "{}" notation as implicit <xm:value> tags.

Concepts Demonstrated by this Sample

- XML maps

Location of Sample Files

This sample is located in the xmlmap folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\xmlmap\SimpleMap.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.

    ♦ On Microsoft Windows systems, from the Start menu navigate to:
    BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.

    ♦ On Linux or Solaris systems, run:
    BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/xmlmap/SimpleMap.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test Form tab of Test View, if necessary.

4. The single method of the web service, acceptPerson, specifies an XML map for incoming data. The Test Form displays a map conforming to the expected schema.
5. Enter values for each of the XML tag values that start with Value_ (replace the entire text string with a new value).
6. Invoke the acceptPerson method.
7. acceptPerson extracts the data from the XML map into Java parameters to the method, formats a response string, then returns a single floating point value.
8. Note the use of the Java String class' trim method to remove extra whitespace from values extracted from XML.

Related Topics

InputMapMultiple.jws Sample

Handling and Shaping XML Messages with XML Maps

@jws:parameter−xml Tag

Test View

SimpleTimer.jws Sample

A simple web service that demonstrates use of the Timer control. Uses the @jws:timer timeout= tag.

Concepts Demonstrated by this Sample

- Use of a Timer control
- Declaration and use of a client callback
- Use of Conversations

Location of Sample Files

This sample is located in the timer folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\timer\SimpleTimer.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.

    ♦ On Microsoft Windows systems, from the Start menu navigate to:
    BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.

    ♦ On Linux or Solaris systems, run:
    BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/timer/SimpleTimer.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the

sample.
3. Navigate to the Test Form tab of Test View, if necessary.
4. Invoke the createTimer method to create a new conversational instance.
5. Click on the conversation ID (the large number) in the Message Log to access this conversation's continue and finish methods.
6. Invoke the setAlarm method. This starts the Timer control, which will expire in 5 seconds.
7. When the timer expires, it calls its onTimeout callback, which arrives at SimpleTimer via the timer_onTimeout callback handler.
8. The timer_onTimeout callback handler in turn invokes SimpleTimer's onAlarm callback to the client.
9. You must refresh the browser to see callbacks arriving from the service. The use of a browser for Test View does not allow WebLogic Server to push notification to the browser when callbacks execute.
10. setAlarm uses the Timer control's restart method, so invoking setAlarm before the timer expires will reset it so that it will expire 5 seconds from the invocation of setAlarm.
11. Refresh the browser periodically until the callback.onDone callback entry appears in the Message Log.
12. Select log entries in the Message Log to see the message traffic involved in each interaction.

Related Topics

AdvancedTimer.jws Sample

Controls: Using Resources from a Web Service

Timer Control: Using Timers in Your Web Service

Maintaining State with Conversations

Using Callbacks to Notify Clients of Events

Test View

TraderEJBClient.jws Sample

A web service that demonstrates use of the EJB control TraderEJBControl.ctrl, which represents the TraderEJB Stateless Session Bean and exposes its business interface to web services.

Concepts Demonstrated by this Sample

- Use of an EJB control
- Use of Conversations

Location of Sample Files

This sample is located in the ejbControl folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\ejbControl\AccountEJBClient.jws

How to Run the Sample

To run this web service:

1. Start WebLogic Server in the appropriate domain.

   ♦ On Microsoft Windows systems, from the Start menu navigate to:
   BEA WebLogic Platform 7.0−>WebLogic Workshop−>WebLogic Workshop Examples−>Start Examples Server.

   ♦ On Linux or Solaris systems, run:
   BEA_HOME/weblogic700/samples/workshop/startWebLogic.sh
2. Launch the service either by opening it in WebLogic Workshop and selecting the Start operation or by entering http://localhost:7001/samples/ejbControl/AccountEJBClient.jws in the address bar of your browser. If WebLogic Server is running in the appropriate domain on this machine, you may click here to run the sample.
3. Navigate to the Test Form tab of Test View, if necessary.
4. Enter values for tickerSymbol and numberOfShares in for either the buy or sell method and invoke the method.
5. Click on the Message Log title to return to the methods.
6. TraderEJB, the EJB that is represented by TraderEJBControl, is a Stateless Session Bean and therefore does not store any information. Its sole purpose is to qualify transacations: it limits all transactions to 500 shares.
7. Examine the TraderEJBClient.jws and TraderEJBControl.ctrl files to explore the relationship between the control and its client. The TraderEJBControl.ctrl file was created using the Add EJB Control dialog.
8. Select log entries in the Message Log to see the message traffic involved in each interaction.

Related Topics

AccountEJBClient.jws Sample

Controls: Using Resources from a Web Service

EJB Control: Using Enterprise Java Beans from a Web Service

Test View

Java Client Samples

Any application can communicate with a web service if it can generate and consume XML messages and use one of the protocols supported by the target web service. WebLogic Workshop provides Java proxy classes that allow any Java program to use a particular WebLogic Workshop web service. The proxy classes allow the Java application to treat the web service as though it is a normal Java class.

The proxy classes, along with supporting classes provided by WebLogic Server, perform the following work:

- convert Java method invocations on the proxy to appropriate XML messages
- manage sending the XML messages to the web service over an appropriate protocol
- receive response messages
- convert received XML response messages into Java types

Concepts Demonstrated by this Sample

- Use of a web service's Java proxy from a JSP
- Use of a web service's Java proxy from a Java console application
- Use of a web service's Java proxy from a Java Swing application.

Location of Sample Files

These samples are located in the proxy folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\proxy

How to Run the Sample

Step by step instructions for building and running the samples may be found in the readme.html file in the proxy sample directory (above). If you are viewing this documentation on a machine running WebLogic Server in the WebLogic Workshop samples domain, you may click here to view the readme.html file.

Related Topics

How Do I: Communicate with a Web Service from a JSP or Servlet?

How Do I: Communicate with a Web Service from Another Java Application?

How Do I: Tell Developers of Non−WebLogic Workshop Clients How to Participate in Conversations?

How Do I: Use the Java Proxy for a Web Service?

.NET Client Sample

Any application can communicate with a web service if it can generate and consume XML messages and use one of the protocols supported by the target web service. This includes applications and web services built in other tools and other languages. The central reason for the development of the web service concept was to enable inter−operation of software components regardless of the operating system or language with which they are implemented. The key to this inter−operation is the WSDL file, which describes a web service in an implementation−independent way.

Visual Studio .NET can use the WSDL file of a web service to produce a web service proxy. A .NET component can then use the proxy class to communicate with the web service as though it were a local object.

Concepts Demonstrated by this Sample

- Use of a WebLogic Workshop web service from an ASP.NET web service.
- Non−WebLogic Workshop web service client participating in a conversation.
- Non−WebLogic Workshop web service accepting a callback.

Location of Sample Files

This sample is located in the interop\dotNET folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\interop\dotNET

How to Run the Sample

Step by step instructions for building and running the .NET web service client may be found in the readme.html file in the interop\dotNET sample directory (above). If you are viewing this documentation on a machine running WebLogic Server in the WebLogic Workshop samples domain, you may click here to view the readme.html file.

Related Topics

How Do I: Communicate with a Web Service from a JSP or Servlet?

How Do I: Communicate with a Web Service from Another Java Application?

How Do I: Tell Developers of Non−WebLogic Workshop Clients How to Participate in Conversations?

How Do I: Use the Java Proxy for a Web Service?

Widgets Sample Application

Widgets represents a complete business process implemented as web services. The sample application makes heavy use of XML maps to demonstrate loose coupling between web service implementation and message format.

Concepts Demonstrated by this Sample

- Coordination of multiple web services.
- Use of XML maps to decouple message format from web service implementation.

Location of Sample Files

This sample is located in the apps\widgets folder of the samples WebLogic Workshop project. In the file system the location is:

BEA_HOME\weblogic700\samples\workshop\applications\samples\apps\widgets

How to Run the Sample

Step by step instructions for running the sample may be found in the readme.html file in the apps\widgets sample directory (above). If you are viewing this documentation on a machine running WebLogic Server in the WebLogic Workshop samples domain, you may click here to view the readme.html file.

Related Topics

Samples

Sample Web Services

WebLogic Workshop Reference

This section provides reference information about WebLogic Workshop, including its visual components (such as the visual development environment) and its programmatic components (such as custom Javadoc tags).

Topics Included in This Section

Class Reference

Provides reference information about WebLogic Workshop classes.

Javadoc Tag Reference

Provides reference information about WebLogic Workshop Javadoc tags.

XML Map Tag Reference

Provides reference information about WebLogic Workshop tags for XML mapping.

User Interface Reference

Provides reference information about the WebLogic Workshop visual development environment, including its dialog boxes and menu commands.

Configuration File Reference

Provides reference information on several files that provide configuration information for WebLogic Workshop.

Command Reference

Provides information on WebLogic Workshop commands and their arguments.

CTRL Files: Implementing Controls

Files with the extension CTRL are WebLogic Workshop controls. They typically include a collection of method definitions that allow you to easily access a resource such as a database or another web service.

To learn more about controls, see Controls: Using Resources from a Web Service.

Types of CTRL Files

The contents of a CTRL file depend on the type of control. CTRL files can represent the following types of controls:

- Service Control: used to communicate with another web service from your service. For more information on Service Controls, see Service Control: Using Another Web Service.
- Database Control: used to access a database from your web service. For more information on Database Controls, see Database Control: Using a Database from Your Web Service.
- EJB Control: used to access an existing Enterprise Java Bean (EJB) from your web service. For more information on EJB Controls, see EJB Control: Using an Enterprise Java Bean from Your Web Service.
- JMS Control: used to access an existing Java Message Service (JMS) queue or topic from your web service. For more information on JMS Controls, see JMS Control: Using Java Message Service Queues and Topics from Your Web Service.

For more information on WebLogic Workshop controls, see Controls: Using Resources from a Web Service.

Using CTRL Files

At times, you will use CTRL files that were provided to you by the implementor of a service or control. You might use this CTRL file as you received it, or you may modify it to customize some aspects of its behavior. In other cases, you may create a CTRL file and manually code method definitions and other information. CTRL files also provide a flexible way for WebLogic Workshop web services to interact with each other. Each of these scenarios is described in more detail below.

Using an Existing CTRL File

In some cases, you may use an existing CTRL file that was produced by another member of your team or another organization. For example, if many web services will use the same database, a single author might create a CTRL file that describes the interface to the database. Then multiple web service authors might use that CTRL file to create a Database Control in their service and use it to access the common database. The same situation can occur for all of the control types.

Writing a CTRL File

You may find occasions in which you need to create a new CTRL file. For example, you may wish to gain access to an existing Enterprise Java Bean via an EJB Control. To do so, you would write the code for a new CTRL file containing the information necessary to access the specific EJB you want to use. The CTRL file syntax and the WebLogic Workshop controls are designed to make this easy; you don't need to know very much about EJBs to use one via an EJB Control.

For more information on EJB Controls, see EJB Control: Using an Enterprise Java Bean from Your Web Service.

Generating a CTRL File

There are two situations in which you may wish to generate CTRL files:

- To allow other WebLogic Workshop web services to access your service: authors of other WebLogic Workshop web services may create a Service Control to access your service by specifying a CTRL file.

For more information on Service Controls, see Service Control: Using Another Web Service.

For more information on generating a CTRL file from a JWS file, see How Do I: Generate a CTRL File?

- To access a web service based on the service's WSDL file: you can generate a Service control CTRL file from the WSDL file, then use the new Service from any web service.

For more information on WSDL files, see WSDL Files.

For more information on Service Controls, see Service Control: Using Another Web Service.

Related Topics

How Do I: Generate a CTRL File?

Structure of a CTRL File

The structure of a CTRL file depends on the type of control it represents. The structure of CTRL files for each type of control is described in the topic that described creation of that type of control. See the following topics for more information:

- Creating a New Database Control
- Creating a New EJB Control
- Creating a New JMS Control
- Service Control: Using Another Web Service

Related Topics

Controls: Using Resources from a Web Service

CTRL Files: Implementing Controls

How Do I: Generate a CTRL File?

WebLogic Workshop will generate a CTRL files defining various types of controls. Some types of controls allow you to modify their CTRL files in various ways, while other types do not allow useful modifications. How you generate a CTRL file depends on the type of control you are creating. The following sections reference information about how to generate the CTRL file for each type of control.

Service Control

A Service control allows WebLogic Workshop web services to use another web service using a simple Java interface. The other web service can originate in any web service building tool.

To learn how to create a CTRL file defining a Service control, see Creating a New Service Control.

To learn about Service controls, see Service Control: Using Another Web Service.

Database Control

A Database control allows WebLogic Workshop web services to use a database using a simple Java interface.

To learn how to create a CTRL file defining a Database control, see Creating a New Database Control.

To learn about Database controls, see Database Control: Using a Database from Your Web Service.

JMS Control

The JMS control allows WebLogic Workshop web services to use JMS queues and topics using a simple Java interface.

To learn how to create a CTRL file defining a JMS control, see Creating a New JMS Control.

To learn about JMS controls, see JMS Control: Using Java Message Service Queues and Topics from Your Web Service.

EJB Control

The EJB control allows WebLogic Workshop web services to use Enterprise Java Beans using a simple Java interface.

To learn how to create a CTRL file defining an EJB control, see Creating a New EJB Control.

To learn about EJB controls, see EJB Control: Using Enterprise Java Beans from a Web Service.

AppView Control

The AppView control allows WebLogic Workshop web services to use a WebLogic Integration ApplicationView using a simple Java interface.

To learn how to create a CTRL file defining an AppView control, see Creating a New AppView Control.

To learn about AppView controls, see AppView Control: Accessing an Enterprise Application from a Web Service.

Related Topics

CTRL Files: Implementing Controls

Controls: Using Resources from a Web Service

JWS Files: Java Web Services

A JWS file contains a Java Web Services. JWS and CTRL files are the core file types that you deal with when building WebLogic Workshop web services. This topic is an overview of the JWS file format.

For a conceptual overview of WebLogic Workshop web services, see Development with WebLogic Workshop.

For more information on CTRL files, see CTRL Files.

What Is a JWS File?

A JWS file contains syntactically correct Java. However, a JWS file also has attributes that allow it to take advantage WebLogic Workshop's powerful facilities for web services:

- It contains Javadoc tags specific to WebLogic Workshop that indicate that helper objects should be automatically generated and associated with the class at both compile time and runtime. These Javadoc tags all begin with @jws:.
- It has the JWS extension, which, when it appears in a URL, indicates to WebLogic Server that the file should be handled as a web service.

A JWS file contains only the logic you need to implement your web service. All of the underlying infrastructure, protocols and web service lifecycle management are handled automatically by WebLogic Server.

Related Topics

Structure of a JWS File

Sample Web Services

WebLogic Workshop Projects

Structure of a JWS File

A JWS file is a valid Java file. If you are not familiar with Java and would like to read about basic features of the language, see Introduction to Java.

This topic discusses the individual elements that occur in a JWS file. For a collection of examples of JWS files, see Sample Web Services.

Parts of a JWS File

The following parts of a JWS file are described in this topic:

Package Statement

Import Statements

Class Declaration

Member Variables

Control Instances

JwsContext Object

Avoiding Improper Initialization

Callback Interface

Package Statement

The package statement specifies the Java package in which the classes and objects in this file reside. In JWS files, the package is dictated by the directory in the project within which the JWS file is located. If the JWS file is in the top level directory of the project, a package statement is not necessary. The file is in the default package for the project.

If the JWS file resides in a subdirectory of the project, the package name must reflect the directory hierarchy. For example, if the JWS file is in the <project>/financial/clientservices directory, it must contain the following package statement:

```
package financial.clientservices;
```

WebLogic Workshop attempts to manage the package statement for you. For example, if you move a JWS file from one directory to another using the Project Tree Pane, WebLogic Workshop will update the package name in the moved file. However, you will still encounter situations in which you must correctly set the package name yourself.

Import Statements

Before you can reference a class or object that is not defined in a Java file, you must import the definition of that class or object (or the package that defines it). You do so with import statements.

Packages or classes you might typically import in a JWS file include:

```
import weblogic.jws.control.TimerControl;  // TimerControl API
import weblogic.jws.control.JwsContext;    // service's execution context
```

Class Declaration

A JWS file must include the declaration of a single top−level public class. This is the class that defines your web service. The name of your web service is the name of this class. The class declaration for a web service called MyService would appear as follows:

```
public class MyService
{
    ...
}
```

All of the remaining items described in this topic are member variables or member functions of the web service's class.

To learn how to document your web service, see Documenting Web Services.

Member Variables

The web service class may contain member variables.

```
public class MyService
{
    int myMemberVariable;
    ...
}
```

Unannotated member variables are normal members of the class. These member variables are automatically persisted and correlated with specific client conversations if the methods of the service are marked as conversational.

To learn more about conversations, see Maintaining State with Conversations.

Some member variables are annotated as having special meaning. See the sections on Control Instances and the Callback Interface, below.

Control Instances

If a member variable is annotated with the @jws:control tag, the member variable represents a control. Controls provide convenient interfaces to resources such as databases, Enterprise Java Beans (EJBs) or other web services.

```
public class MyService
{
    int myMemberVariable;



    /**
     *  @jws:control
     */
    TimerControl delayTimer;
    ...
}
```

For more information on controls, see Controls: Using Resources from a Web Service.

JwsContext Object

Each JWS file may have a special member variable of type weblogic.jws.control.JwsContext annotated with the @jws:context tag. By default, new JWS files created by WebLogic Workshop contain such a member variable with the name context.

```
public class MyService
{


    int myMemberVariable;



    /**
     *  @jws:context
     */
    JwsContext context;
    ...
}
```

If such a JwsContext instance is present in a JWS file, the instance may be used to access aspects of the JWS's context at runtime.

For more information on the JwsContext interface, see JwsContext Interface.

Avoiding Improper Initialization

Controls and context objects are not valid outside the scope of web service methods. Initialization code associated with the declarations of member variables of a JWS file is executed when the JWS class is instantiated. Since instantiation of the JWS class occurs outside of any particular web service method invocation, such initializations will fail or lead to unpredictable results if they reference controls or the context instance.

In the following example code, the serviceResult member variable is initialized by calling a method on the someService control. This code is not valid, since at the time the initialization code is executed the someService control instance is not populated. This code will throw a NullPointerException at runtime.

```
public class MyService
{
    /** @jws:control */
    private SomeServiceControl someService;
    ...
    String serviceResult = someService.getResult();
}
```

The following code is correct:

```
public class MyService
{
    /** @jws:control */
    private SomeServiceControl someService;
    ...
    String serviceResult;



    /**
     * @jws:operation
     */
    public void someMethod()
    {
        ...
        serviceResult = someService.getResult();
        ...
    }
}
```

Callback Interface

The callback interface is an inner interface (an interface declared within another class or interface) of the JWS class, within which the callbacks to the client are defined. When you add a callback to your service in the Design View, the signature for the callback method becomes a member of the callback interface.

```
public class MyService
```

```
{
    int myMemberVariable;


    /**
     *  @jws:control
     */
    TimerControl delayTimer;


    public static interface Callback
    {
        public void myCallback(String stringArg);
    }
    public Callback callback;
    ...
}
```

The callback interface is managed for you by WebLogic Workshop. Although, as with all aspects of JWS and CTRL files, you may edit it manually if you wish.

The contents of the callback interface are exported to the WSDL file for your service.

By specifying callbacks in your service, you are expressing an expectation that your service's clients will handle them. This may not be possible for some clients for a variety of reasons.

To learn how to document the callbacks your web service defines, see Documenting Web Services.

Inner Classes

An inner class is a class declared within another class or interface. Common uses for inner classes include:

- arguments and return values of methods

If your web service publishes a method or callback that accepts or returns a Java object, the object will be translated to or from the class by an implicit or explicit XML map.

For more information on XML maps, see Why Use XML Maps?

- mirror of a database record structure, into which query results can be mapped

You can create an inner class with the member names and types that match the column names returned by a database query. You can then specify that a particular database query should map it's result to an object of that class.

For more information on using a database from your service, see Database Control: Using a Database from Your Web Service.

The example below demonstrates declaration of Customer as an inner class of the MyService class.

```
public class MyService
{
    int myMemberVariable;
    /**
     *  @jws:control
     */
    TimerControl delayTimer;



    public static interface Callback
    {
        public void myCallback(String stringArg);
    }
    public Callback callback;



    public class Customer
    {
        public int custid;
        public String name;
        public String emailAddress;
    }
    ...
}
```

Methods

In this topic, methods are distinguished from internal methods, which are described in the next section.

The term method refers to a method that is exposed to clients of your service. A method is exposed to the client when it is marked with the @jws:operation tag. Methods that are exposed to the client must be declared with the public access modifier.

In the following example, myMethod is a method of the service MyService:

```
public class MyService
{
    int myMemberVariable;



    /**
     *  @jws:control
     */
    TimerControl delayTimer;



    public static interface Callback
    {
        public void myCallback(String stringArg);
    }
    public Callback callback;



    public class Customer
    {
        public int custid;
        public String name;
        public String emailAddress;
    }
    /**
     *  @jws:operation
     */
    public String myMethod(String stringArg)
    {
        ...
    }



    ...
}
```

To learn how to document the methods in your web service, see Documenting Web Services.

Internal Methods

In this topic, internal methods are distinguished from methods, which are described in the preceding section.

The term internal method refers to a method that is not exposed to clients of your service. An internal method has no special annotation; it is a normal Java class member. Internal methods may be declared with whatever access modifier (public, private, protected) your design requires.

In the following example, myInternalMethod is an internal method of the service MyService:

```
public class MyService
{
    int myMemberVariable;



    /**
     *  @jws:control
     */
    TimerControl delayTimer;



    public static interface Callback
    {
        public void myCallback(String stringArg);
    }
    public Callback callback;



    public class Customer
    {
        public int custid;
        public String name;
        public String emailAddress;
    }



    /**
     *  @jws:operation
     */
    public String myMethod(String stringArg)
    {
        ...
    }


    private method myInternalMethod(int intArg)
    {
        ...
    }



    ...
}
```

Related Topics

Introduction to Java

Sample Web Services

WSDL Files: Web Service Descriptions

Files with the WSDL extension contain web service interfaces expressed in the Web Service Description Language (WSDL). WSDL is a standard XML document type controlled by the World Wide Web Consortium (W3C, see www.w3.org for more information).

WSDL files are used to communicate interface information between web service producers and consumers. An interface allows you to utilize a service's capabilities without possessing the source code for the service.

Contents of a WSDL File

WSDL files contain all of the information necessary for a client to invoke the methods of a web service:

- The data types used as method parameters or return values are defined.
- The individual methods are defined. WSDL refers to methods as operations.
- The protocols and message formats allowed for each method are specified.
- The URLs used to access the web service are specified.

Imported WSDL Files

When you want to use another web service that was not built with WebLogic Workshop, you should first obtain the WSDL file for the service you want to use. For public web services, the WSDL file will typically be available on the web site of the organization that publishes the web service. For private web services, contact the organization that supports the web service to obtain the WSDL file.

WSDL files can also be found through both public and private UDDI registries. To learn more about UDDI, visit http://www.uddi.org.

Once you have the WSDL file, you may use WebLogic Workshop to create a CTRL file defining a Service control. The Service control may then be used from your web service like any other WebLogic Workshop control.

For more information on producing a CTRL file from a WSDL file, see Creating a Service Control from a WSDL File.

For more information on CTRL files, see CTRL Files: Implementing Controls.

If the web service you wish to use was built with WebLogic Workshop, you may use a Service control CTRL file directly from the JWS file.

To learn more about Service controls, see Service Control: Using Another Web Service.

Generated WSDL Files

When you want to make your web service available to others, you do so by producing a WSDL file for your web service and making it available to your service's clients.

For more information on generating WSDL files, see How Do I: Publish a WSDL File for My Web Service?.

If your clients are web services built with WebLogic Workshop, they can use a Service control CTRL file generated directly from your web service's JWS file.

For more information on CTRL files, see CTRL Files: Implementing Controls.

To learn more about Service controls, see Service Control: Using Another Web Service.

Related Topics

WebLogic Workshop Projects

W3C WSDL Specification

How Do I: Generate a WSDL File?

See How Do I: Publish a WSDL File for My Web Service? .

Related Topics

WSDL Files: Web Service Descriptions

CTRL Files: Implementing Controls

XMLMAP Files: Shared XML Maps

Files with the extension XMLMAP contain, as you might expect, XML maps. You can use XML maps when your service must handle a particular format for XML messages it receives or send a particular format to other components. XML maps your service uses can be kept inline with the code for your service or stored in a separate map file with an XMLMAP extension.

When you keep XML maps in a map file, you get the added benefit of being able to use the maps from various places in your code; inline maps, on the other hand, may be applied only to the code they precede.

For step−by−step instructions on creating an XML map file, see How Do I: Begin a Reusable XML Map?

Related Topics

Creating Reusable XML Maps

Class Reference

This section contains reference topics for classes included with WebLogic Workshop. Most of these classes contain functionality for controls. In addition, the JwsContext interface exposes several methods that are useful within web services.

Topics Included in This Section

DatabaseControl Interface

Base interface for all Database controls.

EJBControl Interface

Base interface for all EJB controls.

EntityEJBControl Interface

Base interface for EJB controls that represent Entity Beans. Subclass of EJBControl.

JMSControl Interface

Base interface for all JMS controls.

JwsContext Interface

Represents service−related run−time functionality provided by WebLogic Server, such as conversation context.

ServiceControl Interface

Base interface for all Service controls.

SessionEJBControl Interface

Base interface for EJB controls that represent Session Beans. Subclass of EJBControl.

TimerControl Interface

Base interface for all Timer controls.

Related Topics

Controls: Using Resources from Your Web Service

Maintaining State with Conversations

ApplicationViewControl Interface

Base interface for all AppView controls.

Syntax

```
public interface ApplicationViewControl extends Control
```

Remarks

The methods of this interface may be invoked by any web service with an AppView control instance.

Members

Constructors

None.

Methods

public void beginLocalTransaction() throws Exception

Begin a local transaction on this control. This will begin a local transaction on the underlying ApplicationView instance. All work done by this control instance between this call and a call to commitLocalTransaction() or rollbackLocalTransaction() will be committed or rolled back, respectively, as a unit.

If the underlying adapter used by the ApplicationView for this control does not support local transactions, an exception is thrown.

public void commitLocalTransaction() throws Exception

Commit the active local transaction for this control. All work done since the last call to beginLocalTransaction() will be committed into the EIS's permanent state.

If the underlying adapter used by the ApplicationView for this control does not support local transactions, an exception is thrown.

public void disableEventDelivery() throws Exception

Indicates that this control should no longer deliver event notifications. Calling this method when event delivery is not enabled has no effect.

public void enableEventDelivery() throws Exception

Indicates that this control instance should provide event notifications to its container JWS instance. The containing JWS must be conversational. Events will be delivered using the MODE_ALLOW_DUPS receive mode. Calling this method when event delivery is already enabled has no effect.

public void enableEventDelivery(String receiveMode) throws Exception

Indicates that this control instance should provide event notifications to its container JWS instance using the specified receive mode. The containing JWS must be conversational. Calling this method when event delivery is already enabled is has no effect.

receiveMode specifies the mode in which all events should be received. Must be one of the MODE_XXX fields defined below.

public String getEventReceiveMode()

Returns the receive mode specified in the latest call to enableEventDelivery() or null if event delivery is not enabled. Returned mode is one of the MODE_XXX fields defined below.

public void importJar(String jarFileName, boolean overwrite, boolean deploy, List errors, PrintWriter out, boolean quiet) throws Exception

Import a previously exported ApplicationView JAR file. This file can contain ApplicationView, ConnectionFactory, Folder, and Schema definitions.

jarFileName: Either a fully qualified path to the jar to be imported, or a resource name suitable for finding the jar on the system classpath (e.g. samples/appview/Sample1.jar). This method will interpret jarFileName first as a path, and if the file does not exist, will attempt to find it on the classpath.

overwrite: A flag indicating if the imported objects should overwrite any existing objects already in the repository.

deploy: A flag indicating if the deployable imported objects should be deployed automatically after the import.

errors: A list that, upon completion of the import, will contain any non−fatal errors that occurred. Fatal errors are reported via an Exception thrown by this method.

out: A PrintWriter instance to which all messages will be written. If null, no messages will be written.

quiet: A flag indicating whether the import should operate quietly (producing fewer messages).

An Exception is thrown if any fatal error occurs during the import.

public String importJar(String jarFileName) throws Exception

Import a previously exported ApplicationView JAR file using standard options. This method will overwrite any existing objects with the same name as those in the jar, will deploy deployable objects, and will print a summary of operations to System.out. If non−fatal errors occur, this method will print a summary of those errors to System.out.

jarFileName: Either a fully qualified path to the jar to be imported, or a resource name suitable for finding the jar on the system classpath (e.g. samples/appview/Sample1.jar). This method will interpret jarFileName first as a path, and if the file does not exist, will attempt to find it on the classpath.

Returns a string description of the import operation, including any non−fatal errors.

An Exception is thrown if any fatal error occurs during the import.

public boolean isEventDeliveryEnabled()

Returns true if event delivery is enabled, false otherwise. This is true after a call to enableEventDelivery() and before a call to disableEventDelivery().

public void rollbackLocalTransaction() throws Exception

Rollback the active local transaction for this control. All work done since the last call to beginLocalTransaction() will be discarded.

If the underlying adapter used by the ApplicationView for this control does not support local transactions, an exception is thrown.

Fields

static public final String MODE_ALLOW_DUPS

Use this mode if it is acceptable to receive a given event more than once. This will generally only be possible in the event of a system shutdown or failure, and will generally not occur during normal operations. This mode is the default and will give the best performance.

static public final String MODE_ONCE_AND_ONLY_ONCE

Use this mode if it is critical to receive all events and that each event be received only once, even in the event of a system failure. This mode will yield performance somewhat lower than when using MODE_ALLOW_DUPS.

Events

public void onEvent(Object event) throws Exception;

The onEvent event is generated whenever an application view event is received. This method is private, and should not be used by JWS clients.

event: The event object representing the event in the EIS.

public void onAsyncServiceResponse(Object asr) throws Exception;

The onAsyncServiceResponse event is generated whenever an application view async response is received. This method is private, and should not be used by JWS clients.

event: The async response object representing the response from an async service invocation.

public void onAsyncServiceError(String requestID, String errorMessage) throws Exception

The onAsyncServiceError event is generated whenever an application view async request ends in error.

requestID: The identifier returned when this request was submitted.

errorMessage: The text of the error that has occurred.

Related Topics

AppView Control: Accessing an Enterprise Application from a Web Service

Using an AppView Control

DatabaseControl Interface

Base interface for all Database controls.

Syntax

```
package weblogic.jws.control;

public interface DatabaseControl extends Control
```

Remarks

The methods of this interface may be invoked by any web service with a Database control instance.

Members

Constructors

none.

Methods

java.sql.Connection getConnection()

Returns the database connection associated with this instance of the Database control.

Fields

none.

Events

none.

Related Topics

Database Control: Using a Database from Your Web Service

@jws:connection Tag

CustomerDBClient.jws Sample

LuckyNumberDBClient.jws Sample

EJBControl Interface

Base interface for all EJB controls. Subclasses EntityEJBControl and SessionEJBControl are used directly by EJB controls.

Syntax

```
package weblogic.jws.control;
```

```
public class EJBControl extends Control
```

Remarks

The methods of this interface may be invoked by any web service with a EJB control instance.

Members

Constructors

none.

Methods

Object getEJBBeanInstance()

Returns the current target instance of the bean business interface used for business interface method invocations.

Throwable getEJBException()

Returns the last EJB exception serviced by the EJB control on the developers behalf.

Object getEJBHomeInstance()

Returns an instance of the home interface associated with the target bean component.

boolean hasEJBBeanInstance()

Returns true if the EJB control currently has a bean instance upon which business interface methods may be invoked.

Fields

none.

Events

none.

Related Topics

EJB Control: Using Enterprise Java Beans from a Web Service

Overview: Enterprise Java Beans (EJBs)

EntityEJBControl Interface

SessionEJBControl Interface

TraderEJBClient.jws Sample

AccountEJBClient.jws Sample

EntityEJBControl Interface

Base interface for EJB controls that represent Entity Beans. Subclass of EJBControl.

Syntax

```
package weblogic.jws.control;

public class EntityEJBControl extends EJBControl
```

Remarks

The methods of this interface may be invoked by any web service with a EJB control instance.

Members

Constructors

none.

Methods

For inherited methods, see EJBControl Interface.

Object getEJBNextBeanInstance()

Supports iteration through a Collection of entity bean instances returned by a multi−select finder method.

Fields

none.

Events

none.

Related Topic

Overview: Enterprise Java Beans (EJBs)

EJB Control: Using Enterprise Java Beans from a Web Service

EJBControl Interface

SessionEJBControl Interface

JMSControl Interface

Base interface for all JMS controls.

Syntax

```
package weblogic.jws.control;


public class JMSControl extends Control
```

Remarks

The methods of this interface may be invoked by any web service with a JMS control instance.

Only one of the three callbacks may have a handler implemented in any given JWS file. Implementing more than one callback handler is an error.

Members

Constructors

none.

Methods

Map getHeaders()

Gets the JMS headers of the last message received. If no message has been received then null is returned. May contain one or more HEADER_xxx fields described below as keys.

Map getProperties()

Gets the JMS properties of the last message received. If no message has been received then null is returned. The return value maps property names (Strings) to property values.

javax.jms.Session getSession()

Returns the JMS session used by this control.

void setHeaders(Map headers)

Sets the JMS headers to be assigned to the next JMS message sent. Typical usage is to create a java.util.HashMap to hold the headers, then populate it using the HEADER_xxx fields described below as keys.

Note that these headers are set only on the next message, subsequent messages will not get these headers. Also note that if the next message is sent through publishMessage(), then any headers set through this map will override headers set in the message itself.

void setProperties(Map properties)

Sets the JMS properties to be assigned to the next JMS message sent. Note that these properties are set only on the next message, subsequent messages will not get these properties. Also note that if the next message is sent through publishMessage(), then any properties set through this map will override properties set in the message itself.

void subscribe()

Indicates that this control is now interested in receiving incoming messages published to the topic.

void unsubscribe()

Indicates that this control is no longer interested in receiving incoming messages published to the topic.

Fields

public static final String HEADER_CORRELATIONID

Used with getHeaders()/setHeaders() methods.

public static final String HEADER_DELIVERYMODE

Used with getHeaders()/setHeaders() methods.

public static final String HEADER_EXPIRATION

Used with getHeaders()/setHeaders() methods.

public static final String HEADER_MESSAGEID

Used with getHeaders()/setHeaders() methods.

public static final String HEADER_PRIORITY

Used with getHeaders()/setHeaders() methods.

public static final String HEADER_REDELIVERED

Used with getHeaders()/setHeaders() methods.

public static final String HEADER_TIMESTAMP

Used with getHeaders()/setHeaders() methods.

public static final String HEADER_TYPE

Used with getHeaders()/setHeaders() methods.

Related Topics

JMS Control: Using Java Message Service Queues and Topics from Your Web Service

Overview: Messaging Systems and JMS

SimpleJMS.jws Sample

CustomJMSClient.jws Sample

Represents the execution context of the web service. Methods in this interface can be used to access out−of−band data for communication with other web service architectures and to manage conversations.

Syntax

```
public interface JwsContext extends Control
```

Package

```
weblogic.jws.control.JwsContext
```

Remarks

A JwsContext instance named context is included by default in each web service you create with WebLogic Workshop.

Members

Methods

public void finishConversation()

Marks the current conversation instance to be destroyed after the currently executing method or event handler returns.

public String getCallbackPassword()

Gets the password used for callbacks to the client in the current conversation. The password may have been specified by the client as part of the callbackLocation conversation start SOAP header, or it may have been set by a call to JwsContext.setCallbackPassword().

public String getCallbackURL()

Gets the URL that will be used for callbacks to the client during the current conversation. The callback URL is specified by the client in the callbackLocation conversation start SOAP header or may be set by a call to JwsContext.setCallbackURL().

public String getCallbackUsername()

Gets the username used for callbacks to the client in the current conversation. The username may have been specified by the client as part of the callbackLocation conversation start SOAP header, or it may have been set by a call to JwsContext.setCallbackUsername().

public java.security.Principal getCallerPrincipal()

Returns the security principal associated with the current method invocation if authentication was performed.

public long getCurrentAge()

Gets the current age (in seconds) of the conversation.

public long getCurrentIdleTime()

Gets the number of seconds since the last client request or the last call to JwsContext.resetIdleTime() for the current conversation.

public org.w3c.dom.Element[] getInputHeaders()

Returns the SOAP headers that arrived with the current method invocation message.

public Logger getLogger(String name)

Gets an instance of a Logger class, which you can use to send messages from your code to a log file.

public long getMaxAge()

Gets the duration (in seconds) of the current maximum age for the conversation.

public long getMaxIdleTime()

Gets the current value of the maximum idle time for the conversation.

public ServiceHandle getService()

Returns the weblogic.jws.ServiceHandle for this instance of this web service. See ServiceHandle Interface.

public boolean getUnderstoodInputHeaders()

Returns the value most recently set by a call to JwsContext.setUnderstoodInputHeaders().

public boolean isCallerInRole(String rolename)

Returns true if the authenticated principal is within the specified security role.

public boolean isFinished()

Determines whether or not this conversation instance has had finish() called on it.

public void resetIdleTime()

Resets the current idle timer associated with the current conversation.

public void setCallbackLocation(URL url)

Sets callback location. The callback location is typically provided by the client in the callbackLocation conversation start SOAP header, but may be set explicitly using this method. The callback location must be set before attempting to send a callback to the client.

public void setCallbackLocation(String url)

Sets callback location. The callback location is typically provided by the client in the callbackLocation conversation start SOAP header, but may be set explicitly using this method. The callback location must be set before attempting to send a callback to the client.

public void setCallbackPassword(String password)

Sets the password used for callbacks to the client in the current conversation. Overrides the current password if it was specified by the client as part of the callbackLocation conversation start SOAP header.

public void setCallbackUsername(String username)

Sets the username used for callbacks to the client in the current conversation. Overrides the current username if it was specified by the client as part of the callbackLocation conversation start SOAP header.

public void setMaxAge(java.util.Date date)

Sets a new maximum age for the conversation to an absolute date.

public void setMaxAge(String duration)

Sets a new maximum age for the conversation using the duration string format.

public void setMaxIdleTime(long seconds)

Sets the maximum amount of time (in seconds) that the current conversation will idle before it expires.

public void setMaxIdleTime(String duration)

Sets the maximum amount of time (in Duration string format) that the current conversation will idle before it expires.

public void setOutputHeaders(org.w3c.dom.Element[] headers)

Set the SOAP headers to be sent with outgoing messages to the client.

public void setUnderstoodInputHeaders(boolean understood)

Call with true if all incoming SOAP headers were understood.

If any SOAP headers with mustUnderstand set are present and setUnderstoodInputHeaders() is not called with true, a SOAP fault will be returned to the client.

Fields

None.

Callbacks

public onException(Exception e, String methodName, Object[] args)

Received when a method marked as an operation throws an uncaught exception.

public onFinish(boolean expired)

Received when the current conversation is about to end.

Related Topics

Maintaining State with Conversations

JwsContext.finishConversation() Method

Marks the current conversation instance as requiring removal after the currently executing method or event handler returns.

Syntax

```
public void finishConversation()
```

Parameters

None.

Return Value

None.

Exceptions

None.

Remarks

Call this method to force removal of the conversation instance as an alternative to waiting for it to be removed by WebLogic Server when the conversation times out or its "finish" method is reached. Calling this method from within another method is equivalent to marking the calling method with the @conversation phase="finish" tag.

Related Topics

@jws:conversation Tag

Overview: Conversations

JwsContext.getCallbackLocation() Method

Gets the URL that will be used for callbacks during the current conversation.

Syntax

```
public String getCallbackLocation()
```

Parameters

None.

Return Value

The url that will be used for callbacks.

Exceptions

None.

Remarks

You can call getCallbackLocation() to retrieve the URL set with JwsContext.setCallbackLocation, or to discover the URL if it was set via SOAP headers.

Related Topics

JwsContext.setCallbackLocation(String) Method

JwsContext.setCallbackLocation(URL) Method

JwsContext.getCallbackPassword() Method

Gets the password used for callbacks to the client in the current conversation.

Syntax

```
public String getCallbackPassword()
```

Parameters

None.

Return Value

The password that will be used for callbacks to the client in the current conversation.

Exceptions

None.

Remarks

The password may have been specified by the client as part of the callbackLocation conversation start SOAP header, or it may have been set by a call to JwsContext.setCallbackPassword().

Related Topics

JwsContext Interface

JwsContext.getCallbackUsername() Method

JwsContext.getCallbackPassword() Method

JwsContext.setCallbackPassword(String) Method

Overview: Security

JwsContext.getCallbackUsername() Method

Gets the username used for callbacks to the client in the current conversation.

Syntax

```
public String getCallbackUsername()
```

Parameters

None.

Return Value

The username that will be used for callbacks to the client in the current conversation.

Exceptions

None.

Remarks

The username may have been specified by the client as part of the callbackLocation conversation start SOAP header, or it may have been set by a call to JwsContext.setCallbackUsername().

Related Topics

JwsContext Interface

JwsContext.setCallbackUsername(String) Method

JwsContext.getCallbackPassword() Method

JwsContext.setCallbackPassword(String) Method

Overview: Security

JwsContext.getCallerPrincipal() Method

Returns the security principal associated with the current method invocation if authentication was performed.

Syntax

```
public java.security.Principal getCallerPrincipal()
```

Parameters

None.

Return Value

The Principal that was produced by authentication.

Exceptions

None

Remarks

To learn about security in WebLogic Workshop web services, see Overview: Security.

Related Topics

JwsContext Interface

JwsContext.isCallerInRole(String) Method

JwsContext.getCurrentAge() Method

Gets the age of the current conversation in seconds.

Syntax

```
public long getCurrentAge()
```

Parameters

None.

Return Value

The number of seconds that have passed since the conversation was started.

Exceptions

IllegalStateException

Thrown if the method is called from a service instance that is not conversational.

Remarks

None.

Related Topics

JwsContext.setMaxAge(Date) Method

JwsContext.setMaxAge(String) Method

JwsContext.getMaxAge() Method

Managing Conversation Lifetime

JwsContext.getCurrentIdleTime() Method

Gets the number of seconds since the last activity for the conversation.

Syntax

```
public long getCurrentIdleTime()
```

Parameters

None.

Return Value

The number of seconds since the last activity affecting the conversation.

Exceptions

IllegalStateException

Thrown if the method is called from a service instance that is not conversational.

Remarks

The conversation's idle time may be the time since the last conversational activity, such as the last client request. The idle time may also be the time since JwsContext.resetIdleTime() was last called.

Related Topics

JwsContext.resetIdleTime() Method

JwsContext.setMaxIdleTime(long) Method

JwsContext.setMaxIdleTime(String) Method

JwsContext.getMaxIdleTime() Method

Managing Conversation Lifetime

JwsContext.getInputHeaders() Method

Returns the SOAP headers that arrived with the current method invocation message.

Syntax

```
public org.w3c.dom.Element[] getInputHeaders()
```

Parameters

None.

Return Value

An array of org.w3c.dom.Element objects containing the SOAP headers that arrived with the current method invocation.

Exceptions

None.

Remarks

The SOAP headers used by WebLogic Workshop to manage conversations are included in the list of headers returned.

Related Topics

JwsContext Interface

JwsContext.getUnderstoodInputHeaders() Method

JwsContext.setUnderstoodInputHeaders(boolean) Method

JwsContext.setOutputHeaders(Element[]) Method

JwsContext.getLogger(String) Method

Gets an instance of the Logger class, which you can use to send messages from your code to a log file.

Syntax

```
public Logger getLogger(String categoryName)
```

Parameters

categoryName

The name of the category by which log messages should be grouped.

Return Value

A Logger class that may be used to send messages to the application log.

Exceptions

None.

Remarks

Use getLogger to log messages from your web service Java code to a text file. By default, for the domain in which you develop and debug WebLogic Workshop web services, this text file is located at the following path of your WebLogic Workshop installation:

BEA_HOME/weblogic700/samples/workshop/jws.log

Use the categoryName parameter to specific text that will be included with log entries. For example, you might specify the name of the JWS file so that you can more easily find relevant messages when scanning the log file. A log message my appear as follows for an entry in which categoryName is "MyService".

```
16:18:11 ERROR MyService    : My log message.
```

Note: You can customize aspects of the logging configuration, including the name of the application log file, its size limit, and so on. You configure logging using the workshopLogCfg.xml file. For more information, see workshopLogCfg.xml Configuration File.

The Logger class returned by the getLogger method includes four methods that you can use to print log entries to a text file. Each of the methods is available in two variations −− one that simply sends a message and another that sends an message and the contents of an exception or error (in effect, any class that inherits from Throwable).

debug Method

```
void debug(String)
void debug(String, Throwable)
```

Not surprisingly, you should use the debug method to log messages for debugging. These are debugging messages that you might otherwise write to a command console using System.out.println(). The advantage of debugging using logs is that logged messages can be kept and stored; with a console, you must copy and paste messages into a text file to store them.

Before deploying your web service, however, you should either remove code for debugging or ensure that the production service is configured not to log debugging messages. Logging messages for debugging can unnecessarily bloat a message log.

info Method

```
void info(String)
void info(String, Throwable)
```

Use the info method to log informational messages that highlight the progress of your web service.

warn Method

```
void warn(String)
void warn(String, Throwable)
```

Use the warn method to log messages about potentially harmful situations.

error Method

```
void error(String)
void error(String, Throwable)
```

Use the error method to log error events that might still allow the service to continue running.

Note: The Logger class returned by the getLogger method is part of the log4j package available from the Jakarta project. For general information about log4j, see log4j project. For information about the log4j application programming interface (API), see Short Introduction to log4j.

Related Topics

workshopLogCfg.xml Configuration File

JwsContext.getMaxAge() Method

Gets the time representing the longest the conversation may remain active before finishing.

Syntax

```
public long getMaxAge()
```

Parameters

None.

Return Value

The number of seconds since a conversation started before which it will finish.

Exceptions

IllegalStateException

Thrown if the method is called from a service instance that is not conversational.

Remarks

None.

Related Topics

JwsContext.setMaxAge(Date) Method

JwsContext.setMaxAge(String) Method

Managing Conversation Lifetime

JwsContext.getMaxIdleTime() Method

Gets the number of seconds that the conversation can remain idle before finishing due to client inactivity.

Syntax

```
public long getMaxIdleTime()
```

Parameters

None.

Return Value

The number of seconds that the conversation can remain idle before finishing due to client inactivity.

Exceptions

IllegalStateException

Thrown if the method is called from a service instance that is not conversational.

Remarks

A conversation is idle if the service is not receiving incoming messages through an operation method. Note that messages received through callback handlers do not reset a conversation idle time.

Related Topics

JwsContext.resetIdleTime() Method

JwsContext.setMaxIdleTime(long) Method

JwsContext.setMaxIdleTime(String) Method

Managing Conversation Lifetime

JwsContext.getService() Method

Returns the weblogic.jws.ServiceHandle for this instance of this web service.

Syntax

```
public ServiceHandle getService()
```

Parameters

None.

Return Value

A weblogic.jws.ServiceHandle object representing the current web service.

Exceptions

None.

Remarks

The ServiceHandle may be used to obtain information such as the current conversation ID.

Related Topics

JwsContext Interface

ServiceHandle Interface

JwsContext.getUnderstoodInputHeaders() Method

Returns the value most recently set by a call to JwsContext.setUnderstoodInputHeaders().

Syntax

```
public boolean getUnderstoodInputHeaders();
```

Parameters

None.

Return Value

The value most recently set by a call to JwsContext.setUnderstoodInputHeaders(). false if JwsContext.setUnderstoodInputHeaders() has not been called.

Exceptions

None.

Remarks

None.

Related Topics

JwsContext Interface

JwsContext.getInputHeaders() Method

JwsContext.setUnderstoodInputHeaders(boolean) Method

JwsContext.isCallerInRole(String) Method

Returns true if the authenticated principal is within the specified security role.

Syntax

```
public boolean isCallerInRole(String roleName)
```

Parameters

roleName

The name of the security role against which to check the authenticated principal.

Return Value

true if the authenticated principal is in the security role; otherwise false.

Exceptions

None.

Remarks

None.

Related Topics

JwsContext Interface

JwsContext.getCallerPrincipal() Method

Overview: Security

JwsContext.isFinished Method

Determines whether or not this conversation instance has finished through a call to the JwsContext.finishConversation() method.

Syntax

```
public boolean isFinished()
```

Parameters

None.

Return Value

true if the conversation has finished; false if it has not.

Exceptions

None.

Remarks

You can finish a conversation by calling the JwsContext.finishConversation() method on the conversation instance.

Related Topics

JwsContext.finishConversation() Method

JwsContext onFinish

Managing Conversation Lifetime

JwsContext.onException Callback

Received when a method marked as an operation throws an uncaught exception.

Syntax

```
public void onException(Exception e, String methodName, Object[] arguments)
```

Parameters

e

The exception object thrown from the method.

methodName

The name of the method from which the exception was thrown.

arguments

An array containing the parameters of the method that threw the exception.

Return Value

None.

Exceptions

None.

Remarks

Implement a handler for this callback as a single place to catch exceptions thrown from operation methods.

You should not write code to invoke this callback. Rather, you implement a callback handler that will execute when your service receives the callback from WebLogic Server. Your code might look like the following:

```
public void context_onException(Exception e, String methodName, Object[] arguments)
{
    /* The following code might be used for debugging. Prior for deployment, you
    might replace it with code that logs this information instead. Notice that this
    code also ends the conversation in which the service might have been
    participating. This frees unneeded resources. */
    System.out.println("MyService: exception in " + methodName + "(" +
        arguments + "). Exception: " + e);
    context.finishConversation();
    }
}
```

For more information on using this callback, see How Do I: Handle Errors in a Web Service?

Note that not all methods in a web service are necessarily operations. In web services, an operation is a method specifically exposed to clients. In WebLogic Workshop, the source code of an operation is preceded by a @jws:operation annotation.

Related Topics

How Do I: Handle Errors in a Web Service?

JwsContext Interface

JwsContext.onFinish(boolean) Callback

Received when the current conversation is about to end.

Syntax

```
public void onFinish(boolean expired)
```

Parameters

expired

true if the conversation ended by expiring; otherwise false.

Return Value

None.

Exceptions

None.

Remarks

Implement a handler for this callback if you want to add code that executes when a conversation is about to finish. For example, if you think your service's conversation may end before the service has returned a result to a client, you might want to implement an onFinish callback handler that lets the client know a response isn't coming. The conversation finishes after your callback handler has finished executing.

The expired value indicates whether the conversation finished by expiring (in effect, "timing out") or was intentionally finished in keeping with your service's design. Under normal circumstances a conversation ends for the following reasons:

- Your service's execution reaches some item (such as a callback or a method) that is marked to finish the conversation adov−−> its conversation property phase attribute is set to "finish."
- Your code calls the JwsContext.finishConversation method. You might do this to "clean up" in error handling code.

In other cases a conversation may not finish by design. If it does not, it may simply continue until it is ended by WebLogic Server. It will be ended if:

- The conversation outlives the value of the service's conversation−lifetime property max−age attribute.
- The conversation has remained idle longer than the service's conversation−lifetime property max−idle−time attribute.

Note that you should not write code to invoke this callback. Rather, you implement a callback handler that will execute when your service receives the callback from WebLogic Server. Your code might look like the following:

```
public void context_onFinish(boolean expired)
{
    /* If the conversation has ended because it expired, tell the client
    that results will not be coming. */
    if(expired){
        callback.onError("Results unavailable at this time. Please try again.");
    }
}
```

Related Topics

JwsContext Interface

Managing Conversation Lifetime

JwsContext.resetIdleTime() Method

Resets the timer measuring the number of seconds since the last activity for the current conversation.

Syntax

```
public void resetIdleTime()
```

Parameters

None.

Return Value

The number of seconds before which the conversation will be finished due to client inactivity.

Exceptions

IllegalStateException

Thrown if the method is called from a service instance that is not conversational.

Remarks

[None.]

Related Topics

JwsContext.setMaxIdleTime(long) Method

JwsContext.setMaxIdleTime(String) Method

JwsContext.getMaxIdleTime() Method

Managing Conversation Lifetime

JwsContext.setCallbackLocation(String) Method

Sets the URL that will be used for callbacks to the client during the current conversation.

Syntax

```
public void setCallbackLocation(String url)
```

Parameters

url

The url to use for callbacks.

Return Value

None.

Exceptions

MalformedURLException

Thrown if a malformed URL is provided. Either no legal protocol could be found in the specification string or the string could not be parsed.

Remarks

The callback location is typically provided by the client in the callbackLocation conversation start SOAP header, but may be set explicitly using this method. The callback location must be set before attempting to send a callback to the client. JwsContext.getCallbackLocation() may be used to retrieve the URL set by this method or to discover the callback URL if it was set via the SOAP header.

Credentials may be provided as part of the URL (if supported by the scheme) or by using the JwsContext.setCallbackUsername() and JwsContext.setCallbackPassword() methods.

Related Topics

JwsContext.getCallbackLocation() Method

JwsContext.setCallbackLocation(URL) Method

JwsContext.setCallbackLocation(URL) Method

Sets the URL that will be used for callbacks to the client during the current conversation.

Syntax

```
public void setCallbackLocation(URL url)
```

Parameters

url

The url to use for callbacks.

Return Value

None.

Exceptions

MalformedURLException

Thrown if a malformed URL is provided. Either no legal protocol could be found in the specification string or the string could not be parsed.

Remarks

The callback location is typically provided by the client in the callbackLocation conversation start SOAP header, but may be set explicitly using this method. The callback location must be set before attempting to send a callback to the client. JwsContext.getCallbackLocation() may be used to retrieve the URL set by this method or to discover the callback URL if it was set via the SOAP header.

Credentials may be provided as part of the URL (if supported by the scheme) or by using the JwsContext.setCallbackUsername() and JwsContext.setCallbackPassword() methods.

Related Topics

JwsContext.getCallbackLocation() Method

JwsContext.setCallbackLocation(URL) Method

JwsContext.setCallbackPassword(String) Method

Sets the password used for callbacks to the client in the current conversation.

Syntax

```
public void setCallbackPassword(String password)
```

Parameters

password

The password that will be sent with client callbacks.

Return Value

None.

Exceptions

None.

Remarks

Overrides the current password if it was specified by the client as part of the callbackLocation conversation start SOAP header.

Related Topics

JwsContext Interface

JwsContext.getCallbackPassword() Method

JwsContext.getCallbackUsername() Method

JwsContext.setCallbackUsername(String) Method

JwsContext.setCallbackUsername(String) Method

Sets the username used for callbacks to the client in the current conversation.

Syntax

```
public void setCallbackUsername(String username)
```

Parameters

username

The username that will be sent with client callbacks.

Return Value

None.

Exceptions

None.

Remarks

Overrides the current username if it was specified by the client as part of the callbackLocation conversation start SOAP header.

Related Topics

JwsContext Interface

JwsContext.getCallbackPassword() Method

JwsContext.getCallbackUsername() Method

JwsContext.setCallbackPassword(String) Method

JwsContext.setMaxAge(Date) Method

Sets a conversation's maximum age by specifying a time in seconds.

Syntax

```
public void setMaxAge(java.util.Date date)
```

Parameters

date

The time after which the conversation will finish. If the date value is null, the age timeout will be disabled.

Return Value

None.

Exceptions

IllegalStateException

Thrown if the method is called from a service instance that is not conversational.

IllegalArgumentException

Thrown to indicate that a method has been passed an illegal or inappropriate argument.

Remarks

Calling the setMaxAge method from your service's code sets the maximum conversation age (relative to the current time) for that service instance. The date parameter marks the time after which the WebLogic Server will be allowed to finish this conversation. Note that setting the maximum age with the setMaxAge method overrides the default setting or the setting given in the service's JWS file. This is an absolute age that isn't affected by network traffic.

If the date value results in zero seconds, the maximum age timeout will be disabled. If the date value is in the past, the conversation will finish immediately.

Related Topics

JwsContext.setMaxAge(String) Method

JwsContext.getMaxAge() Method

JwsContext.getCurrentAge() Method

Managing Conversation Lifetime

JwsContext.setMaxAge (String) Method

Sets a conversation's maximum age by specifying a duration as a string.

Syntax

```
public void setMaxAge(String duration)
```

Parameters

duration

The period after which the conversation will finish.

Return Value

None.

Exceptions

IllegalStateException

Thrown if the method is called from a service instance that is not conversational.

IllegalArgumentException

Thrown to indicate that a method has been passed an illegal or inappropriate argument.

Remarks

Call the setMaxAge method to sets the maximum conversation age (relative to the current time) for that conversation. The duration parameter marks the time after which the WebLogic Server will be allowed to finish this conversation. Note that setting the maximum age with the setMaxAge method overrides the default setting or the setting given in the service's JWS file. This is an absolute age that isn't affected by network traffic.

If the duration value results in zero seconds, the maximum age timeout will be disabled. If the duration value is in the past, the conversation will finish immediately.

Related Topics

JwsContext.setMaxAge(Date) Method

JwsContext.getMaxAge() Method

JwsContext.getCurrentAge() Method

Managing Conversation Lifetime

JwsContext.setMaxIdleTime(long) Method

Sets the number of seconds that the conversation can remain idle before finishing due to client inactivity.

Syntax

```
public void setMaxIdleTime(long seconds)
```

Parameters

seconds

The number of seconds the conversation can remain idle before it will expire.

Return Value

None.

Exceptions

IllegalStateException

Thrown if the method is called from a service instance that is not conversational.

IllegalArgumentException

Thrown to indicate that a method has been passed an illegal or inappropriate argument.

Remarks

Conversation idle time is the amount of time that can pass between incoming messages before the service instance is finished due to client inactivity. You can initialize a conversation's idle time to a default through the setting given at the top of the JWS file, but each instance can be set to a different idle time value through the setMaxIdleTime method.

To disable idle time expiration, set the maximum idle time value to zero.

Related Topics

JwsContext.resetIdleTime() Method

JwsContext.setMaxIdleTime(String) Method

JwsContext.getMaxIdleTime() Method

Managing Conversation Lifetime

JwsContext.setMaxIdleTime(String) Method

Sets the time (in seconds) that the conversation can remain idle before finishing due to client inactivity.

Syntax

```
public void setMaxIdleTime(String duration)
```

Parameters

duration

The number of seconds the conversation can remain idle before it will expire.

Return Value

None.

Exceptions

IllegalStateException

Thrown if the method is called from a service instance that is not conversational.

IllegalArgumentException

Thrown to indicate that a method has been passed an illegal or inappropriate argument.

Remarks

Conversation idle time is the amount of time that can pass between incoming messages before the service instance is finished due to client inactivity. You can initialize a conversation's idle time to a default through the setting given at the top of the JWS file, but each instance can be set to a different idle time value through the setMaxIdleTime method.

To disable idle time expiration, set the maximum idle time value to zero.

Related Topics

JwsContext.resetIdleTime() Method

JwsContext.setMaxIdleTime(long) Method

JwsContext.getMaxIdleTime() Method

Managing Conversation Lifetime

JwsContext.setOutputHeaders(Element[]) Method

Set the SOAP headers to be sent with outgoing messages to the client.

Syntax

```
public void setOutputHeaders(org.w3c.dom.Element[] headers)
```

Parameters

headers

An array of org.w3c.dom.Element objects containing valid SOAP headers.

Return Value

None.

Exceptions

None.

Remarks

None.

Related Topics

JwsContext Interface

JwsContext.getInputHeaders() Method

JwsContext.setUnderstoodInputHeaders(boolean) Method

Call with true if all incoming SOAP headers were understood.

Syntax

```
public void setUnderstoodInputHeaders(boolean understood)
```

Parameters

understood

true if all required input headers were understood, false otherwise.

Return Value

None.

Exceptions

None.

Remarks

If any SOAP headers with mustUnderstand set are present and setUnderstoodInputHeaders() is not called with true, a SOAP fault will be returned to the client.

Related Topics

JwsContext Interface

JwsContext.getUnderstoodInputHeaders() Method

JwsContext.getInputHeaders() Method

ServiceControl Interface

Base interface for all Service controls.

Syntax

```
package weblogic.jws.control;


public interface ServiceControl extends Control
```

Remarks

The methods of this interface may be invoked by any web service with a Service control instance.

Members

Constructors

String getConversationID()

Retreives the conversation ID of the current conversation with this Service control instance.

Element[] getInputHeaders()

Retrieves the SOAP headers that were included in the most recent arriving callback from the Service control.

String getPassword()

Retrieves the password string that was set by the most recent call to setPassword.

String getUsername()

Retrieves the username string that was set by the most recent call to setUsername.

void reset()

Clears all parameters that were set by previous calls to setConversationID, setOutputHeaders, setPassword or setUsername.

void setConversationID(String conversationID)

Sets the unique key that will be proposed as the conversation ID when initiating a conversation with the Service control.

Note: WebLogic Workshop automatically computes a conversation ID when a WebLogic web service invokes a start method of a Service control. You may use this function to override the automatic value. The only case where it is useful to do so is if you supply the conversation ID of an existing conversation that is currently ongoing on the target web service. You may then invoke methods on the target service that will execute in the context of the specified conversation. However, only the client that originated the conversation may receive callbacks.

void setEndPoint(URL url)

Sets the callback URL that the Service control instance will use it as the base URL for callback invocations. This is set automatically by WebLogic Workshop. You may use this function to override the callback URL if you wish callbacks to be sent to a different destination.

void setOutputHeaders(Element[] headers)

Sets the SOAP headers that will be included in the next outgoing method invocation message to the Service control.

void setPassword(String password)

Sets the password that will be sent with the next outgoing Service control method invocation. Used if the Service control uses HTTP basic authentication.

void setUsername(String username)

Sets the username that will be sent with the next outgoing Service control method invocation. Used if the Service control uses HTTP basic authentication.

Fields

None.

Events

None.

Related Topics

Service Control: Using Another Web Service

ServiceHandle Interface

Represents a reference to a web service (instance of a JWS file). This interface can also refer to a Service control instance that is associated with a JWS service.

Syntax

```
public interface ServiceHandle extends java.io.Serializable
```

Package

```
package weblogic.jws
```

Remarks

None.

Members

Constructors

None.

Methods

public String getConversationID()

Returns the identity of a control instance that the handle refers to. This may be null if the handle is associated with a service instance instead of a control instance.

public String getJNDIBaseName()

Returns a period−separated string based on the URI, which uniquely identifies this service on this server and is used to generate unique JNDI names for objects associated with this service.

public int getScheme()

Returns the protocol scheme that was used to construct this ServiceHandle. The value returned is one of the constants defined in Fields below.

public String getURI()

Returns a protocol−indendent URI which can be used to refer to this service.

public URL getURL()

Returns a URL which defines a reference to this service using the same scheme that was used to construct this service handle.

public URL getURL(int scheme)

Returns a URL which defines a reference to this service using the specified scheme. The scheme argument is one of the constants defined in Fields below.

Fields

public final static int SCHEME_DEFAULT = 0

Indicates the default scheme shall be used. This value is only useful as an argument to getURL(int scheme).

public final static int SCHEME_HTTP = 1

indicates the http: scheme.

public final static int SCHEME_JMS = 2

Indicates the jms: scheme.

public final static int SCHEME_SMTP = 3

Indicates the smtp: scheme.

public final static int SCHEME_FTP = 4

Indicates the ftp: scheme.

public final static int SCHEME_FILE = 5

Indicates the file: scheme.

Callbacks

None.

Related Topics

JwsContext Interface

SessionEJBControl Interface

Base interface for EJB controls that represent Session Beans. Subclass of EJBControl.

Syntax

```
package weblogic.jws.control;

public class SessionEJBControl extends EJBControl
```

Remarks

The methods of this interface may be invoked by any web service with a EJB control instance.

Members

Constructors

none.

Methods

For inherited methods, see EJBControl Interface.

Fields

none.

Events

none.

Related Topic

Overview: Enterprise Java Beans (EJBs)

EJB Control: Using Enterprise Java Beans from a Web Service

TimerControl Interface

Base interface for all Timer controls.

Syntax

```
package weblogic.jws.control;

public class TimerControl extends Control
```

Remarks

The methods of this interface may be invoked by any web service with a Timer control instance.

Members

Constructors

None.

Methods

boolean getCoalesceEvents()

Returns the current value of the coalesce−events attribute.

String getRepeatsEvery()

Returns the current interval specified by the repeats−every attribute of the @jws:timer tag or the most recent call to setRepeatsEvery.

long getTimeout()

Returns the current duration specified by the timeout attribute of the @jws:timer tag of the most recent call to setTimeout.

Date getTimeoutAt()

Returns the time specified by the most recent call to setTimeoutAt, or null.

void restart()

Resets the timer. Any pending event(s) is canceled. The timer will subsequently expire after the repeats−every period has elasped after this call.

void setCoalesceEvents(boolean coalesce)

Enables or disables the coalesce−events behavior. See Creating a New Timer Control.

void setRepeatsEvery(long seconds)

Sets the repeat interval for the timer using seconds since the epoch.

void setRepeatsEvery(String interval)

Sets the repeat interval using an xsd:duration string. See Specifying Time on a Timer Control.

void setTimeout(long seconds)

Sets the time between start or restart and the first expiration of the timer, in seconds.

void setTimeout(String delay)

Sets the time between start or restart and the first expiration of the timer, as an xsd:duration string. See Specifying Time on a Timer Control.

void setTimeoutAt(Date time)

Sets the absolute date and time at which the timer will expire the first time after being started or restarted.

void start()

Starts the timer. The first timer expiration will occur after the period specified by timeout has elasped.

void stop()

Stops the timer. No further timer expiration callbacks will be invoked.

Fields

none.

Events

void onTimeout(long time)

When the timer expires, a callback handler with the name <timer instance>_onTimeout, if present, is invoked. The time at which the timer expired is passed as the time parameter. Note that his may be some time in the past if the onTimeout callback could not be invoked due to high system load.

Related Topics

Timer Control: Using Timers in Your Web Service

SimpleTimer.jws Sample

AdvancedTimer.jws Sample

Javadoc Tag Reference

This section provides reference information about WebLogic Workshop's Javadoc tags.

WebLogic Workshop provides a set of custom tags based on Javadoc technology. Originally developed as a way to embed documentation into source code as comments, Javadoc is extended by WebLogic Workshop through custom tags that help to define a service's functionality. Much of the power of WebLogic Workshop Web Services is accessed via these tags, which are placed in your JWS and CTRL files.

For overview information about the tags, and how they are used within web services built with WebLogic Workshop, see Using the WebLogic Workshop Javadoc Tags.

Tags

@jws:av−identity Tag

Specifies the target Application View for an Application View control.

@jws:av−service Tag

Specifies the Application View service associated with a method of an Application View control.

@jws:connection Tag

Specifies the data source used by a Database control.

@jws:control Tag

Specifies that the object annotated by this tag is a WebLogic Workshop control.

@jws:conversation Tag

Specifies the role a method or callback plays in a service's conversations.

@jws:context Tag

Specifies that the object annotated by this tag is a JwsContext object.

@jws:define Tag

Defines inline data that might otherwise be referenced as an external file.

@jws:ejb Tag

Specifies the target EJB's home interface for an EJB control.

@jws:parameter−xml Tag

Specifies conversion between XML messages and a Java method's parameters.

@jws:jms Tag

Specifies the configuration attributes of a JMS control.

@jws:jms−header Tag

Specifies XML maps for headers of messages processed by a JMS control.

@jws:jms−message Tag

Specifies XML maps for the message body of messages processed by a JMS control.

@jws:jms−property Tag

Specifies XML maps for properties of messages processed by a JMS control.

@jws:conversation−lifetime Tag

Specifies the maximum age and/or the maximum idle time for a service's conversations.

@jws:message−buffer Tag

Specifies that there should be a queue between the service's implementation code and the message transport mechanism for the specified method.

@jws:operation Tag

Specifies that the associated method is part of the web service's public contract and is available for clients to invoke.

@jws:protocol Tag

Specifies which protocols and message formats a web service can accept or a Service control will send to the service it represents.

@jws:return−xml Tag

Specifies conversion between XML messages and a Java method's return value.

@jws:location Tag

Specifies the URL at which a service accepts requests for each supported protocol.

@jws:schema Tag

Specifies the schema file whose types are referenced in the @jws:parameter−xml and @jws:return−xml tags elsewhere in a JWS or CTRL file.

@jws:sql Tag

Specifies the SQL statement and associated attributes that correspond to a method in a Database control.

@jws:target−namespace Tag

Specifies the default XML namespace used for outgoing XML messages and generated WSDL files.

@jws:timer Tag

Specifies configuration attributes for a Timer control.

@jws:wsdl Tag

Specifies a WSDL file that is implemented by a web service or represented by a Service control.

@jws:xmlns Tag

Defines an XML namespace prefix for use elsewhere in a JWS file or Service control CTRL file.

A CTRL file holds a WebLogic Workshop Java Web Service Invoker. A CTRL file reflects the public contract of a web service, and reflects nothing about the private implementation. For example, if the web service refers to complex implementation classes, the CTRL file will not contain references to the same classes. Instead, it will refer to generated inner classes or other generic types.

@jws:av−identity Tag

Specifies the target Application View for an Application View control.

Syntax

```
@jws:av−identity
name="applicationViewName"
 user−id[="username"]
 password[="password"]
```

Attributes

name

Required. Specifies the name of the target Application View. The Application View must be deployed before the Application View control will function.

user−id

Optional. Specifies the username to use when accessing the target Application View.

password

Optional. Specifies the password to use when accessing the target Application View.

Remarks

The following rules apply to this tag's use:

- Only one @jws:av−identity tag may appear within a single Javadoc comment block.
- The @jws:av−identity tag may appear in the Javadoc comment on the main interface defined in a Application View control's CTRL file.

Related Topics

Application View Control: Accessing an Enterprise Application from a Web Service

@jws:av−service Tag

@jws:av−service Tag

Specifies the Application View service associated with a method of an Application View control.

Syntax

```
@jws:av−service
name="avServiceName"
```

Attributes

name

Required. Specifies the name of the service in the target Application View with which this Application View control method is associated.

Remarks

The following rules apply to this tag's use:

- The @jws:av−service tag may only occur on a method declaration in an interface that extends weblogic.jws.control.ApplicationViewControl.

- The @jws:av−service tag may only occur in a CTRL file.
- The @jws:av−service tag may only appear once per method.

Related Topics

Application View Control: Accessing an Enterprise Application from a Web Service

@jws:av−identity Tag

@jws:connection Tag

Specifies the data source used by a Database control.

Syntax

```
@jws:connection    data−source−jndi−name="JNDINameofDataSource"
```

Attributes

data−source−jndi−name

Required. The JNDI name of a defined data source. Default: null.

Remarks

The following rules apply to this tag's use:

- Only one @jws:connection tag may appear within a single Javadoc comment block.
- The @jws:connection tag must appear in the Javadoc comment on the main interface defined in a Database control's CTRL file.

JNDI is the Java Naming and Directory Interface. data−source−jndi−name must be a valid name of a data source as it appears in the local JNDI registry.

To learn how to create, configure and register a new data source, please consult the WebLogic Server documentation on JDBC Data Sources.

Two data sources are pre−configured when WebLogic Workshop is installed. Both operate on the default PointBase database system by default. The pre−configured data source names are:

- cgSampleDataSource: the default data source name that is the initial value for @jws:connection when a new Database control is created. This data source is intended for experimentation. It is used by all Database controls in the samples project.
- cgDataSource: a data source used by WebLogic Workshop for internal bookkeeping. You should not perform any database activities against this data source. When WebLogic Server web services are deployed to a production environment, cgDataSource should be redirected to a more robust database system.

Related Topics

Database Control: Using a Database from Your Web Service

How Do I: Connect a Database Control to a Database Such as SQL Server or Oracle?

How Do I: Configure WebLogic Workshop to Use a Different Database for Internal State?

How Do I: WebLogic Workshop−Enable an Existing WebLogic Server Domain?

@jws:sql Tag

A JWS file holds a WebLogic Workshop Java Web Service. JWS files contain syntactically correct Java code. However, they may also contain special annotation that allows the web services to take advantage of powerful features of the WebLogic Workshop runtime and the WebLogic Server. All JWS files contain a class derived from a special base class called Service that makes certain helper resources available to the class at runtime. The goal of all the helper code is to make it really easy to put a Java− and−J2EE−based service on the net with standard protocols.

@jws:context Tag

Specifies that the object annotated by this tag is a JwsContext object. Causes WebLogic Server to connect the associated JwsContext object to its underlying infrastructure.

Syntax

```
@jws:context
```

Attributes

none

Remarks

The following rules apply to this tag's use:

- Only one @jws:context tag may appear within a single Javadoc comment block.
- Must appear on the JwsContext instance declaration in a JWS file, if present.

If the @jws:context tag is not present on a JwsContext instance declaration, the JwsContext object will not be connected to its underlying infrastructure. Attempts to invoke methods of th JwsContext object will result in Null Pointer Exceptions (NPEs).

Note: All @jws tags are Javadoc tags. Javadoc tags are not recognized unless they are in a Javadoc comment, which must begin with /** (two asterisks).

Related Topics

JwsContext Interface

Structure of a JWS File

A JWS file holds a WebLogic Workshop Java Web Service. JWS files contain syntactically correct Java code. However, they may also contain special annotation that allows the web services to take advantage of powerful features of the WebLogic Workshop runtime and the WebLogic Server. All JWS files contain a class derived from a special base class called Service that makes certain helper resources available to the class at runtime.

The goal of all the helper code is to make it really easy to put a Java− and−J2EE−based service on the net with standard protocols.

@jws:control Tag

Specifies that the object annotated by this tag is a WebLogic Workshop control. Causes the object to be displayed as a control in Design View and causes WebLogic Server to connect the control to its underlying infrastructure.

Syntax

```
@jws:control
```

Attributes

none

Remarks

The following rules apply to this tag's use:

- Only one @jws:control tag may appear within a single Javadoc comment block.
- Must appear on each control instance declaration in a JWS file.

If the @jws:control tag is not present on a control instance declaration, the control will not be connected to its underlying infrastructure and therefore the control will not operate. Attempts to invoke control methods will result in Null Pointer Exceptions (NPEs).

Note: All @jws tags are Javadoc tags. Javadoc tags are not recognized unless they are in a Javadoc comment, which must begin with /** (two asterisks).

Related Topics

Controls: Using Resources from a Web Service

Structure of a JWS File

@jws:conversation Tag

Specifies the role a method or callback plays in a service's conversations. Note that these roles are different between methods and callbacks.

Syntax

```
@jws:conversation
        phase="none" | "start" | "continue" | "finish"
```

Attributes

phase

One of four values indicating the method or callback's role in conversations. Note that only continue and finish are available for callbacks.

Remarks

Set the conversation phase attribute for a method or callback to specify its role in the service's conversations.

The following rules apply to this tag's use:

- Only one @jws:conversation tag can appear within a single Javadoc comment block.
- Optionally may appear in front of an @jws:operation method in a JWS file.

Possible attribute values are as follows:

- "start" adov−−> May be applied to methods only. Specifies that a call to the method starts a new conversation. Each call will create a new conversation context and an accompanying unique conversation ID, save the service's state, and start its idle and age timer.

- "continue" adov−−> May be applied to methods and callbacks. Specifies that a call to this method or callback is part of a conversation in progress. WebLogic Server will attempt to use the incoming conversation ID to correlate each call to an existing conversation. Each call to a "continue" method will save the service's state and reset its idle timer.

Set the phase attribute to "continue" for any method or callback that is likely to be used for communication with a client in connection with an ongoing request. In particular, these are methods that are clearly intended to be called subsequent to the conversation's start and before its finish. These include requests for or responses with additional information, requests for progress or status, and so on.

- "finish" adov−−> May be applied to methods and callbacks. Specifies that a call to this method or callback finishes an existing conversation. A call will mark the conversation for destruction by WebLogic Server. At some point after a finish method returns successfully, all data associated with the conversation's context will be removed.

It is also possible to finish a conversation by calling the JwsContext interface finishConversation method. For more information, see Managing Conversation Lifetime.

- "none" adov−−> May be applied to methods only. Specifies that a call to this method does not participate in any conversation. Methods marked none should not attempt to access conversational state (member variables of the JWS class).

The phase attributes default values are as follows:

- When applied to methods, the phase attribute's default value is "none".
- When applied to callbacks, the phase attribute's default value is "continue".

Related Topics

Overview: Conversations

JwsContext Interface

@jws:conversation−lifetime Tag

Specifies the maximum age and/or the maximum idle time for a service's conversations.

Syntax

```
@jws:conversation−lifetime
[maxIdleTime="duration"]
[maxAge="duration"]
```

Attributes

maxIdleTime

The amount of time that the conversation may remain idle before it is finished by WebLogic Server. Note that only activity between a web service and its client will reset the idle timer; activity between a web service and controls does not reset the idle timer.

Default: 0 seconds. Conversations with 0−length idle timeout will never timeout due to inactivity.

maxAge

The amount of time (since it started) that the conversation may remain active before it is finished by WebLogic Server. Default: 1 day (24 hours).

Remarks

Use the conversation−lifetime tag to set initial values for maximum idle time and maximum age.

Assign values in the form of expressions of time: "2 minutes," "5 days," and so on.

You may also set values for this tag using the conversation−lifetime property in the Properties pane.

This tag must precede any method or class declarations in the JWS file.

You may also set these values using the setMaxIdleTime and setMaxAge methods of the JwsContext interface. See JwsContext Interface.

The idle timer may be reset using the resetIdleTime method of the JwsContext interface.

When a conversation is terminated due to expiration of the idle time or maximum age, the conversation termination never occurs during the execution of a web service method. The conversation will actually terminate after completion of the method that is executing when the conversation is terminated.

When a conversation is terminated, the optional JwsContext onFinish callback is invoked. You may implement a handler for this callback to receive notification of conversation expiration. See JwsContext Interface.

Related Topics

@jws:conversation Tag

Managing Conversation Lifetime

JwsContext Interface

A CTRL file holds a WebLogic Workshop Java Web Service Invoker. A CTRL file reflects the public contract of a web service, and reflects nothing about the private implementation. For example, if the web service refers to complex implementation classes, the CTRL file will not contain references to the same classes. Instead, it will refer to generated inner classes or other generic types.

@jws:define Tag

Defines inline data that might otherwise be referenced as an external file.

Syntax

```
@jws:define
name="nameOfInlineData"
value::
    data referred to by
    the name attribute
::
```

Attributes

name

Required. The name to use when referring to the data.

value

Required. The data referred to by the name. Can be a string that contains a multiline value delimited by :: (two colons) delimiters.

Remarks

The following rules apply to this tag's use:

- Optionally may appear in a comment at the end of a CTRL file.

The @jws:define tag is used to define inline data that might otherwise be referenced as an external file.

This is used, for example, to include relevant schema or WSDL files in a CTRL or JWS file.

Example

In the example CTRL file below, the @jws:define tag defines the name WorldpProxyWsdl to refer to the contents of a WSDL file. The entire contents of the WSDL file are included as the value of the @jws:define tag. The @jws:wsdl tag then references the name defined by the @jws:define tag.

```
/**
 * @jws:wsdl file="#WorldpProxyWsdl"
 */
public interface WorldpProxy extends ServiceControl
{
   ...
}
/** @jws:define name="WorldpProxyWsdl" value::
 *  <xml version="1.0" encoding="utf-8"?>
 *  <definitions xmlns:s="http://www.w3.org/2001/XMLSchema">
 *    ...< remainder of contents of WSDL file>...
 *  </definitions>
 *  ::
 */
```

Related Topics

[@jws:wsdl Tag]()

[CTRL Files: Implementing Controls]()

[WSDL Files: Web Service Descriptions]()

[Service Control: Using Another Web Service]()

@jws:ejb Tag

Specifies the target EJB's home interface for an EJB control.

Syntax

```
@jws:ejb
        home−jndi−name="JNDINameOfTargetEJB"]
```

Attributes

home−jndi−name

Required. Specifies the JNDI name of the target EJB's home interface. Typically "EJBNameHome".

Remarks

The following rules apply to this tag's use:

- Only one @jws:ejb tag may appear within a single Javadoc comment block.
- Must appear in front of the main interface definition in an EJB control's CTRL file.

The target EJB must be deployed in WebLogic Server in the same domain as WebLogic Workshop project containing the EJB control.

Related Topics

EJB Control: Using Enterprise Java Beans from a Web Service

A CTRL file holds a WebLogic Workshop Java Web Service Invoker. A CTRL file reflects the public contract of a web service, and reflects nothing about the private implementation. For example, if the web service refers to complex implementation classes, the CTRL file will not contain references to the same classes. Instead, it will refer to generated inner classes or other generic types.

@jws:jms Tag

Specifies the configuration attributes of a JMS control.

Syntax

```
@jws:jms
type="queue | topic"
 send-jndi-name="JNDInameOfSendDestination"]
 receive-jndi-name="JNDInameOfReceiveDestination"]
 connection-factory-jndi-name="JNDInameOfConnectionFactory"]
 send-correlation-property="JMSheaderOrPropertyFieldName"]
 receive-correlation-property="JMSheaderOrPropertyFieldName"]
 receive-selector="selectorString"]
```

Attributes

type

Required. Specifies the type of the destination: can be “queue” or “topic”.

send−jndi−name

Required. Specifies the JNDI name of the JMS destination (queue or topic) to which messages will be sent.

No default.

receive−jndi−name

Optional. Specifies the JNDI name of the JMS destination (queue or topic) from which messages will be received. Note that subscription to a topic also requires a call to the JMS control's subscribe method.

connection−factory−jndi−name

Optional. Specifies the JNDI name of the connection factory used to obtain JMS connections. If not specified the default WebLogic Workshop connection factory is used (cg.jms.QueueConnectionFactory).

send−correlation−property

Optional. Specifies the name of the JMS header or property field to which the WebLogic Workshop conversation ID is written in outgoing messages. If not specified, the default is JmsControl.HEADER_CORRELATIONID. See remarks below.

receive−correlation−property

Optional. Specifies the name of the JMS header or property field from which the WebLogic Workshop conversation ID is read in incoming messages. If not specified, the default is JmsControl.HEADER_CORRELATIONID. See remarks below.

receive−selector

Optional. Specifies a selector string used to select messages. This syntax is based on a subset of the SQL92 conditional expression syntax. To learn more about how to specify message selector strings, please consult the Filtering Messages section of Programming WebLogic JMS in the WebLogic Server documentation.

Remarks

The following rules apply to this tag's use:

- Only one @jws:jms tag may appear within a single Javadoc comment block.
- The @jws:jms tag may appear in the Javadoc comment on the main interface defined in a JMS control's CTRL file.

The send−correlation−property and receive−correlation−property attributes are set to default values that will provide proper operation in most cases. If the application with which the JMS control is communicating already uses the default property name ("CorrelationId") for other purposes, another property name should be used.

Since many web service instances can use JMS controls that listen on the same queue, the underlying dispatching mechanism needs to correlate each message received by the JMS control to the correct web service instance. The conversation ID is used as the correlation ID.

For this to work the application on the other end (the “foreign” application) has to participate in a simple correlation protocol. The JMS control will place the conversation ID in the JMS header identified by the send−correlation−property attribute on every message the JMS control publishes. The foreign application must place the conversation ID into a JMS header that is identified by the receive−correlation−property attribute in all messages that constitute responses.

A web service may not send messages via a JMS control from within a method with a conversation phase of "none". These methods have no conversation ID and therefore cannot participate in the protocol described above.

Note that conversational correlation is not supported for JMS topics. All current conversational instances that subscribe to a topic will receive a callback when a message arrives on the topic.

Related Topics

JMS Control: Using Java Message Service Queues and Topics from Your Web Service

@jws:jms−header Tag

@jws:jms−message Tag

@jws:jms−property Tag

@jws:jms−header Tag

In a JMS control, specifies the XML map used to translate between JMS control Java method and callback parameters and JMS message headers.

Syntax

```
@jws:jms-header
xml-map="JMSheaderXMLMap"
```

Attributes

xml−map

Required. Specifies an XML document fragment (an XML map), optionally containing substitution tokens enclosed in curly braces ("{}"). The XML map must be delimited by "::" and must include <header> as the outermost element.

No default.

Remarks

The following rules apply to this tag's use:

- Only one @jws:jws−header tag may appear within a single Javadoc comment block.
- Optionally may appear in front of a method in a CTRL file defining a JMS control.
- Optionally may appear in front of a callback in a CTRL file defining a JMS control.

On a method of a JMS control, tells the control to use the XML map to set values in the outgoing XML message headers from one or more method parameters.

On a callback of a JMS control, tells the control to use the XML map to set one or more callback parameters from values in the incoming XML message headers.

For an example, see [Sending and Receiving XML Messages with a JMS Control](#).

Example

The example below demonstrates use of the @jws:jms−header tag:

```
/**
 * @jws:jms-message xm-map::
 *   <person>
 *     <identifier>{personID}</identifier>
 *   </person>
 * ::
 *
 * @jws::jms-header xml-map::
 *   <header>
 *     <command>GetPerson</command>
 *   </header>
 * ::
 *
 */
public void sendID(String personID);
```

The message sent when sendID is invoked will include the JMS message header command with the (hard−coded) value GetPerson.

The outermost element must be <header>. Multiple headers may be included within the <header> element. Each member identifies a JMS message header with a name that is the name of the element name and a value that is the value of the element.

Individual property elements may not have child elements.

As with all XML maps, substitutions may be made using the {} syntax within the XML map. To learn about the general capabilities of XML maps, see Why Use XML Maps?

Related Topics

JMS Control: Using Java Message Service Queues and Topics from Your Web Service

Manipulating JMS Message Headers and Message Properties in a JMS Control

@jws:jms Tag

@jws:jms−message Tag

@jws:jms−property Tag

@jws:jms−message Tag

Specifies XML maps for the message body of messages processed by a JMS control.

Syntax

```
@jws:jms−message
xml−map="JMSmessageXMLMap"
```

Attributes

xml−map

Required. Specifies an XML document fragment (an XML map), optionally containing substitution tokens enclosed in curly braces ("{}"). The XML map must be delimited by "::". No default.

Remarks

The following rules apply to this tag's use:

- Only one @jws:jws−message tag may appear within a single Javadoc comment block.
- Optionally may appear in front of a method in a CTRL file defining a JMS control.
- Optionally may appear in front of a callback in a CTRL file defining a JMS control.

On a method of a JMS control, tells the control to use the XML map to set values in the outgoing XML message body from one or more method parameters.

On a callback of a JMS control, tells the control to use the XML map to set one or more callback parameters from values in the incoming XML message body.

For an example, see Sending and Receiving XML Messages with a JMS Control.

Example

The example below demonstrates use of the @jws:jms−message tag:

```
/**
 * @jws:jms-message xml-map::
 *   <person>
 *     <identifier>{personID}</identifier>
 *   </person>
 * ::
 */
public void sendID(String personID);
```

The body of the message sent when sendID is invoked will consist of the <person> XML document with the string value of personID substituted for {personID}.

Multiple properties may be included within the <property> element.

Individual property elements may not have child elements.

As with all XML maps, substitutions may be made using the {} syntax within the XML map. To learn about the general capabilities of XML maps, see Why Use XML Maps?

Related Topics

JMS Control: Using Java Message Service Queues and Topics from Your Web Service

Manipulating JMS Message Headers and Message Properties in a JMS Control

@jws:jms Tag

@jws:jms−header Tag

@jws:jms−property Tag

@jws:jms−property Tag

In a JMS control, specifies the XML map used to translate between JMS control Java method and callback parameters and JMS message properties.

Syntax

```
@jws:jms-property
xml-map="JMSpropertyXMLMap"
```

Attributes

xml−map

Required. Specifies an XML document fragment (an XML map), optionally containing substitution tokens enclosed in curly braces ("{}"). The XML map must be delimited by "::" and must include <property> as the outermost element.

No default.

Remarks

The following rules apply to this tag's use:

- Only one @jws:jws−property tag may appear within a single Javadoc comment block.
- Optionally may appear in front of a method in a CTRL file defining a JMS control.
- Optionally may appear in front of a callback in a CTRL file defining a JMS control.

On a method of a JMS control, tells the control to use the XML map to set properties in the outgoing XML message from one or more method parameters.

On a callback of a JMS control, tells the control to use the XML map to set one or more callback parameters from properties in the incoming XML message.

For an example, see Sending and Receiving XML Messages with a JMS Control.

Example

The example below demonstrates use of the @jws:jms−property tag:

```
/**
 * @jws:jms−message xm−map::
 *   <person>
 *     <identifier>{personID}</identifier>
 *   </person>
 * ::
 *
 * @jws::jms−property xml−map::
 *   <property>
 *     <command>GetPerson</command>
 *   </property>
 * ::
 *
 */
public void sendID(String personID);
```

The message sent when sendID is invoked will include the JMS message property command with the (hard−coded) value GetPerson.

The outermost element must be <property>. Multiple properties may be included within the <property> element. Each member identifies a JMS message property with a name that is the element name and a value that is the the value of the element.

Individual property elements may not have child elements.

As with all XML maps, substitutions may be made using the {} syntax within the XML map. To learn about the general capabilities of XML maps, see Why Use XML Maps?

Related Topics

JMS Control: Using Java Message Service Queues and Topics from Your Web Service

Manipulating JMS Message Headers and Message Properties in a JMS Control

@jws:jms Tag

@jws:jms−header Tag

@jws:jms−message Tag

@jws:location Tag

Specifies the URL at which a service accepts requests for each supported protocol.

Syntax

```
@jws:location
        [http-url="httpEndPoint"]
        [jms-url="jmsEndPoint"]
```

Attributes

htp−url

Optional on CTRL files; prohibited on JWS files. The URL of the target service's HTTP end point.

jms−url

Optional on CTRL files; prohibited on JWS files. The URL of the target service's JMS end point.

Remarks

The following rules apply to this tag's use:

When optionally applied to an interface in a CTRL file:

- Only one @jws:location tag may be present in the interface's Javadoc comment block.

Values for this tag's attributes are typically computed automatically when the web service is compiled or deployed. The values are derived from the deployment environment of the web service.

You may wish to change the attribute values if a web service or Service control is moved to a different file system location or host.

Related Topics

@jws:protocol Tag

Service Control: Using Another Web Service

@jws:message−buffer Tag

Specifies that there should be a queue between the service's implementation code and the message transport wire for the specified method.

Syntax

```
@jws:buffer


[enable="true | false"]


[retry−count="protocolName"]
[retry−delay="targetServiceEndPoint"]
```

Attributes

enable

Required. Specifies whether the messages arriving or departing via the associated method or callback are queued.

retryCount

Optional. The number of times WebLogic Server will attempt to deliver queued messages to the service.

retryDelay

Optional. The amount of time that should pass between message delivery attempts.

Remarks

The following rules apply to this tag's use:

- Only one @jws:buffer tag may appear within a single Javadoc block.
- Optionally may appear in front of a @jws:operation method in a JWS file.
- Optionally may appear in front of a callback method (event declaration) in a JWS file.
- Optionally may appear in front of an event handler method (control_eventName method) in a JWS file.
- Optionally may appear in front of a method declaration in a CTRL file.

The @jws:buffer tag may only be placed on methods or callbacks with return type void.

The @jws:buffer tag places a queue between the method and the communication wire. For incoming messages, this means that the message is queued before it is processed. For outgoing messages, this means that the message is queued before it is sent on the wire.

Note that the queue is transparent to the method that is marked with the @jws:buffer tag. When multiple messages arrive, they may be queued (or for the client, in the case of a callback) while the method is processing a previous message. Callers invoking a buffered method or callback will always experience an immediate return.

The retryDelay attribute can be specified using friendly names for units of time. For example, the default unit is seconds, and can be expressed as "30 seconds". The following are also possible values:

- 5 minutes
- 24 hours
- 3 days

The unit for the retryDelay value can also be expressed as a partial name, as follows:

- 30 s
- 30 sec
- 30 seconds

Note that all parameters to methods and callbacks that are marked with the @jws:message−buffer tag must be serializable (must implement java.io.Serializable). For an example, see Asynchronous Methods.

Controlling Retry Behavior

You may control retry behavior of buffered methods at runtime by using weblogic.jws.RetryException. See Controlling Retry Behavior in the topic Asynchronous Methods.

Related Topics

Using Asynchrony to Enable Long−Running Operations

Asynchronous Methods

A JWS file holds a WebLogic Workshop Java Web Service. JWS files contain syntactically correct Java code. However, they may also contain special annotation that allows the web services to take advantage of powerful features of the WebLogic Workshop runtime and the WebLogic Server. All JWS files contain a class derived from a special base class called Service that makes certain helper resources available to the class at runtime. The goal of all the helper code is to make it really easy to put a Java− and−J2EE−based service on the net with standard protocols.

@jws:operation Tag

Specifies that the associated method is part of the web service's public contract and is available for clients to invoke.

Syntax

```
@jws:operation
```

Attributes

none

Remarks

The following rules apply to this tag's use:

- Only one @jws:operation tag may appear within a single Javadoc comment block.
- Must appear on each method in a JWS file that the web service would like to publish.

The associated method must be public.

Related Topics

Structure of a JWS File

@jws:parameter−xml Tag

Specifies characteristics for marshaling data between XML messages and the data in a Java declaration's parameters.

Syntax

```
@jws:parameter−xml
[include−java−types="javaTypes"]
[schema−element="schemaPrefix:schemaType"]
[xml−map :: mapText ::]
```

Attributes

include−java−types

Optional. One or more Java types that should be used for serializing message data.

schema−element

Optional. The name of an XML schema type, such as "xsd:string".

xml−map

Optional. Indicates that parameters in the annotated declaration should be mapped to XML message values as described in an XML map or script.

Remarks

Use the @jws:parameter−xml tag to define how the data received or sent via parameters should be treated. Each of the tag's attributes represents a characteristic of the translation between Java data types and XML values. (The @jws:return−xml applies the same functionality to a method or callback's return values.)

include−java−types

The include−java−types attribute lists Java data types that should be used when the Java type sent or received in an XML message may be a subclass of a type in the declaration.

```
public void updateAddress(Address newAddressData)
```

To support addresses in the United States and Great Britain, the Address object passed to this method would need to support data for the postal code formats of either nation. To enable this support, and to keep things simple, you might implement two subclasses of the Address object adov−−> one including a field for usPostalCode, the other with a field for gbPostalCode. Either of these subclasses, USAddress and GBAddress, are legal as parameters to the updateAddress method.

However, as the updateAddress method is declared, WebLogic Server has no way of knowing which subclasses of Address the method should accept as parameter types. To respect the exact interface you have defined with this declaration, WLS will serialize all incoming compatible objects as Address adov−−> the type you have specified in the declaration.

To specify that USAddress and GBAddress (along with data members unique to those types) should be appropriately passed to your method, specify them with the include−java−types attribute, as follows:

```
/**
 * @jws:operation
 * @jws:parameter-xml include-java-types="com.myCompany.USAddress
 *  com.myCompany.GBAddress"
 */
public void updateAddress(Address newAddressData)
{...}
```

The include−java−types attribute is also useful when a Java Collection class is being converted. Java Collection classes contain Objects, which cannot be successfully converted to or from XML without additional information. The include−java−types attribute may contain a space−separated list of Java types that are stored in the Collection. For example, if a method accepts ArrayList that may contain Integers and Strings, specify include−java−types as "Integer String".

schema−element

Use the schema−element attribute to indicate to that your service supports specific complex schema types. You should only use this attribute when your implementation is aware of the type and handles it appropriately.

Use the xml−map attribute specify that the parameters of the declaration should be mapped to XML values using the XML map provided. The xml−map attribute may include either the text of an XML map or code to invoke one, or code to invoke a script. For more information on XML maps, see Why Use XML Maps?

Related Topics

@jws:return−xml Tag

How Do XML Maps Work?

ConsolidateAddress.jws Sample

ExplicitTypes.jws Sample

InputMapMultiple.jws Sample

OutputMap.jws Sample

OutputScriptMap.jws Sample

SimpleMap.jws Sample

Specifies which protocols and message formats a web service can accept or a Service control will send to the service it represents.

Syntax

```
@jws:protocol
form-get="true | false"
form-post="true | false"
http-soap="true | false"
http-xml="true | false"
jms-soap="true | false"
jms-xml="true | false"
soap-style="document | rpc"
```

Attributes

form−get

Specifies that messages encoded as HTTP GET requests are understood.

Default: true.

form−post

Specifies that messages encoded as HTTP POST requests are understood.

Default: true.

http−soap

Specifies that SOAP messages conveyed over HTTP are understood.

Default: true.

http−xml

Specifies that XML messages conveyed over HTTP are understood. Note: cannot be used with conversational methods or callbacks. See below.

Default: false.

jms−soap

Specifies that SOAP messages conveyed over JMS are understood.

Default: false.

jms−xml

Specifies that XML messages conveyed over JMS are understood. Note: cannot be used with conversational methods or callbacks. See below.

Default: false.

soap−style

Specifies which of the two method call styles specified by SOAP is supported. "document" specifies the style commonly referred to as "document literal". "rpc" specifies the style commonly referred to as "SOAP RPC".

Default: "document".

Remarks

The following rules apply to this tag's use:

- A single @jws:protocol tag may appear within a single Javadoc comment block.
- Optionally may appear in front of a class in a JWS file.
- Optionally may appear in front of an interface in a CTRL file.

The @jws:protocol tag specifies which network protocols and message formats may be used to communicate with a web service.

When Applied to a Class in a JWS File

When applied to a class in a JWS file, @jws:protocol tag specifies which protocols and messages formats the web service is able to receive. The web service will listen for requests on the listed protocols and will understand messages that arrive in the listed formats.

The default behavior for a JWS file is to enable SOAP with "document literal" formatting of method calls, via HTTP POST, and HTTP GET.

When Applied to an Interface in a CTRL File

A Service control's CTRL file must specify a single protocol binding.

The Service control represents a web service, referred to as the target service. The target service's WSDL description specifies which protocols and message formats the target service accepts. The Service control's interface in its CTRL file must contain a @jws:protocol tag that selects exactly one of the protocol/message format combinations the target service is capable of receiving.

Raw XML Protocols (http−xml and jms−xml) Do Not Support Conversations

WebLogic Workshop uses SOAP headers to convey conversation ID and other conversational information. Raw XML messages don't carry SOAP headers and therefore cannot participate in conversations.

Callback Protocol is Inferred

The protocol and message format used for callbacks is always the same as the protocol and message format used by the start method that started the current conversation. It is an error to attempt to override the protocol or message format of a callback.

Examples

The @jws:location tag is used to control protocol bindings for a web service. Consider the following example:

```
/**
 * @jws:protocol http−get="true" http−post="false" http−soap="false"
 */
```

A web service with this @jws:protocol tag:

- Will not accept HTTP SOAP messages.
- Will accept form−based GET messages.
- Will not accept form−based POST messages.

In the example CTRL file below, the @jws:location tag specifies a URL for the Bank.jws web service that this CTRL file represents, and the @jws:protocol tag specifies that this Service control will communicate with the BankControl using SOAP over HTTP.

```
import weblogic.jws.control.ServiceControl;




/**
 * @jws:location http-url="http://localhost:7001/webapp/Bank.jws"


 * @jws:protocol http-soap="true"
 */
public interface BankControl extends ServiceControl
{
}
```

Related Topics

@jws:location Tag

WSDL Files: Web Service Descriptions

Service Control: Using Another Web Service

@jws:return−xml Tag

Defines mappings between the return type of the method marked by this tag and XML message data.

Syntax

```
@jws:return-xml
[include-java-types="javaTypes"]
[schema-element="schemaPrefix:schemaType"]
[xml-map :: mapText ::]
```

Attributes

include−java−types

Optional. One or more Java types that should be used for serializing message data.

schema−element

Optional. The name of an XML schema type, such as "xsd:string".

xml−map

Optional. Indicates that the return value in the annotated declaration should be mapped to XML message values as described in an XML map or script.

Remarks

Use the @jws:return−xml tag to define how the data received or sent via the return value should be treated. Each of the tag's attributes represents a characteristic of the translation between Java data types and XML values. (The @jws:parameter−xml tag applies the same functionality to a method or callback's parameter values.)

include−java−types

The include−java−types attribute lists Java data types that should be used when the Java type sent or received in an XML message may be a subclass of a type in the declaration. Consider the following declaration:

```
public Address getCurrentAddress(String employeeID)
{...}
```

To support addresses in the United States and Great Britain, the Address object returned by this method would need to support data for the postal code formats of either nation. To enable this support, and to keep things simple, you might implement two subclasses of the Address object adov−−> one including a field for usPostalCode, the other with a field for gbPostalCode. Either of these subclasses, USAddress and GBAddress, are (from the Java perspective) legal as a type to return from the updateAddress method because they are subclasses of Address.

However, as the updateAddress method is declared, WebLogic Server has no way of knowing which subclasses of Address the method should be sent. To respect the exact interface you have defined with this declaration, WLS will serialize all outgoing compatible objects as Address adov−−> the type you have specified in the declaration.

To specify that USAddress and GBAddress (along with data members unique to those types) should be appropriately returned by your method, specify them with the include−java−types attribute, as follows:

```
/**
 * @jws:operation
 * @jws:return−xml include−java−types="com.myCompany.USAddress
 *  com.myCompany.GBAddress"
 */
public Address updateAddress(String employeeID)
{...}
```

The include−java−types attribute is also useful when a Java Collection class is being converted. Java Collection classes contain Objects, which cannot be successfully converted to or from XML without additional information. The include−java−types attribute may contain a space−separated list of Java types that are stored in the Collection. For example, if a method returns an ArrayList that may contain Integers and Strings, specify include−java−types as "Integer String".

schema−element

Use the schema−element attribute to indicate to that this method or callback supports a specific complex schema type. You should only use this attribute when your implementation is aware of the type you and handles it appropriately.

Use the xml−map attribute to specify that the return value of the declaration should be mapped to XML values using the XML map provided. The xml−map attribute may include either the text of an XML map or code to invoke one, or code to invoke a script. For more information on XML maps, see Why Use XML Maps?

Related Topics

@jws:parameter−xml Tag

How Do XML Maps Work?

ConsolidateAddress.jws Sample

ExplicitTypes.jws Sample

InputMapMultiple.jws Sample

OutputMap.jws Sample

OutputScriptMap.jws Sample

SimpleMap.jws Sample

@jws:schema Tag

Specifies the schema file whose types are referenced in the @jws:parameter−xml schema and @jws:return−xml schema tags elsewhere in a JWS or CTRL file.

Syntax

```
@jws:schema
import="schemaToImport"
```

Attributes

import

Required. The filename or @jws:define tag referring to the schema to import.

Remarks

The following rules apply to this tag's use:

- Multiple @jws:schema tags are allowed within a single Javadoc comment block.
- Optionally may appear in front of a class in a JWS file.
- Optionally may appear in front of an interface in a CTRL file.

The import attribute should refer to a file or a @jws:define tag that contains a schema file. The filename resolution rules are the same as for the @jws:implements tag.

Any number of schema files can be imported by using multiple @jws:schema tag declarations. When outputting WSDL files, imported schemas are copied and inserted inline into the WSDL.

Related Topics

@jws:parameter−xml Tag

@jws:return−xml Tag

@jws:define Tag

@jws:wsdl Tag

@jws:sql Tag

Specifies the SQL statement and associated attributes that correspond to a method in a Database control.

Syntax

```
@jws:sql
    statement="sqlStatement"


    [iterator−element−type="javaType"]


    [array−max−length=integer OR "all"]
```

Attributes

statement

Required. The SQL (Structured Query Language) statement that will be executed when the associated Database control method is invoked. The statement may contain substitutions of the form {varName}, where paramName is a parameter of the Database control method (or a member accessible from the parameter).

If the statement is a single line, it may be formatted with and equals sign and quotes, as follows:

```
/**
 * @jws:sql statement="SELECT * FROM sometable"
 */
public HashMap getEverything();
```

If the statement spans multiple lines, it is delimited with :: (two colons), as follows:

```
/**
 * @jws:sql statement::
 *     SELECT name
 *     FROM employees
 *     WHERE name LIKE {partialName}
 * ::
 */
public String[] partialNameSearch(String partialName);
```

For a discussion of parameter substitution in the statement attribute, see Parameter Substitution in @jws:sql Statements.

iterator−element−type

Required if Database control method return type is java.util.Iterator. Specifies the underlying class of the Iterator that will be returned by the Database control method. The class must meet specific criteria that are described in the Returning an Object section of Returning a Single Row from a Database Control Method.

array−max−length

Optional. If the return type of the associated Database control method is an array, specifies the maximum size of the array that will be returned. This attribute can be used to set an upper limit on memory usage by a Database control method.

Default value: 1024

Note that if the associated query returns more rows than specified by array−max−length, there is no way to access the excess rows.

The special value "all" will cause all rows that satisfy the query to be returned. Note that this can possibly exhaust all available memory if not used carefully.

If you wish to limit memory usage, you should return an Iterator or a ResultSet. To learn about returning Iterators and ResultSets from a Database control method, see Returning Multiple Rows from a Database Control Method.

Remarks

The following rules apply to this tag's use:

- The @jws:sql tag may only occur on a method declaration in an interface that extends weblogic.jws.control.DatabaseControl.
- The @jws:sql tag may only occur in a CTRL file.
- The @jws:sql tag may only appear once per method.

Related Topics

Database Control: Using a Database from Your Web Service

Parameter Substitution in @jws:sql Statements

Returning Multiple Rows from a Database Control Method

@jws:target−namespace Tag

Specifies the XML namespace used for outgoing XML messages and generated WSDL files.

Syntax

```
@jws:target−namespace
        namespace="XMLNamespace"
           default
```

Attributes

namespace

Required. Specifies the default namespace used for outgoing XML messages and generated WSDL files.

Remarks

The following rules apply to this tag's use:

- Optionally may appear on the main class in a JWS file.
- Optionally may appear on the main interface in a CTRL file.

If no target namespace is specified (this tag is not present), the namespace used is http://www.openuri.org/.

Note: Before deploying web services, you should define a unique default namespace for your organization. If multiple organizations were to leave the default namespace as http://www.openuri.org/, it could result in namespace conflicts.

The @jws:target−namespace tag affects the behavior of the following XML generation actions:

- If there is no XML map specified for an incoming or outgoing message, a default XML map is used (the natural mapping). The elements of the resulting XML message must be specified in some namespace. The value of @jws:target−namespace is used.

- If a custom XML map is present for an incoming or outgoing message but no namespace is specified in the XML map, the value of @jws:target−namespace is used.
- WSDL files generated from JWS files specify a namespace for the elements in the WSDL file. The value of @jws:target−namespace is used.

Note that if @jws:target−namespace is not defined, the default value of http://www.openuri.org/ is used.

For more information on XML maps, see Why Use XML Maps?

For more information on WSDL files, see WSDL Files.

Related Topics

Introduction to XML

@jws:timer Tag

Specifies configuration attributes for a Timer control.

Syntax

```
@jws:timer

    [timeout="timeSpecification"]

    [repeats−every="timeSpecification"]
    [coalesce−events="true" | "false"]
```

Attributes

These attributes determine the default behavior of the Timer control. The Timer control may be configured during its lifetime by calling methods of the TimerContol class. To learn more about the TimerControl class, see TimerControl Class.

timeout

Optional. Specifies the time between a call to the Timer control's start or restart method and the Timer control's first onTimeout callback. When the timer expires, the Timer control's onTimeout callback is called.

The timeout attribute expects a relative time specification. For example, "1 second". Specification of an absolute time as the value of the timeout attribute is not supported. Absolute time may only be specified by calling the TimerControl.setTimeoutAt method.

To learn how to specify values for the timeout attribute, see Specifying Time on a Timer Control.

Default: "0 sec". The timer will fire immediately after being started.

repeats−every

Optional. Specifies the interval between subsequent firings of the Timer control after the first expiration. If a valid interval greater than "0 seconds" is specified, the Timer control will continue firing at that interval until stopped.

The repeats−every attribute expects a relative time specification. For example, "1 second". Specification of an absolute time as the value of the repeats−every attribute is not supported.

To learn how to specify values for the repeats−every attribute, see Specifying Time on a Timer Control.

Default: "0 sec". The timer will fire once (at the time specified by the timeout attribute) but will never fire again.

coalesce−events

Optional. Specifies whether multiple undelivered firing events of a Timer control are delivered as a single onTimeout or as separate callbacks. At times, a Timer control may be unable to deliver one or more callbacks to its referring service. This may occur because the referring service is busy or because high system load delays delivery. A set of undelivered callbacks may accumulate. If the coalesce−events attribute is true, these accumulated callbacks are collapsed into a single callback when the service becomes available. If coalesce−events is false, the accumulated callbacks are delivered individually.

Default: "false".

Remarks

The following rules apply to this tag's use:

- The @jws:timer tag may only occur on an declaration of an object of type weblogic.jws.control.TimerControl or of a type that extends TimerControl.
- The @jws:timer tag is typically found on declaration that also carry the @jws:control tag.

A Timer control callback will never interrupt the current thread of execution of a web service operation (a method marked with the @jws:operation tag). If a Timer control expires during execution of a web service operation, the onTimeout callback will be invoked immediately after the current web service operation completes.

Related Topics

Timer Control: Using Timers in Your Web Service

Controls: Using Resources from a Web Service

@jws:wsdl Tag

Specifies a WSDL file that is implemented by a web service or represented by a Service control.

Syntax

```
@jws:wsdl
      file="fileName"
```

Attributes

file

Required. Specifies the name of a WSDL file. fileName may begin with the # character, in which case the referenced WSDL file is expected to be found in line in the current file as the value of a @jws:define tag with name attribute value fileName. This arrangement is illustrated in the following example:

```
import weblogic.jws.control.ServiceControl;
/**
 * @jws:location http-url="creditreport/IRS.jws" jms-url="creditreport/IRS.jws"
 * @jws:wsdl file="#IRSWsdl"
 */
public interface IRSControl extends ServiceControl
{
    ...
}
/**
 @jws:define name="IRSWsdl" value::
      <?xml version=1.0 encoding=utf-8?>
      <definitions ...>
        ...remainder of WSDL here...
      </definitions>
  ::
 */
```

Remarks

The following rules apply to this tag's use:

When optionally applied to a class in a JWS file:

- Only one @jws:wsdl tag may be present within the class' Javadoc comment block.

When optionally applied to an interface in a CTRL file defining a Service control:

- Only one @jws:location tag may be present in the interface's Javadoc comment block.

A WSDL (Web Services Description Language) file conveys the public contract of a web service. When the @jws:wsdl tag is applied to a class in a JWS file or an interface in a CTRL file, it indicates that the service in the JWS file or the service represented by the CTRL file implements the public contract expressed in the WSDL file.
WebLogic Workshop validates the actual interface described by the service or Service control. Errors are generated at compile time if the defined interface does not comply with that described in the WSDL.
A Service control generated from a WSDL file is always annotated as implementing the WSDL file from which it was generated.

Related Topics

Service Control: Using Another Web Service

WSDL Files: Web Service Descriptions

@jws:define Tag

@jws:xmlns Tag

Defines an XML namespace prefix for use elsewhere in a JWS file or Service control CTRL file.

Syntax

```
@jws:xmlns
        prefix="namespacePrefix"]
        namespace="namespaceURI"
```

Attributes

prefix

Required. Specifies the XML namespace prefix being defined.

namespace

Required. Specifies the URI that the prefix represents.

Remarks

The following rules apply to this tag's use:

- Multiple @jws:xmlns tags may appear within a single Javadoc comment block.
- May appear on the main class declaration in a JWS file or the main interface declaration in a CTRL file..

The following XML namespace prefixes are implicitly defined in WebLogic Workshop:

- xm, with namespace URI "http://www.bea.com/2002/04/xmlmap/"
- xsd, with namespace URI "http://www.w3.org/2001/XMLSchema"

The xm prefix is used in XML map tags such as <xm:value> and <xm:multiple>.

If the xm or xsd prefixes are explicitly defined in a JWS or CTRL file, the implicit definitions described above are overridden for that file.

To learn about XML namespaces and prefixes, see Introduction to XML.

Related Topics

Introduction to XML

XML Map Tag Reference

Through XML maps, you can customize the way values in XML messages received and sent by your service's methods are mapped to the variables of your method's implementation. You use the following tags and attributes inside XML maps.

Topics Included in This Section

<xm:attribute> Tag

Specifies the variable to use when mapping an XML attribute value.

<xm:bind> Attribute

Declares a variable within a map and initializes to a variable of the declaration.

<xm:java−import> Tag

Specifies Java classes that should be imported to support variables in an XML map file.

<xm:map−file> Tag

Specifies that its contents constitute an XML map file.

<xm:multiple> Attribute

Specifies the data structure variable to which a specific repeating XML element should be mapped.

<xm:use> Tag

Specifies a function or XML map to invoke for processing XML.

<xm:value> Tag

Specifies the variable to use when mapping a specific XML element content.

<xm:xml−map> Tag

Specifies that its contents constitute an XML map; also provides the signature defining calls to this map.

Related Topics

Why Use XML Maps?

<xm:attribute> Tag

Specifies the variable to use when mapping an XML attribute value.

Syntax

```
<xm:attribute
    name="attributeName"
    obj="[dataType ]attributeValueVariable[, …]"
>
```

Attributes

name

The name of the attribute that is being mapped.

obj

The portion of the Java declaration to which this attribute's value should be mapped, such as a parameter or "return".

Remarks

Use the <xm:attribute> tag to assign an attribute value from an XML message to a portion of your Java declaration. In the following example, max_records is an anticipated attribute of the <query> element:

```
/**
 * @jws:operation


 * @jws:parameter-xml xml-map::
 * <query>
 *     <xm:attribute name="max_records" obj="resultSize"/>
 *     <element>
 *         <name>{criterionName}</name>
 *         <value>{criterionValue}</value>
 *     </element>
 * </query>
 * ::
 */
public void searchData(String criterionName, String criterionValue, int resultSize)
{
    System.out.println("The maximum number of records to return is " + resultSize);
}
```

As an alternative to using the <xm:attribute> tag, you can use curly braces shorthand. The following snippet is equivalent to the preceding example:

```
/**
 * @jws:parameter-xml xml-map::
 * <query max_records="{resultSize}">
 *     <element>
 *         <name>{criterionName}</name>
 *         <value>{criterionValue}</value>
 *     </element>
 * </query>
 * ::
 */
public void searchData(String criterionName, String criterionValue, int resultSize)
{
    System.out.println("The maximum number of records to return is " + resultSize);
}
```

Specify a data type when the variable is not declared with a type elsewhere in the map's context (such as elsewhere in the map, in Java code at the class level, or in the code for the method to which the map applies).

```
/**
 * @jws:parameter-xml xml-map::
 * <query max_records="{int resultSize}">
 *     <element>
 *         <name>{java.sql.String criterionName}</name>
 *         <value>{java.sql.String criterionValue}</value>
 *     </element>
 * </query>
 * ::
 */
public void searchData(String criterionName, String criterionValue, int resultSize)
{
    System.out.println("The maximum number of records to return is " + resultSize);
}
```

Note   The xm prefix and its URI are declared implicitly in any JWS file. However, you must declare the namespace prefix and URI to use the prefix in XMLMAP files.

Related Topics

How Do XML Maps Work?

Why Use XML Maps?

<xm:bind> Attribute

Declares a variable within a map and initializes to a variable of the declaration.

Syntax

```
<elementName
    xm:bind="elementName variableNameVariable is dataStructure.arrayMember[,...]"
>
```

Attributes

bind

A statement declaring a variable and initializing it to a parameter variable.

Remarks

The <xm:bind> attribute is especially useful when you need to bind the XML value to a member of a data structure, while also assigning it a short variable name. The <xm:bind> attribute also specifies that the object corresponding to the variable you're binding should not be instantiated unless the tag is encountered in the instance document.

In the following example, <xm:bind> makes it possible to declare a new variable a of type Address (which must be a type available in the scope of the code, such as an internal class), then initialize the new variable to the address member of the customerData structure. Using <xm:bind> as an attribute of the <address> tag means that the <address> element's value will be mapped to the new variable. In addition, the new a variable can be used in the <street> and <zip> elements.

```
/**
 * @jws:parameter-xml xml-map::
 *   <customer>
 *     <name>{String customerData.name}</name>
 *     <address xm:bind="Address a is customerData.address">
 *       <street>{a.street}</street>
 *       <zip>{a.zip}</zip>
 *     </address>
 *   </customer>
 *   ::
 */
public void addCustomerData(MyStructure customerData)
{
   System.out.println("Customer name is " + customerData.get("name"));
   System.out.println("Customer zipcode is " +
         ((Address)customerData.get("address")).zip);
}
```

Note that the is operator is a reserved word in the context of the <xm:bind> attribute. This means that using variables or types called "is" in the value of the <xm:bind> attribute will generate an error.

Note   The xm prefix and its URI are declared implicitly in any JWS file. However, you must declare the namespace prefix and URI to use the prefix in XMLMAP files.

Related Topics

Declaring Variables with the <xm:bind> Attribute

Why Use XML Maps?

<xm:java−import> Tag

Specifies Java classes that should be imported to support variables in an XML map file.

Syntax

```
<xm:java−import
    class="fullyQualifiedClassName"
/>
```

Attributes

class

The fully−qualified name of the class.

Remarks

Use the <xm:java−import> tag in an XMLMAP file just as you would use the import directive in a JAVA or JWS file. As with the Java import directive in Java code, you use <xm:java−import> to specify classes and packages needed for type context in XML maps. For example, to use Date as a short type name in XML maps, first import the Date type as follows:

```
<xm:java-import class="java.sql.Date"/>
```

You may use multiple <xm:java−import> tags, but all must occur immediately following the <xm:map−file> tag and before any XML maps.

Note   The xm prefix and its URI are declared implicitly in any JWS file. However, you must declare the namespace prefix and URI to use the prefix in XMLMAP files.

Related Topics

Creating Reusable XML Maps

<xm:map−file> Tag

Specifies that its contents constitute an XML map file.

Syntax

```
<xm:map-file
    xmlns:xm="http:bea.com/jws/xmlmap"
>
```

Attributes

xmlns:xm

URI defining the namespace for tags with an xm prefix.

Remarks

The <xm:map−file> tag is used only in map files — files with an .xmlmap extension that contain XML maps. A map file must begin and end with <xm:map−file> tags.

Note   The xm prefix and its URI are declared implicitly in any JWS file. However, you must declare the namespace prefix and URI to use the prefix in XMLMAP files. You may change the prefix by declaring another prefix.

Related Topics

Creating Reusable XML Maps

<xm:multiple> Attribute

Specifies the data structure variable to which a specific repeating XML element should be mapped.

Syntax

```
<elementName
    xm:multiple="[dataType ]arrayMemberVariable in [dataType ]arrayVariable[, …]"
>
```

Attributes

xm:multiple

A statement expressing the mapping between an element's value and a method variable, and the variable's role as a member of an array.

Remarks

Use the <xm:multiple> tag to capture the values of repeating XML elements. In the following XML example, the <part> element (and its children) repeats three times. Because this repetition resembles a list, you can capture the repeating values by mapping them to a data structure.

```
<order>
    <part>
        <partID>19573</partID>
        <partQuantity>1</partQuantity>
    </part>
    <part>
        <partID>28912</partID>
        <partQuantity>1</partQuantity>
    </part>
    <part>
        <partID>39485</partID>
        <partQuantity>57</partQuantity>
    </part>
</order>
```

The <xm:multiple> attribute assumes that your Java code has declared collections to store the values of each repeating element. The syntax of the <xm:multiple> attribute's value essentially assigns each value of the repeating element to a place in the collection, which your code can then inspect iteratively.

Note   The in operator is a reserved word in the context of the <xm:multiple> attribute. This means that using variables or types called "in" in the value of the <xm:multiple> attribute will generate an error.

The following example is designed to operate on an incoming XML message like the preceding example. In this example, the <xm:multiple> attribute specifies that the contents of the <patID> and <numberOfItems> elements should be added as members of the serialNumber and quantity arrays.

```
/**
 * @jws:operation
 * @jws:parameter-xml xml-map::
 * <placeOrder>
 * <order>
 *      <part xm:multiple="String serial in serialNumber, int quant in quantity">
 *         <partID>{serial}</partID>
 *         <numberOfItems>{quant}</numberOfItems>
 *      </part>
 * </order>
 * </placeOrder>
 *
 * ::
 */
public void placeOrder(String[] serialNumber, int[] quantity)
{
      for (int i = 0; i < serialNumber.length; i++)
      {
        System.out.println("Ordered " + quantity[i] + " of part " + serialNumber[i]);
      }
}
```

Note   The xm prefix and its URI are declared implicitly in any JWS file. However, you must declare the namespace prefix and URI to use the prefix in XMLMAP files. You may change the prefix by declaring another prefix.

Related Topics

Handling Repeating XML Values with <xm:multiple>

<xm:use> Tag

Specifies a function or XML map to invoke for processing XML.

Syntax

```
<xm:use
    call="fileName([dataType ]elementValueVariable)"
/>
```

Attributes

call

A call to an XML map or script function that is external to the service source code.

Remarks

Use the <xm:use> tag when you have an XML map in a map file or ECMAScript functions in a JSX file and you want to invoke the map or script from within your service source code. The value of the call attribute specifies the map or function name, along with parameters to pass to the map or function. The names of parameters in parenthesis must match the names of items in the Java declaration that you are mapping.

fileName can refer to one of the following three types of files:

- An XML map contained in a file with an .xmlmap extension.
- An ECMAScript file with a .jsx extension.

Using XML programming extensions available with WebLogic Server, you can manipulate XML with ECMAScript. Note that the .jsx file must contain two functions: mymapToXML and mymapFromXML.

Note   The xm prefix and its URI are declared implicitly in any JWS file. However, you must declare the namespace prefix and URI to use the prefix in XMLMAP files. You may change the prefix by declaring another prefix.

Related Topics

Creating Reusable XML Maps

Handling XML with ECMAScript Extensions

<xm:value> Tag

Specifies the variable to use when mapping a specific XML element content.

Syntax

```
<xm:value
    obj="[dataType ]elementValueVariable[, ...]"
/>
```

Attributes

obj

The portion of the Java declaration to which this element content should be mapped.

Remarks

Use the <xm:value> tag to assign element content from an XML message to a portion of your Java declaration. The following example maps criterionName and criterionValue to the <name> and <value> elements, respectively.

```
/**
 * @jws:operation
 * @jws:parameter-xml xml-map::
 * <query>
 *     <xm:attribute name="max_records" obj="resultSize"/>
 *     <element>
 *         <name><xm:value obj="criterionName"/></name>
 *         <value><xm:value obj="criterionValue"/></value>
 *     </element>
 * </query>
 * ::
 */
public void searchData(String criterionName, String criterionValue, int resultSize)
{
    System.out.println("The maximum number of records to return is " + resultSize);
}
```

As an alternative to using the <xm:value> tag, you can use curly braces shorthand. The following snippet is equivalent to the preceding example:

```
/**
 * @jws:operation
 * @jws:parameter-xml xml-map::
 * <query max_records="{resultSize}">
 *     <element>
 *         <name>{criterionName}</name>
 *         <value>{criterionValue}</value>
 *     </element>
 * </query>
 * ::
 */
public void searchData(String criterionName, String criterionValue, int resultSize)
{
    System.out.println("The maximum number of records to return is " + resultSize);
}
```

Specify a data type when the variable is not declared with a type elsewhere in the map's context (such as elsewhere in the map, in Java code at the class level, or in the code for the method to which the map applies).

```
/**
 * @jws:parameter-xml xml-map::
 * <query max_records="{resultSize}">
 *     <element>
 *         <name>{java.sql.String criterionName}</name>
 *         <value>{java.sql.String criterionValue}</value>
 *     </element>
 * </query>
 * ::
 */
public void searchData(String criterionName, String criterionValue, int resultSize)
{
    System.out.println("The maximum number of records to return is " + resultSize);
}
```

Note   The xm prefix and its URI are declared implicitly in any JWS file. However, you must declare the namespace prefix and URI to use the prefix in XMLMAP files. You may change the prefix by declaring another prefix.

Related Topics

How Do XML Maps Work?

Why Use XML Maps?

<xm:xml-map> Tag

Specifies that its contents constitute an XML map; also provides the signature defining calls to this map.

Syntax

```
<xm:xml-map
    signature="mapSignature"
>
```

Attributes

signature

The signature of map enclosed in <xm:xml−map> tags.

Remarks

The <xm:xml−map> tag is used only in map files — files with an .xmlmap extension that contain XML maps. You must enclose every map in a map file between <xm:xml−map> tags. The signature attribute specifies name of the XML map and parameters expected for a call to the enclosed map. For example, consider the following map:

```
<!-- Defined in a file called "BookMaps.xmlmap" -->
<xm:xml-map signature="getInventory(String partID)">
    <checkInventory>
        <partID>{partID}</partID>
    </checkInventory>
</xm:xml-map>
```

A call to the map in this example might look something like the following.

```
/**
 * @jws:operation
 * @jws:parameter-xml xml-map::
 * <checkInventory>
 *     {BookMaps.getInventoryString(String ISBN)}
 * </checkInventory>
 * ::
 *
 */
public String checkInventory(String ISBN)
{
    return "You checked for copies of " + ISBN + ".";
}
```

Note   The xm prefix and its URI are declared implicitly in any JWS file. However, you must declare the namespace prefix and URI to use the prefix in XMLMAP files. You typically do this in the <xm:map−file> tag.

Related Topics

Creating Reusable XML Maps

<xm:map−file> Tag

Command Reference

The following are commands associated with WebLogic Workshop.

Topics Included in This Section

JwsCompile Command

Pre−compiles one or more web services, producing an EAR file that may be deployed to a production server.

startWebLogic Command

Starts WebLogic Server from the command line, allowing specification of options.

JwsCompile Command

The JwsCompile tool compiles one or more web services in a WebLogic Workshop project. Optionally, JwsCompile may produce either: 1) an EAR file (containing multiple web services) that may be deployed to a production server; or 2) a CTRL file or WSDL file from a single web service; or 3) a CTRL file from a WSDL file.

Syntax

```
JwsCompile −a [ −f −nowarn −p −v −x ]
```

```
JwsCompile [ −f −nowarn −p −v ] <JWS File Name>
```

```
JwsCompile −a −ear [ −app −f −hostname −http−port −https−port −nowarn −p −protocol −t −v −x <Jw
```

```
JwsCompile −ctrl [ −f −hostname −http−port −https−port −nowarn −out −p −protocol −v ] <JWS File
```

```
JwsCompile −wsdl [ −f −hostname −http−port −https−port −nowarn −out −p −protocol −v ] <JWS File
```

Parameters

−a

Specifies that all JWS files in the project should be compiled.

−app <Application Name>

Specifies the name of the application. See A Web Service's URL below. Do not include leading or trailing slashes ('/') in <Application Name>.

−ctrl

Generates a CTRL file from the specified JWS or WSDL file. The CTRL file defines a WebLogic Workshop Service control.

−default

When used with the −ear parameter, specifies that the EAR file should be deployed as the default application on the production server. For example, an application compiled by the following command
jwsCompile −ear HelloWorld.ear −app HelloWorld HelloWorld.jws
will be accessed on the production server through the following URL
    http://myProductionServer:7001/HelloWorld/HelloWorld.jws
But an application compiled by the following command
jwsCompile −ear HelloWorld.ear −default HelloWorld.jws
will be accessed on the production server through the following URL
http://myProductionServer:7001/HelloWorld.jws

−ear <Output EAR file>

Specifies the name of the EAR file to be produced. May include an absolute path or a path relative to the current directory on the command line. No EAR file is produced if the −ear argument is not present.

Since −ear is typically used to generate an EAR file containing web services to be deployed to a different destination server, the URL of the web services on the destination server should be specified either in weblogic−jws−config.xml or by use of a cobination of the −protocol, −hostname, −http−port, −https−port and −app arguments.

−f

Forces compilation of a JWS file(s) even it is believed to be up to date. Use in conjunction with the −t parameter

−hostname <Hostname>

Specifies the hostname to use in the URL of each web service in the project. Overrides any values specified in weblogic−jws−config.xml. Default value is the hostname of the host on which JwsCompile is run.

−http−port <Port Number>

Specifies the port to use in the URL of each HTTP−bound web service in the project. Overrides any values specified in weblogic−jws−config.xml. Default value is 7001.

−https−port <Port Number>

Specifies the port to use in the URL of each HTTPS−bound web service in the project. Overrides any values specified in weblogic−jws−config.xml. Default value is 7002.

−nowarn

Disables warnings.

−out <Output File>

Specifies the name of the output file for −ctrl and −wsdl. May include an absolute path or a path relative to the directory in which jwsCompile is run.

−p <Project Directory>

Specifies where project sources are. May be an absolute path or a path relative to the directory in which JwsCompile is run.

−protocol <http | https>

Specifies the protocol to use in the URL of each web service in the project. Overrides the global <protocol> value specified in weblogic−jws−config.xml. However, −protocol will be overridden by individual <jws><protocol> specifications in weblogic−jws−config.xml. Default value is http.

−t <Target Directory>

Specifies the destination directory for partial results of JwsCompile. Generated EAR, CTRL and WSDL files are not placed in <Target Directory>; the location of those files is specified by the −out argument. If the same <Target Directory> is specified for subsequent invocations of JwsCompile, any up−to−date compilation products in the <Target Directory> are reused and the associated source files are be recompiled.

If −t is not specified, a temporary directory is created for partial compilation results. The directory is deleted upon completion of JwsCompile.

−v

Reports progress verbosely.

−wsdl

Generates a WSDL file from the specified JWS file.

−x <Exclude JWS>

When −a is specified, indicates that <Exclude JWS> should be excluded from compilation. Multiple −x arguments may be specified.

<JWS File Name>

The name of the JWS file to compile. Only one JWS file may be specified, except when used with the −ear flag. In that case, multiple JWS files can be specified.

Remarks

JwsCompile may be used for several purposes. Typical uses are listed below:

JwsCompile −a [−f −nowarn −p −v −x]

Compiles all web services in the current project or the project indicated by −p, with the exception of any web services named in −x arguments. Does not produce an EAR file. This form may be used to perform compile−time error checking on all web services in the project.

JwsCompile [−f −nowarn −p −v] <JWS File Name>

Compiles the named web services in the current project or the project indicated by −p. Does not produce an EAR file. Multiple <JWS File Name> specifications may be included.

JwsCompile −ear [−a −app −f −hostname −http−port −https−port −nowarn −p −protocol −t −v −x <JWS File Name>]

If <Jws File Name> is not specified, then all web services in the current project or the project indicated by −p are compiled. An EAR file is produced that may be deployed to a WebLogic Workshop−enabled domain. (Note if <Jws File Name> is not specified then the −a flag must be used.) The −protocol, −hostname, −http−port, −https−port, and −app flags, if specified, combine to form the URL of the application on the destination server.

If <Jws File Name> is supplied then only those JWS files which are specified are compiled. Again an EAR file is produced that can be deployed to WebLogic Workshop−enabled domain.

JwsCompile −ctrl [−f −hostname −http−port −https−port −nowarn −out −p −protocol −v] <JWS File Name>

Compiles the named web service and produces a CTRL file containing a WebLogic Workshop Service control for the named web service. Only one JWS file may be specified. The name of the CTRL file produced may be specified using the −out flag. If −out is not specified, the resulting file is output directly to the consol. The

−protocol, −hostname, −http−port, −https−port, and −app flags, if specified, combine to form the URL of the application on the destination server.

JwsCompile −wsdl [−f −hostname −http−port −https−port −nowarn −out −p −protocol −v] <JWS File Name>

Compiles the named web service and produces a WSDL file describing the named web service. Only one JWS file may be specified. The name of the WSDL file produced may be specified using the −out flag. If −out is not specified, the resulting file is output directly to the console. The −protocol, −hostname, −http−port, −https−port, and −app flags, if specified, combine to form the URL of the application on the destination server.

JwsCompile may be invoked from anywhere within a WebLogic Workshop project directory; it determines the root of the project directory by traversing up the directory hierarchy until it locates a WEB−INF folder.

Note: JwsCompile does not "cook" the deployed JWSs in the project on the development server. The only product of JwsCompile is a CTRL, WSDL or EAR file at the location specified by −out or −ear, and possibly partial compilation products in the directory specified by −t.

A Web Service's URL

The URL of a web service is of the form:

```
protocol://host:port/application/service.jws
```

The following JwsCompile arguments control the specified portions of the deployed web services' URLs:

−protocol specifies "protocol"

−hostname specifies "host"

−http−port or −https−port specifies "port"

−app specifies "application"

Related Topics

weblogic−jws−config.xml Configuration File

startWebLogic Command

Starts WebLogic Server in a specific domain.

Syntax

```
startWebLogic [nodebug] [nolog] [nopointbase] [production]
```

Parameters

nodebug

Optional. Specifies that WebLogic Server should not be run in debug mode. Running without debug mode increases performance slightly but disables debugging of web services from the WebLogic Workshop visual development environment. Default: debug is enabled.

nolog

Optional. Disables logging, which my increase performance depending on logging configuration. Default: logging is enabled.

nopointbase

Optional. Specifies that the Pointbase database should not be started with WebLogic Server. By default, startWebLogic starts the Pointbase database. You may wish to use nopointbase if the Pointbase database is already running, or if you have reconfigured WebLogic Workshop to use a different database.

production

Optional. Specifies that both WebLogic Workshop and WebLogic Server should be run in production mode. Default: development mode.

WebLogic Workshop Modes

WebLogic Workshop can be run in development mode or production mode. This mode is controlled by the weblogic.jws.ProductionMode Java property. When WebLogic Workshop is in production mode, only web services that are packaged into a deployed EAR file may be accessed by clients. Web services that are in exploded form in the project tree are not accessible. In development mode, the opposite is true: web services in exploded form are accessible and web services in deployed EAR files are not. In addition, in production mode the web page for each web service is limited to just the Overview pane; the Console, Test Form and Test XML panes are not available.

WebLogic Server Modes

WebLogic Server may also be run in either development mode or production mode. This mode is controlled by the weblogic.ProductionModeEnabled Java property. In development mode, WebLogic Server's application poller is active, meaning WebLogic Server will automatically discover new applications (new WebLogic Workshop projects) that are created and populated. In production mode, the application poller is disabled.

Note: You can read more about starting and stopping WebLogic Server at e−docs.bea.com.

Remarks

startWebLogic starts WebLogic Server in the domain from which it is run. A typical WebLogic Server installation may contain many configured domains. Each domain's configuration is controlled by the config.xml file. A good rule of thumb is that startWebLogic should be run from the directory containing the config.xml file for the domain whose server you wish to run.

Related Topics

jws−config.properties Configuration File

Configuration File Reference

The following configuration files are associated with WebLogic Workshop.

Topics Included in This Section

jws−config.properties Configuration File

The jws−config.properties file specifies WebLogic Workshop runtime configuration for a WebLogic Server domain.

workshopLogCfg.xml Configuration File

The workshopLogCfg.xml file specifies the WebLogic Workshop logging configuration and logging levels for WebLogic Server. You may use the logging facility in your web services.

weblogic−jws−config.xml Configuration File

The weblogic−jws−config.xml file allows you to configure runtime parameters of web services, both on the development server and production servers. weblogic−jws−config.xml is specific to a WebLogic Workshop project.

Workshop.properties Configuration File

The Workshop.properties file specifies configuration parameters for the WebLogic Workshop visual development environment.

.Workshop Configuration File

The .Workshop file specifies configuration parameters for the WebLogic Workshop visual development environment.

jws−config.properties Configuration File

The jws−config.properties file specifies domain−wide configuration parameters for the WebLogic Workshop runtime. The file is located in the domain root directory.

Properties

The jws−config.properties file defines the following properties:

weblogic.jws.InternalJMSServer=<JMS server name>

Specifies the name of the JMS server to be used by WebLogic Workshop JMS controls, JMS transport and JMS queues used as message buffers. The named JMS server must be configured in the current domain.

Default value: cgJMSServer

weblogic.jws.InternalJMSConnFactory=<JMS connection factory JNDI name>

Specifies the JNDI name of the JMS connection factory to be used by WebLogic Workshop JMS controls, JMS transport and JMS queues used as message buffers. The named JMS connection factory must be configured in the current domain.

Default value: weblogic.jws.jms.QueueConnectionFactory

weblogic.jws.ConversationDataSource=<data source JNDI name>

Specifies the JNDI name of the data source to be used for conversational persistence. The named data source must be configured in the current domain.

Default value: cgDataSource

weblogic.jws.JMSControlDataSource=<data source JNDI name>

Specifies the JNDI name of the data source to be used for JMS control state management. The named data source must be configured in the current domain.

Default value: cgDataSource

weblogic.jws.ConversationMaxKeyLength=<integer>

Specifies the maximum length of a conversation ID. Conversational web services have an associated conversation identifier. The identifier is proposed by the client and must be guaranteed to be unique across all web services and conversations in the current domain.

Default value: 768

Note: The conversation ID is used as the primary key in a conversation state table that resides in the database associated with the data source identified by weblogic.jws.ConversationDataSource (above). Setting weblogic.jws.ConversationMaxKeyLength to a value larger than what can be accommodated by the associated database table will result in errors.

Related Topics

How Do I: WebLogic Workshop−Enable an Existing WebLogic Server Domain?

How Do I: Create a New WebLogic Workshop−enabled WebLogic Server Domain?

workshopLogCfg.xml Configuration File

The workshopLogCfg.xml file specifies the WebLogic Workshop logging configuration and logging levels for WebLogic Server. You may use the logging facility in your web services. You may also be directed by BEA Technical Support personnel to modify WebLogic Workshop's logging behavior in order to diagnose problems you experience. This topic describes the basic features of WebLogic Workshop's logging apparatus.

Log4j Library

WebLogic Workshop uses the log4j Java logging facility developed by the Jakarta Project of the Apache Foundation. You can learn more about log4j at The Log4j Project. A brief introduction to log4j is included below.

Loggers

Log4j defines the Logger class. An application may create multiple loggers, each with a unique name. In a typical usage of log4j, an application creates a Logger instance for each application class that will emit log messages. The name of the logger is typically the same as the partially qualified name of the application class. For example, the application class com.mycompany.MyClass might create a Logger with the name "mycompany.MyClass". Loggers exist in a namespace hierarchy and inherit behavior from their ancestors in the hierarchy.

The Logger class defines four methods for emitting log messages: debug, info, warn and error. The application class invokes the appropriate method (on its local Logger) for the situation being reported.

Appenders

Log4j defines Appenders to represent destinations for logging output. Multiple Appenders may be defined. For example, an application may define an Appender that sends log messages to the console, and another Appender that writes log messages to a file. Individual Loggers may be configured to write to zero or more Appenders. One example usage would be to send all logging messages (all levels) to a log file, but only error level messages to the console.

Layouts

Log4J defines Layouts to control the format of log messages. Each Layout specifies a particular message format in which may be substituted the data such as the current time and date, the log level, the Logger name, the log message and other information. A specific Layout is associated with each Appender. This allows you to specify a different log message format for console output than for file output, for example.

Configuration

All aspects of log4j configuration at runtime. This is typically accomplished with an XML configuration file whose format is defines by the log4j library. The configuration file may be specified at runtime using the log4j.configuration Java property.

The configuration file may declare Loggers, Appenders and Layouts and configure combinations of them to provide the logging style desired by the application.

WebLogic Workshop Logging

Configuration File

By default, WebLogic Workshop's logging configuration is defined in the file workshopLogCfg.xml, which is in BEA_HOME/weblogic700/common/lib. You may override this default by specifying the location of an alternative log4j configuration file with the log4j.configuration Java property. For example, on the command line used to start WebLogic Server, you could specify −Dlog4j.configuration=<path to config file>.

By default, WebLogic Workshop uses two log files: workshop.log and jws.log.

workshop.log Log File

The file workshop.log is configured to receive all internal logging messages emitted by the WebLogic Workshop runtime.

jws.log Log File

The file jws.log is configured to receive all logging messages emitted by user code associated with web services. This includes code in JWS files, but also code in JSX and JAVA files that are used by a web service.

To cause your web service to emit messages to the jws.log file, create a Logger for your web service. You may create a Logger by calling the JwsContext.getLogger method, typically with the qualified name of your web service's class as the Logger name (e.g. "async.HelloWorldAsync" for the samples web service async/HelloWorldAsync.jws. Then simply call the Logger's debug, info, warn or error methods as appropriate.

The example code below illustrates use of logging in a JWS file. Portions of the source file have been omitted for brevity.

```
package async;
import weblogic.jws.control.JwsContext;
import weblogic.jws.control.TimerControl;
import weblogic.jws.util.Logger;



public class HelloWorldAsync
{
    /** @jws:context */
    JwsContext context;
    /**
     * @jws:operation
     * @jws:conversation phase="start"
     */
    public void HelloAsync()
    {
        Logger logger = context.getLogger("async.HelloWorldAsync");
        logger.debug("about to start timer");
        // all we do here is start the timer.
        helloDelay.start();
        logger.debug("timer started");
        return;
    }
    private void helloDelay_onTimeout(long time)
    {
        Logger logger = context.getLogger("async.HelloWorldAsync");
        // send the client a hello message.
        logger.debug("in timer handler: calling client");
        callback.onHelloResult("Hello, asynchronous world");

        // we don't want any more timer events for this conversation.
        logger.debug("in timer handler: stopping timer");
        helloDelay.stop();

        return;
    }
}
```

The code above results in the following content in jws.log when the HelloAsync method of HelloWorldAsync.jws is invoked.

```
20 May 2002 13:54:13,466 DEBUG HelloWorldAsync: about to start timer
20 May 2002 13:54:13,559 DEBUG HelloWorldAsync: timer started
20 May 2002 13:54:18,934 DEBUG HelloWorldAsync: in timer handler: calling client
20 May 2002 13:54:18,981 DEBUG HelloWorldAsync: in timer handler: stopping timer
```

Related Topics

Configuration File Reference

weblogic−jws−config.xml Configuration File

The weblogic−jws−config file allows you to configure runtime parameters of web services, both on the development server and production servers. This file is used by JwsCompile when preparing WebLogic Workshop projects for deployment to a production server.

Elements

The following xml elements may appear in weblogic−jws−config.xml

<config>

The <config> element is the root element of the weblogic−jws−config.xml file. All other elements are children or grandchilden of <config>.

<protocol> (global)

The global <protocol> element appears as the immediate child of the <config> element. The global <protocol> element defines the default exposure protocol for your web application. Unless otherwise specified in the jws−specific <protocol> element, web resources within the web application, including web services, will be exposed on the protocol specified. Possible values are http or https.

<protocol> (jws−specific)

The jws−specific <protocol> element appears as the immediate child of the <jws> element. The jws−specific <protocol> element overrides the value of global <protocol> element. Possible values are http or https.

<hostname>

The <hostname> element specifies the name of the machine where your web application will be deployed.

<http−port>

The <http−port> element specifies which port should be used for http traffic.

<https−port>

The <https−port> element specifies which port should be used for https traffic.

<jws>

The <jws> tag takes a pair of <class−name> / <protocol> elements as children. For example,

```
<jws>

  <class−name>HelloWorld</class−name>

  <protocol>http</protocol>
```

```
</jws>

<jws>

  <class−name>HelloWorldSecure</class−name>

  <protocol>https</protocol>

</jws>
```

Use the <jws>, <class−name> and <protocol> elements to specify the transport protocol for individual web services in a web application.

<class−name>

The <class−name> element is used in conjunction with the <jws> and <protocol> elements to specify the transport protocols of individual web services with a web application. Possible values are the names of web services within the web application. For example, to specify a web service Foo.jws, use <class−name>Foo</class−name>. Values of the <class−name> element are case sensitive.

<transaction−isolation−level>

The <transaction−isolation−level> tag is used to specify the transaction isolation level of the EJB's that back JWS's. The value will of <transaction−isolation−level> be mapped to the EJBs' deployment descriptors during a build. Default settings are TRANSACTION_READ_COMMITTED for a production build (compiling an EAR for deployment) and TRANSACTION_SERIALIZABLE for a non−production build (compiling to run in Test View). Valid values are

TRANSACTION_READ_COMMITTED

TRANSACTION_READ_UNCOMMITTED

TRANSACTION_REPEATABLE_READ

TRANSACTION_SERIALIZABLE

<ejb−concurrency−strategy>

The <ejb−concurrency−strategy> element is used to specify the ejb concurrecny strategy of the EJBs that back JWSs. Valid values are

Exclusive

Database

Optimistic

Default values are Exclusive for non−production builds, and Database for production builds. For clustered environments, a value of Database is required. Non−clustered environments can use Exclusive.

<file−filter>

The <file−filter> element includes an arbitrary number of child <include−files> and <exclude−files> elements (<include−files> and <exclude−files> elements in turn can include an arbitrary number of child <rule>

elements). Use the nested <file−filter><include−files><rule> elements nested to specify files to be included in deployment EARS. Use the nested <file−filter><exclude−files><rule> elements nested to specify files to be excluded from deployment EARS. See the <rule> element for details on including and excluding files from deployment EARS.

<include−files>

Use the <include−files> element to include specific files in deployment EARS. The <include−files> element is a child element to the <file−filter> element. The <include−files> element can have an arbitrary number of child <rule> elements. See the <rule> element for details on including files in deployment EARS.

<exclude−files>

Use the <exclude−files> element to exclude specific files from deployment EARS. The <exclude−files> element is a child element to the <file−filter> element. The <exclude−files> element can have an arbitrary number of child <rule> elements. See the <rule> element for details on excluding files from deployment EARS.

<rule>

Use the <rule> element to specify individual files for exclusion or inclusion from a deployment EAR file. When jwsCompile produces a deployment EAR it checks the weblogic−jws−config.xml file to see which files in the web service project to include and exclude. By default the following file types are excluded from deployment EARS: JWS, CTRL, WSDL, JAVA, JSX, XMLMAP, CLASS. You must explicitly override the default to include these file types in deployment EARS. You can override the default behavior either by including <include−files> and <rule> elements, or by using the −x parameter of the EAR generation tool jwsCompile (see JwsCompile Command for details).

The value of the <rule> element is a file pattern relative to the project root. For example, <rule>/misc/*.txt</rule> can be used to include or exclude all TXT files in the /misc folder from the deployment EAR. The following example shows how to use the <file−filter>, <include−files>, <exclude−files>, and <rule> elements to include and exclude files and file patterns from a deployment EAR:

```
<file−filter>

    <include−files>

        <rule>/Services.h</rule>

        <rule>/PageBuffer.java</rule>

        <rule>/wsdl/StockTrader.wsdl</rule>

    </include−files>

    <include−files>

        <rule>/etc/*</rule>

    </include−files>

    <exclude−files>
```

```
            <rule>*.h</rule>

            <rule>*.java</rule>

        </exclude−files>

    </file−filter>
```

Related Topics

JwsCompile Command

.Workshop Configuration File

The .Workshop file specifies user−specific configuration parameters for the WebLogic Workshop visual development environment. Global properties (on this machine) of the WebLogic Workshop development environment are stored in the Workshop.properties Configuration File. Any property described in Workshop.properties may be overridden by specifying a value for that property in the .Workshop file.

The .Workshop file is located in the user's home directory. On Windows systems, the user's home directory is indicated by the %USERPROFILE% environment variable. On Unix and Linux systems, the .Workshop file is located in the ~ (tilde) directory (sometimes also represented in the environment by $HOME).

Each property that may be set is listed below with its default value.

In addition to the properties listed here, the .Workshop file also stores the most recent WebLogic Workshop visual development environment user interface position and size information.

.Workshop is a Java property file that is read as a java.util.PropertyResourceBundle. As such, the following characters must be escaped with a backslash ('\') if used in a property value: {, }, =, :, \, and EOL.

The .Workshop file for the current user is overwritten every time a WebLogic Workshop visual development environment session is ended. It is recommended that you use the Preferences Dialog to modify settings whenever possible.

Properties

recent.lastProject=C\:\\bea\\weblogic700\\samples\\workshop\\applications\\samples

Specifies the project opened in the previous WebLogic Workshop visual development environment session. This value overrides the same property in the Workshop.properties Configuration File.

recent.lastFiles=C\:\\bea\\weblogic700\\samples\\workshop\\applications\\samples\\HelloWorld.jws

Specifies the list of files that were open in the previous WebLogic Workshop visual development environment session. File names are separated by semicolons. This value overrides the same property in the Workshop.properties Configuration File.

recent.projectN=<project path>

The Nth most recently opened project.

recent.fileN=<file path>

The Nth most recently opened file.

source.editing.overwrite=0

Specifies whether the Source View editor was most recently in insert or overwrite mode.

Related Topics

Configuration File Reference

Workshop.properties Configuration File

Workshop.properties Configuration File

Note: The Workshop.properties file is required for proper operation of WebLogic Workshop. The visual development environment will not function if this file is corrupted or deleted. Use caution if you choose to edit this file, and make a backup copy of the original version.

All properties in this file may be overridden on a per−user basis by specifying a value for that property in the user's .Workshop file. It is recommended that changes be made to the .Workshop file instead of this file. See .Workshop Configuration File.

The Workshop.properties file specifies global (for this machine) configuration properties for the WebLogic Workshop visual development environment. Each property that may be set is listed below with its default value.

Workshop.properties is located in BEA_HOME\weblogic700\workshop.

Workshop.properties is a Java property file that is read as a java.util.PropertyResourceBundle. As such, the following characters must be escaped with a backslash ('\') if used in a property value: {, }, =, :, \, and EOL.

Properties

controls.applicationView.enabled=1

Determines whether the AppView control is enabled in the WebLogic Workshop visual development environment. Note that the AppView control require an additional license. The AppView control is pre−configured in the workshop sample domain.

defaultJndiDataSource=cgSampleDataSource

The JDBC data source that is used by default by new Database controls.

paths.serverRoot=localhost

Specifies the hostname that is used in communication with WebLogic Server. May be set via the "Name" setting on the Paths page of the Preferences Dialog.

paths.port=7001

Specifies the port that is used in communication with WebLogic Server. May be set via the "Port" setting on the Paths page of the Preferences Dialog.

paths.rootDirectory=C\:\\bea\\weblogic700\\samples

In association with paths.domain, determines the path to the domain directory of the WebLogic Server configuration against which WebLogic Workshop runs. May be set via the "Config directory" setting on the Paths page of the Preferences Dialog.

To use a remote WebLogic Server, mount or map the parent of the server's domain directory to a local mount point or drive letter, then specify that mount point or drive letter here.

paths.domain=workshop

In association with paths.rootDirectory, determines the path to the domain directory of the WebLogic Server configuration against which WebLogic Workshop runs. May be set via the "Domain" option menu on the Paths page of the Preferences Dialog.

paths.browser=C\:\\Program Files\\Internet Explorer\\IExplore.exe

Specifies the path to the browser application that is launched when the "Debug−>Start" or "Debug−>Start and Debug" menu actions are selected.

paths.httpRoot=http\://localhost\:7001

Specifies the root of the URL used to test web services when the "Debug−>Start" or "Debug−>Start and Debug" menu actions are selected.

paths.startCmd=startWebLogic

Specifies the command used to start WebLogic Server when the "Tools−>Start WebLogic Server" menu action is selected.

paths.stopCmd=stopWebLogic

Specifies the command used to stop WebLogic Server when the "Tools−>Start WebLogic Server" menu action is selected.

paths.classPath=C\:\\bea\\jdk131_02\\jre\\lib\\rt.jar;

C\:\\bea\\weblogic700\\workshop\\stdlib.jar;

C\:\\bea\\weblogic700\\server\\lib\\weblogic.jar

Specifies the class path that the WebLogic Workshop visual development environment uses to find classes. Used in code completion and error highlighting. Add class files or JAR files to this path to enable code completion and recognition of additional Java classes.

The separator between class path entries is operating system specific. Use semicolons on Windows systems and colons on Unix and Linux systems.

recent.lastFiles=C\:\\bea\\weblogic700\\samples\\workshop\\applications\\samples\\HelloWorld.jws

Specifies the default list of files opened in the previous WebLogic Workshop visual development environment session. File names are separated by semicolons. This value is overridden by the same property in the .Workshop file in the user's home directory. See .Workshop Configuration File.

recent.lastProject=C\:\\bea\\weblogic700\\samples\\workshop\\applications\\samples

Specifies the default project opened in the previous WebLogic Workshop visual development environment session. This value is overridden by the same property in the .Workshop file in the user's home directory. See .Workshop Configuration File.

source.display.fontName=Monospaced

Specifies the font used in Source View. May be set via the "Font" setting in the "Source view font" section of the Display page of the Preferences Dialog.

windows.font=SansSerif

Specifies the font used for the WebLogic Workshop visual development environment's user interface elements (menus, dialogs, etc.) in non−English locales.

windows.font_en=Tahoma

Specifies the font used for the WebLogic Workshop visual development environment's user interface elements (menus, dialogs, etc.) in English locales.

Related Topics

Preferences Dialog

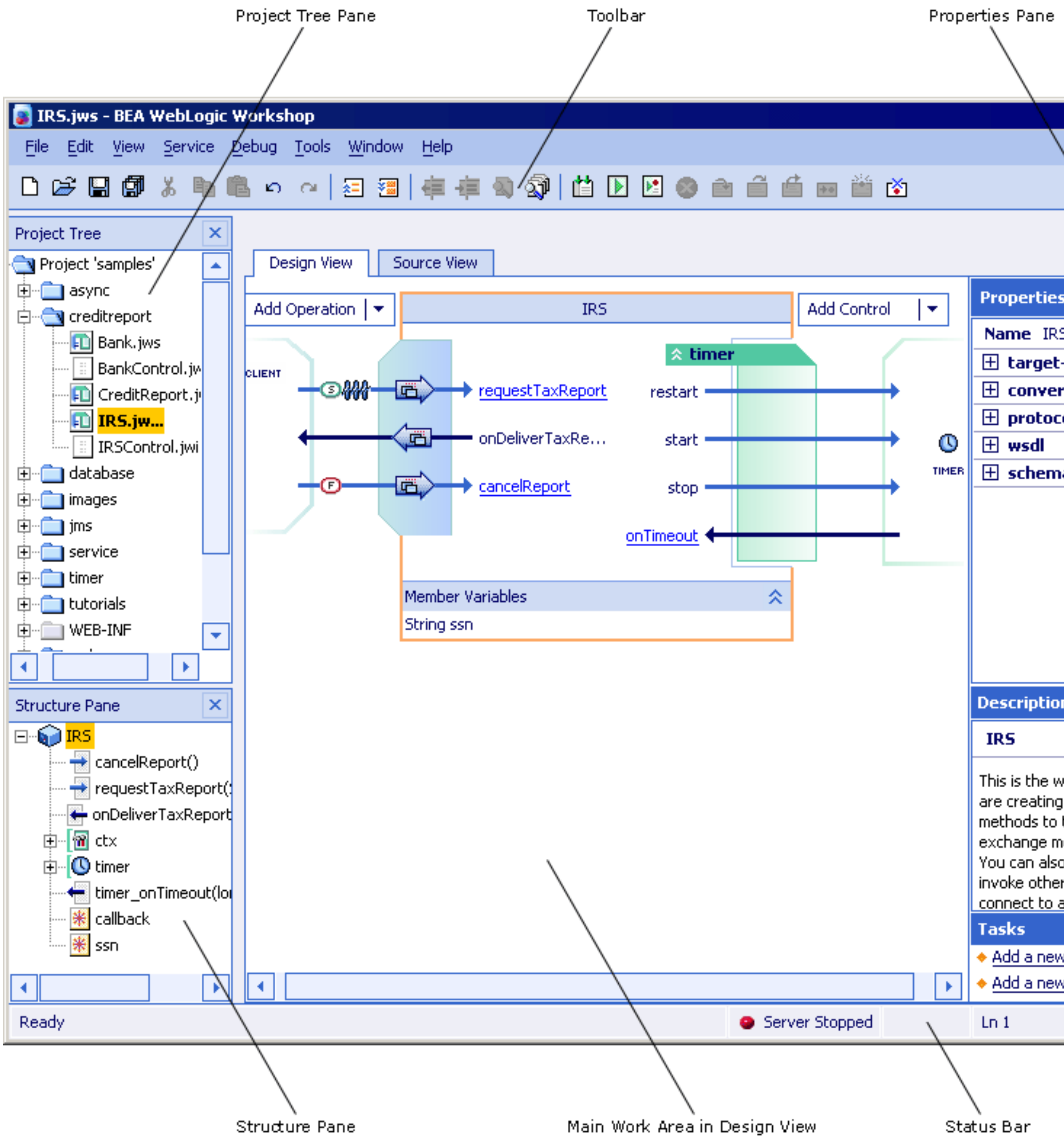
.Workshop Configuration File

Configuration File Reference

User Interface Reference

This section explains the graphical aspects of WebLogic Workshop's visual development environment.

The image below shows the six major regions of the development environment: the Project Tree, Toolbar, Properties Editor, Structure Pane, Design View, and the Status Bar. For detailed explanations of each region see the individual topics listed below.



Topics Included in This Section

[Design View]()

[Project Tree]()

[Properties Editor]()

[Source View]()

Status Bar

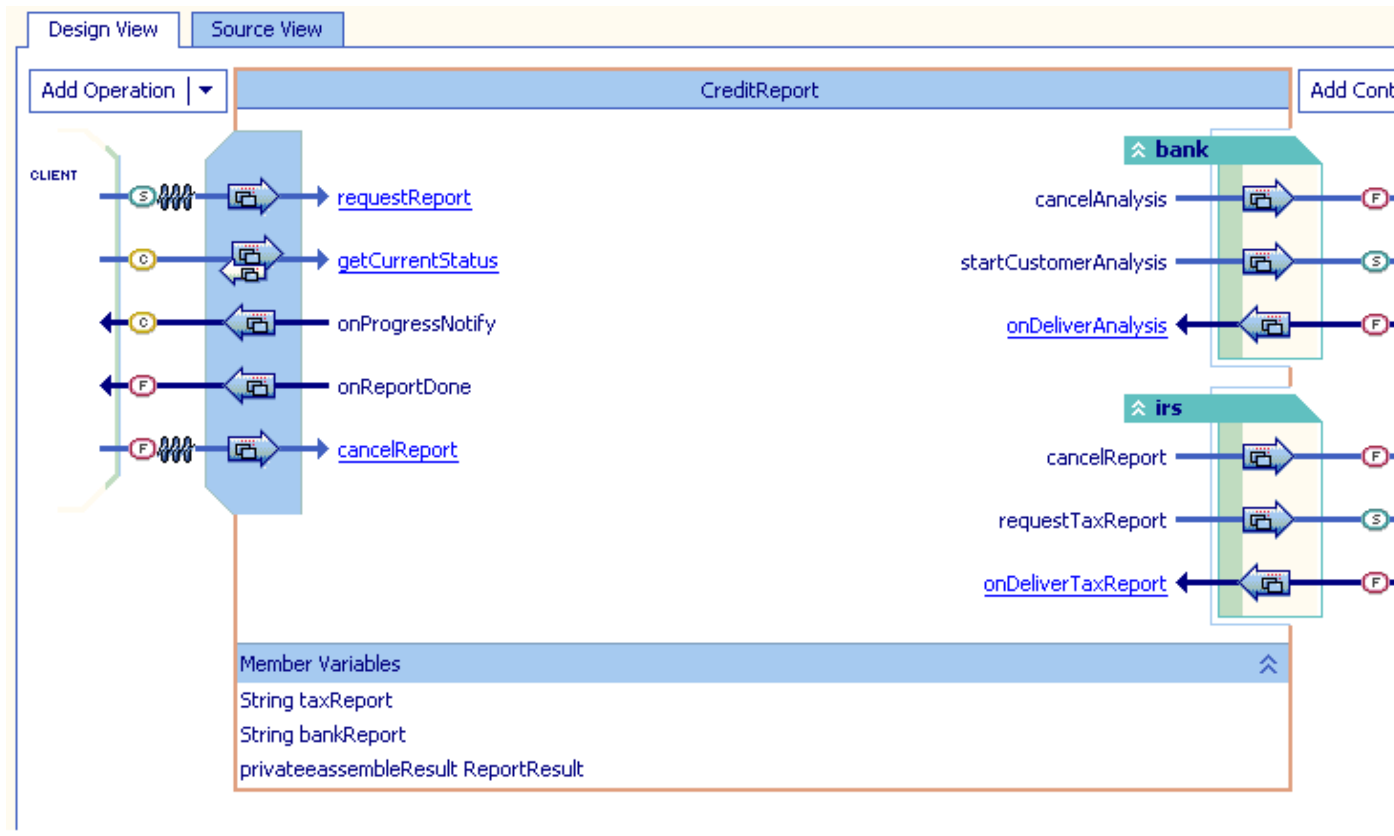
Structure Pane

Toolbar

Preferences Dialog

Design View

Design View provides a graphical representation of your web service, giving you a unified picture of how your web service is put together and how it interacts with its client application and external resources, such as databases and other web services. Note that only JWS files can be displayed in Design View; CTRL files only appear as components of the web service displayed in Design View.



Design View Basics

In Design View, the web service itself is represented by a box in the center of the screen. The client application is represented on the left side of the web service. Interactions between the web service and the client are represented by arrows. Arrows pointing from the client to the web service represent the operations the client may invoke on the web service; arrows pointing from the web service to the client represent the callbacks through which the web service sends asynchronous messages to the client.

The different types of external resources available to the web service are represented on the right side of the web service. These resources can be databases, Enterprise Java Beans and even other web services. Controls, the interfaces between a web service and its external resources, are represented by arrows linking the web service and the resource. Arrows pointing from the web service to a control represent the Java methods through which a web service invokes the control; arrows pointing from the a control to the web service represent callbacks the control may use to notify the web service of events.

Building and Editing Components in Design View

To add an interface between the client and the web service, click on the dropdown box labeled "Add Operation". To add a control, click on the dropdown box labeled "Add Control".

Once you have added a new method, callback or control, you can edit its properties through the Properties Editor. For finer control over the functionality of your web service, switch to Source View and edit the underlying code directly. To view the source code for a particular method, click on the method's name. To view the source code for a particular control, right−click on the name of the control and select 'Go to .ctrl definition'.

XML Maps

Because a web service built using the WebLogic Workshop is, essentially, a Java class which communicates with the external world via XML (Extensible Markup Language) messages, the arrows also represent the Java/XML interfaces, called "XML maps", that your web service implements. To view and edit a particular XML map, double click the image of the arrow. For more information about the basic architecture of web service communications, see "What Are Web Services?" For more information about XML maps see Why Use XML Maps?

Conversations and Buffers

Ovals containing the letters "S", "C" and "F" show how your web service supports conversations. The presence of an oval containing "S" means that the method starts a conversation. An oval containing the letter "C" means the method or callback continues a conversation. An oval containing the letter "F" means the method or callback finishes a conversation. Use conversations to manage asynchronous exchanges between client and service and service and controls.

In the illustration above the methods and callbacks that link the client and the service are participating in a conversation; the methods and callbacks that link the controls to the service are also conversational. For more information about conversations see Overview: Conversations. For instructions on how to add conversations to a web service displayed in Design View see How Do I: Add Support for Conversations?

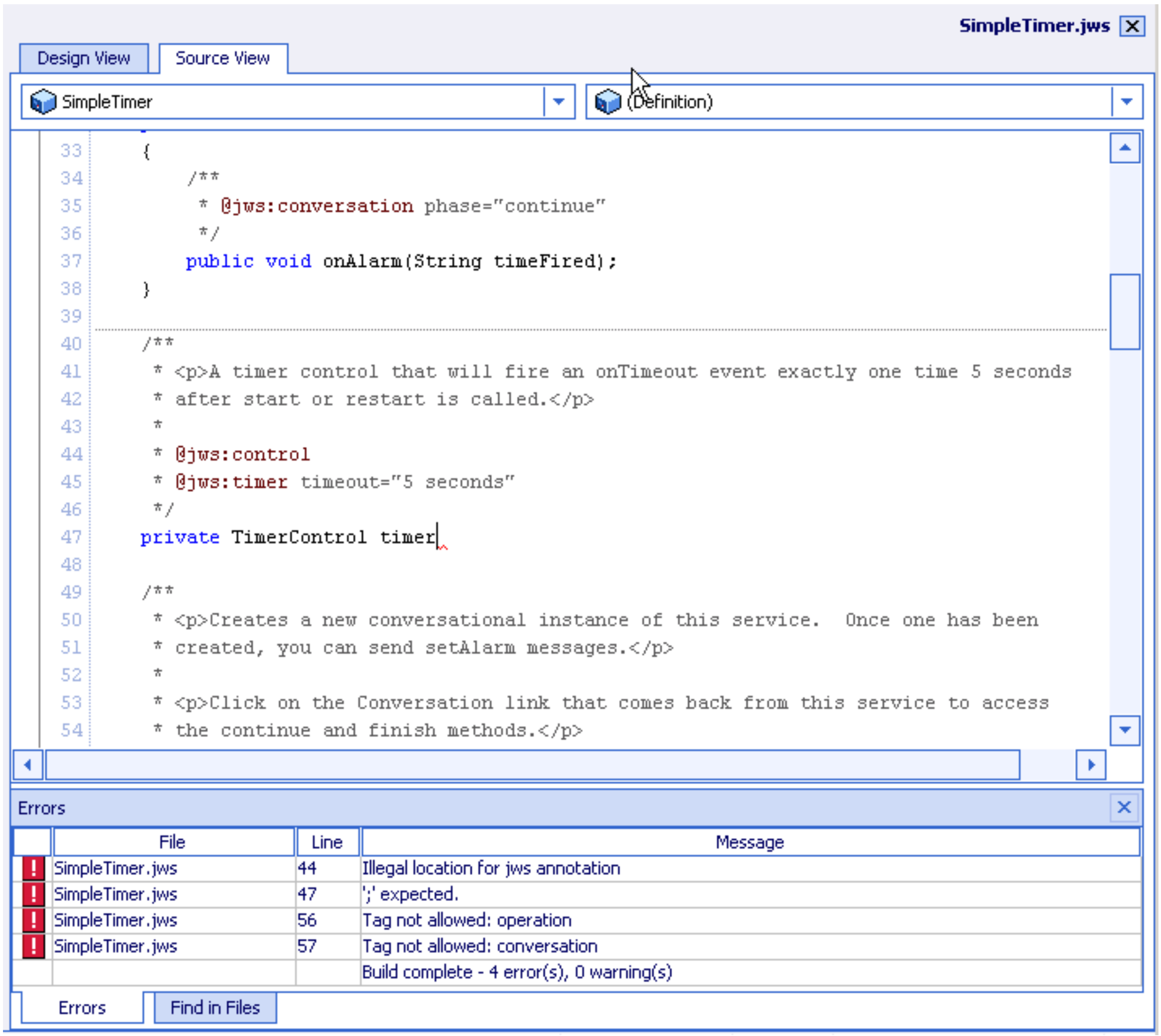
Message queues, a.k.a. "buffers", are represented by springs. In the illustration above the methods requestReport and cancelReport are buffered. Use buffers to avoid overloading the servers running your web service, as well as to prevent clients from blocking waiting for long−running operations. For more information about message queuing and buffers see Customizing a Service Control: Buffering Methods and Callbacks and Using Asynchrony to Enable Long−Running Operations.

Related Topics

Step 2: Tour WebLogic Workshop

Error Pane

When you build a web service, WebLogic Workshop automatically switches to Source View and displays the Error Pane at the bottom. The Error Pane is used to report errors during build. Each error displays the filename containing the error, the line number of the error and the error message, as shown below.



```
    {
        /**
         * @jws:conversation phase="continue"
         */
        public void onAlarm(String timeFired);
    }

    /**
     * <p>A timer control that will fire an onTimeout event exactly one time 5 seconds
     * after start or restart is called.</p>
     *
     * @jws:control
     * @jws:timer timeout="5 seconds"
     */
    private TimerControl timer

    /**
     * <p>Creates a new conversational instance of this service.  Once one has been
     * created, you can send setAlarm messages.</p>
     *
     * <p>Click on the Conversation link that comes back from this service to access
     * the continue and finish methods.</p>
```

Errors

	File	Line	Message
❗	SimpleTimer.jws	44	Illegal location for jws annotation
❗	SimpleTimer.jws	47	';' expected.
❗	SimpleTimer.jws	56	Tag not allowed: operation
❗	SimpleTimer.jws	57	Tag not allowed: conversation
			Build complete - 4 error(s), 0 warning(s)

Many errors have associated prescriptions, which can help you determine how to fix the error. To see the prescription for an error, hold the cursor over the error entry in the Error Pane and wait for the tooltip containing the prescription to appear.

If you double−click on an error entry in the Error Pane, WebLogic Workshop will open the appropriate file and position the cursor on the line on which the error occurred.

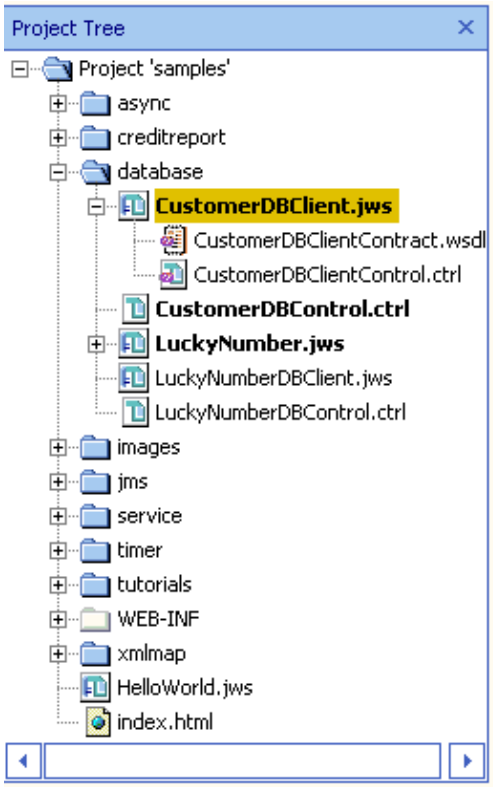
Related Topics

Source View

User Interface Reference

Project Tree

The Project Tree displays the files of your web service project organized in a directory structure.



Drop and Drag

The Project Tree supports drop and drag functionality. You can rearrange the file and directory structure by selecting and dragging elements within the Project Tree. You can also drag files and folders from your local file system into the Project Tree.

Note that controls can be added to a web service by dragging a CTRL file from the Project Tree into the main work area displayed in design mode.

Autogeneration of WSDL and CTRL Files

To autogenerate a WSDL or CTRL file from a JWS file, right click on the appropriate JWS file and select either Generate WSDL from JWS or Generate CTRL from JWS. When a WSDL or CTRL file is autogenerated in this way, the autogenerated file appears indented underneath the originating JWS file. Autogenerated files are automatically updated when changes are made to their originating JWS file. However, if an autogenerated file is at any time edited directly, it looses its ability to be automatically updated.

An asterisk (*) appearing to the right of a file in the Project Tree indicates the file has been changed but not yet saved.

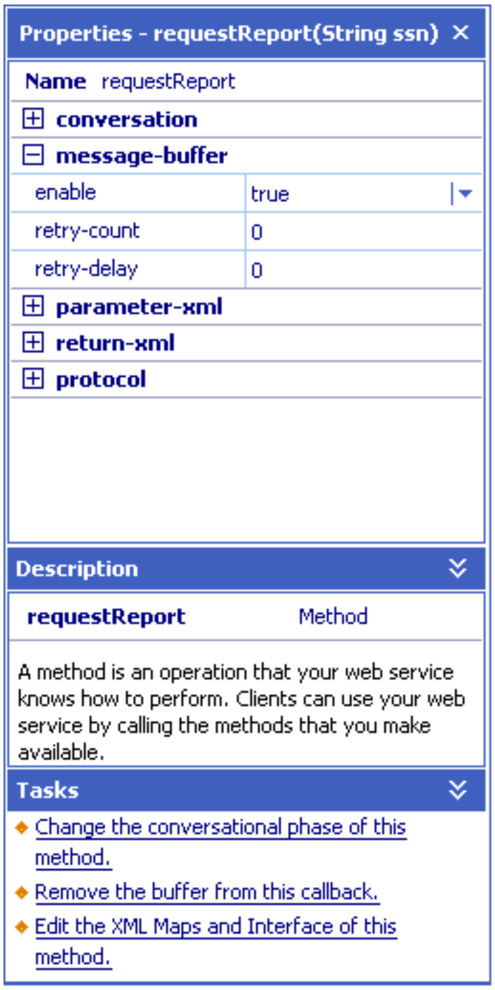
Related Topics

JWS Files: Java Web Services

CTRL Files: Implementing Controls

WSDL Files: Web Service Descriptions

Properties Editor



Properties - requestReport(String ssn)

Name	requestReport
⊞ conversation	
⊟ message-buffer	
enable	true
retry-count	0
retry-delay	0
⊞ parameter-xml	
⊞ return-xml	
⊞ protocol	

Description

requestReport Method

A method is an operation that your web service knows how to perform. Clients can use your web service by calling the methods that you make available.

Tasks

- Change the conversational phase of this method.
- Remove the buffer from this callback.
- Edit the XML Maps and Interface of this method.

Instead of working directly with a web service's source code, you can use the Properties Editor to build and edit your web service. The contents of the Properties Editor change depending on which web service element is selected in Design View. For example, when a method is selected, the Properties Editor displays the method's conversational, buffer, and XML related properties; when a database control is selected, the Properties Editor displays the database's data source information. (Note that selecting items in the Structure Pane also changes the contents of the Properties Editor.)

You can edit an element's properties directly from the Properties Editor; any changes made in the Properties Editor will be reflected in the source code automatically.

The Description Pane provides a synopsis of the item currently selected in Design View. The Description Pane is good place to learn about the different kinds of elements within a web service. (You can collapse or expand the Description Pane by clicking the chevron in the upper right hand corner.)

The Tasks Pane provides a list of possible actions for the currently selected item in Design View. Click on an action from the list to invoke the action.

Related Topics

Design View

Structure Pane

Source View

Any file which contains text can be viewed and edited in Source View. You can view a file in Source View by clicking the Source View tab, or, to navigate to a particular method within a web service, double click the method name in Design View. Once a file is displayed in Source View, you can navigate to a particular method by selecting from the dropdown lists directly below the Design and Source View tabs. Navigating to particular method causes that method to be highlighted with a yellow background color.

Code Editing Features

Source View provides the following code editing features.

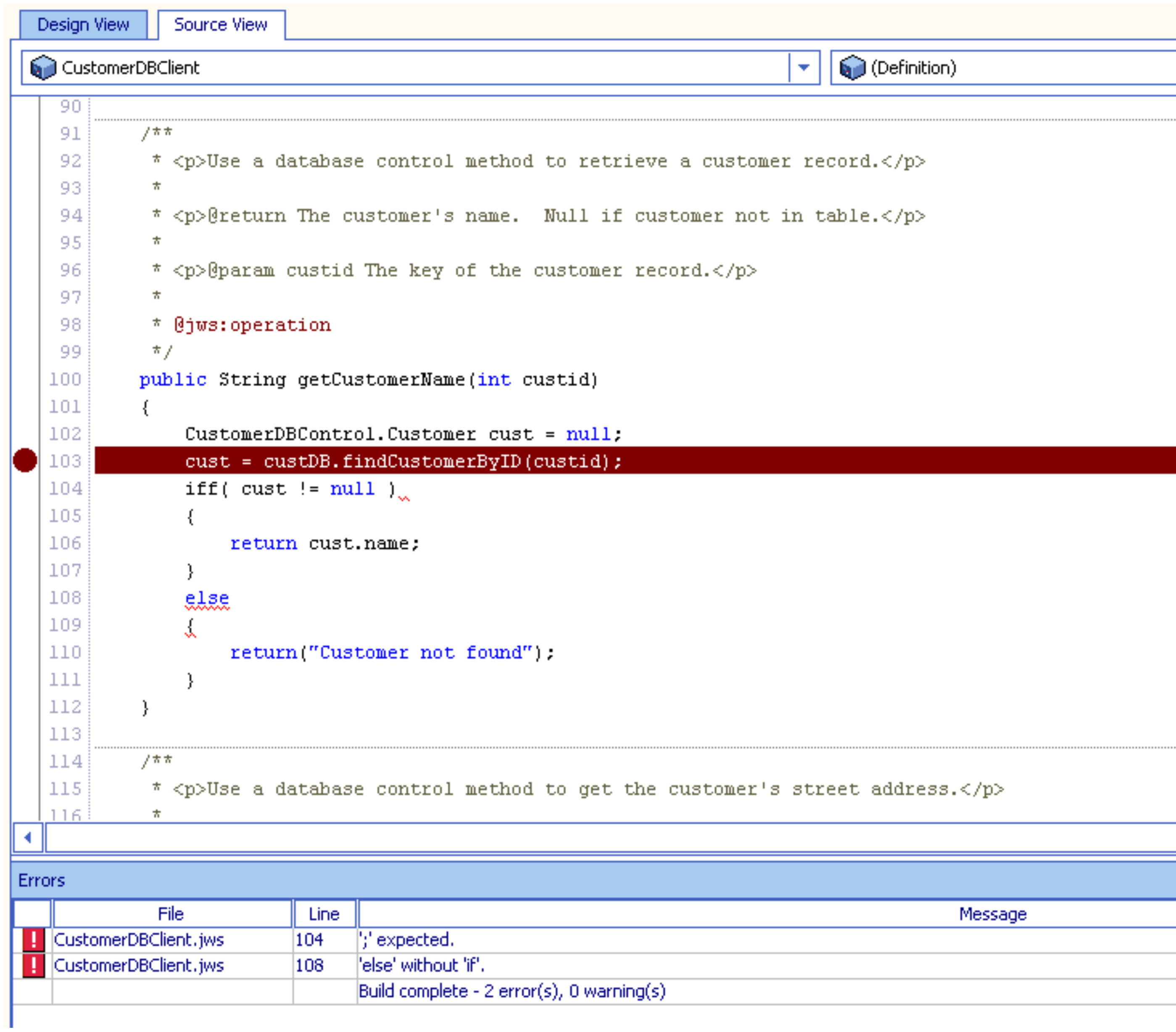
Java class files (JWS, CTRL and JAVA files) are parsed and color coded to highlight the various elements of the source code. For example, keywords are colored blue, comments are colored grey, and @jws annotations are colored red. You can customize the color coding through the Preferences dialog. Access the Preferences dialog by selecting Preferences from the Tools menu.

Code completion and error highlighting are performed for classes that on a class path used by the WebLogic Workshop visual development environment. The class path is defined by the paths.classPath property in the Workshop.properties file. If you would like Source View to provide code completion and error highlighting for additional classes, edit the Workshop.properties file to add class files or JAR files to the paths.classPath, then restart the WebLogic Workshop visual development environment. To learn more about WebLogic Workshop properties, see Workshop.properties Configuration File.

Breakpoints can be set by clicking in the empty column to the right of the line numbers. Breakpoints are indicated by a red dot and red background highlighting. Breakpoints can be removed by clicking on the red dot.

When you compile a file displayed in Source View, the Error Pane will appear at the bottom of the screen. If any error message occur, double clicking on a particular message moves the cursor next to the offending code.

Red squiggly lines appear underneath code which is syntactically incorrect. Hover the cursor over the red squiggly lines to receive hints about how to correct the syntax.



```
    /**
     * <p>Use a database control method to retrieve a customer record.</p>
     *
     * <p>@return The customer's name.  Null if customer not in table.</p>
     *
     * <p>@param custid The key of the customer record.</p>
     *
     * @jws:operation
     */
    public String getCustomerName(int custid)
    {
        CustomerDBControl.Customer cust = null;
        cust = custDB.findCustomerByID(custid);
        iff( cust != null )
        {
            return cust.name;
        }
        else
        {
            return("Customer not found");
        }
    }

    /**
     * <p>Use a database control method to get the customer's street address.</p>
     *
```

	File	Line	Message
!	CustomerDBClient.jws	104	';' expected.
!	CustomerDBClient.jws	108	'else' without 'if'.
			Build complete - 2 error(s), 0 warning(s)

Related Topics

Design View

Status Bar

The Status Bar provides information about the current state of the WebLogic Workshop visual development environment and the WebLogic Server.

Main Status Area

The Main Status Area displays the current state of the WebLogic Workshop visual development environment. Possible messages are listed below.

- Ready Indicates that WebLogic Workshop is ready to execute the next task.
- Saving [filename] Indicates that the WebLogic Workshop is busy saving a file.
- Build Started Indicates the WebLogic Workshop is busy compiling the current web service.
- Build Completed Indicates that the current file was successfully compiled.

- Build Contained Errors Indicates that the current file could not be successfully compiled because it contained errors. Examine the Errors Pane for specific error information.

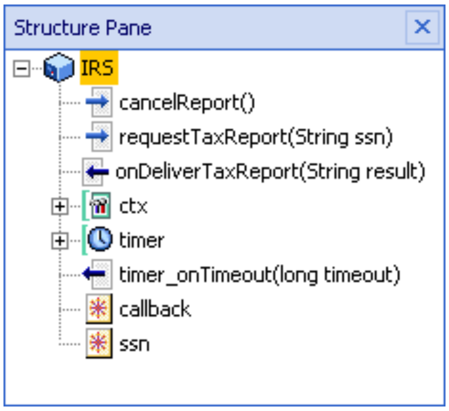
Server Status Area

The Server Status Area displays the current state of the WebLogic Server. (The WebLogic Server is responsible for running a finished web service built with the WebLogic Workshop.) When the WebLogic Server is running, the Server Status Area will display a green ball followed by the message "Server Running". When the WebLogic Server is not running, the Server Status Area will display a red ball followed by the message "Server Stopped". To turn the WebLogic Server on or off, click the Tools menu and select either Start WebLogic Server or Stop WebLogic Server.

Structure Pane

The Structure Pane displays a synopsis of the contents of the current file including methods, callbacks, controls, and member variables.

You can display file contents in either alphabetical order or according to the order in which they appear in the file. To change the display order of the structure pane select Preferences from the Tools menu.


Structure Pane
IRS
cancelReport()
requestTaxReport(String ssn)
onDeliverTaxReport(String result)
ctx
timer
timer_onTimeout(long timeout)
callback
ssn

You can also navigate to different parts of the current file by double clicking on the elements in the Structure Pane. If you want to see myMethod() in the currently displayed file, double click on myMethod() in the Structure Pane, and that method will be displayed with yellow highlighting in Source View.

Test View

The Test View page is loaded in your web browser when you build and start your service. You can use Test View to test the public methods of your service.

Test View has four tabs that provide information about your service:

- Overview
- Console
- Test Form
- Test XML
- Warnings

By default, the Test Form tab is selected when the page loads.

In addition, each page lists the address of the service being tested. Each part of this address is a link to the JwsConsole, a page that lists the files in each level of the project directory.

- JwsConsole

Overview Tab

The Overview tab displays public information about your service, including:

- The WSDL that describes the complete public contract for your web service. Clients of your web service use the WSDL file to determine how to call your web service and what data will be returned. The WSDL exposes both methods and callbacks on the service.
- The callback WSDL for clients that cannot handle the callbacks described in the complete WSDL. Clients wishing to receive callbacks by implementing a callback listening service can communicate with your service using this WSDL.
- The Java source code for a WebLogic Workshop service control for the service. A developer using WebLogic Workshop who wishes to call your service from theirs can use this source to construct a service control.
- The Java proxy for calling your service from Java code. For example, a Java client can call your web service by creating a class from the Java proxy and importing it.
- The service description for your service, which lists the service's available methods and callbacks.
- Links to useful information such as specifications for WSDL and SOAP.

Note: To learn how the comments associated with each method are obtained, and how you can document your web service's methods, see Documenting Web Services.

Console Tab

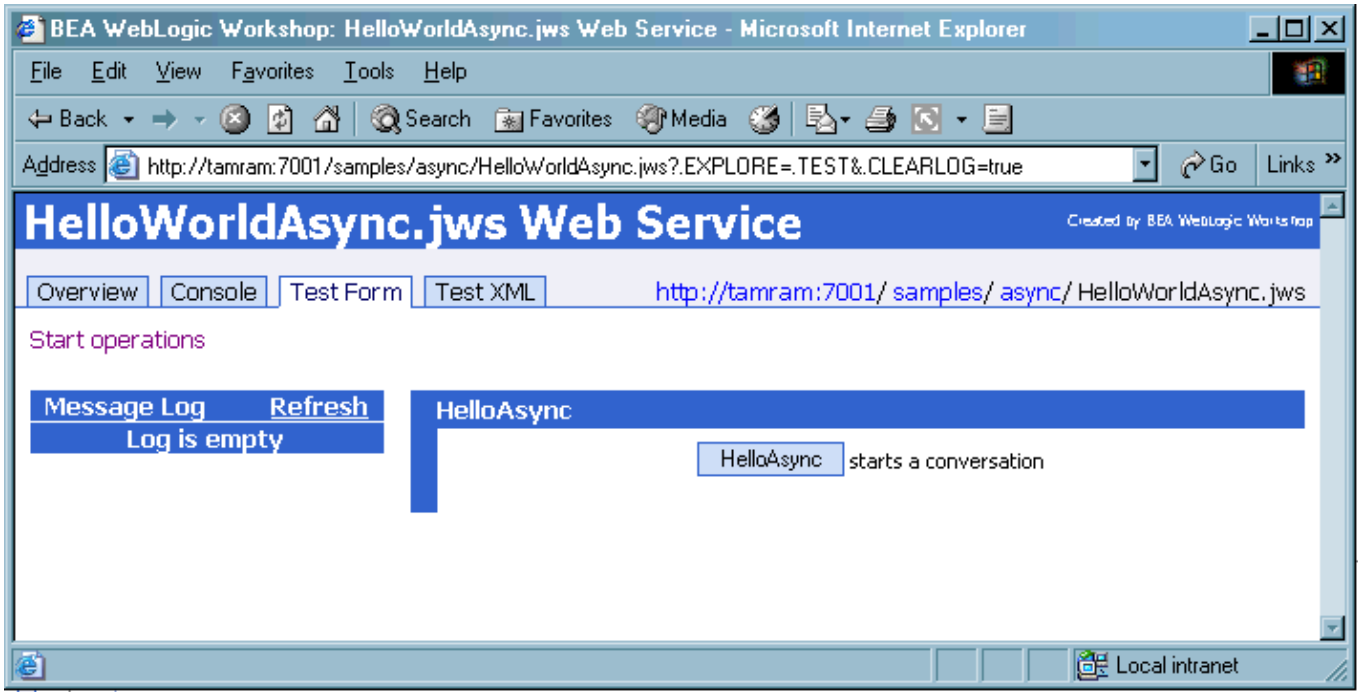
The Console tab displays private information about your service, including:

- How the service is implemented on the back end, and with what version of WebLogic Workshop it was created. For example, services built using WebLogic Workshop are implemented using Enterprise JavaBeans; they may also use other J2EE components.
- Links to the WebLogic Server console application.
- Settings for the message log on the Test Form tab.
- Persistence settings, for clearing the conversational state and logs for the service, and for redeploying the Enterprise JavaBeans underlying the service.

Test Form Tab

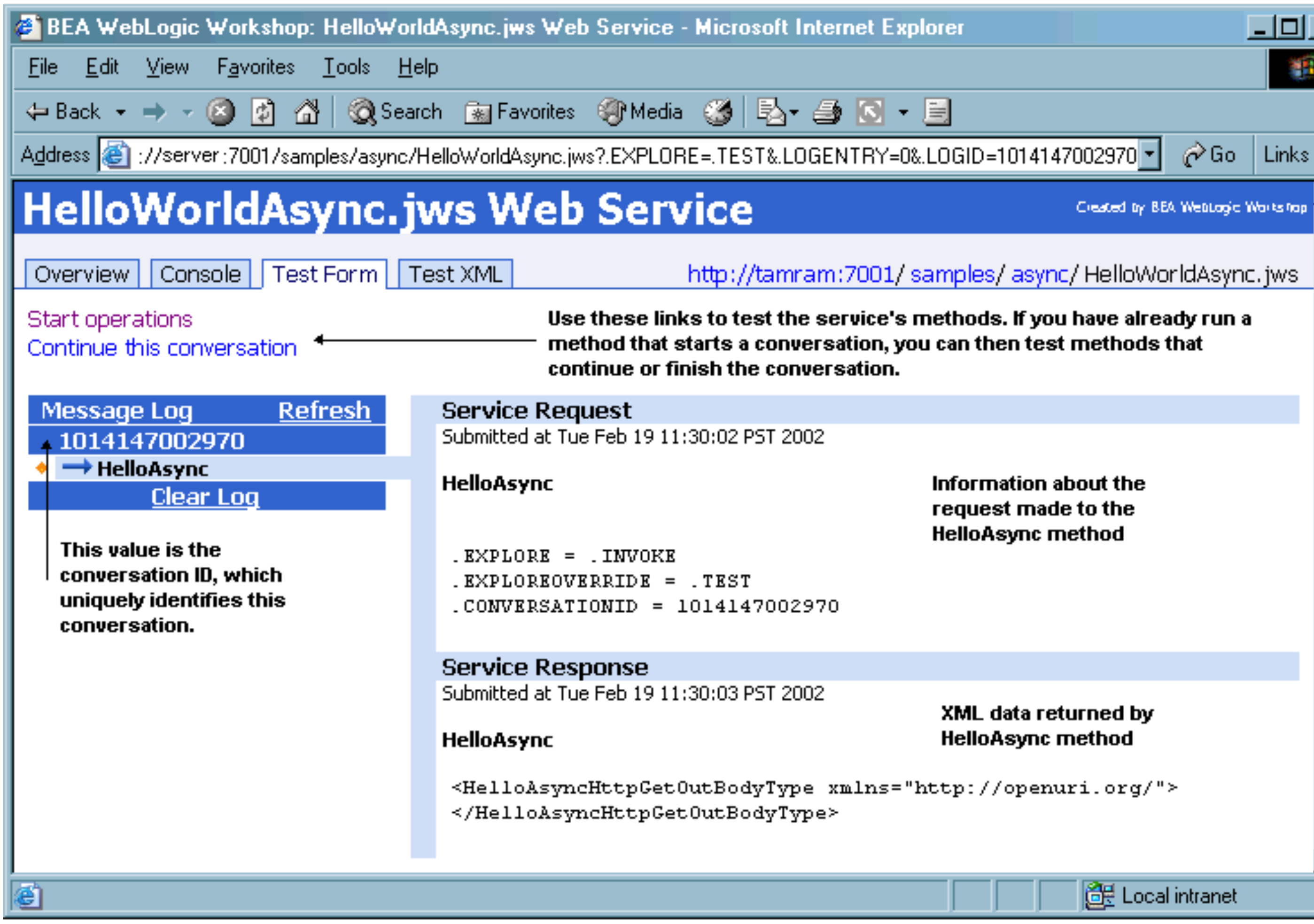
The Test Form tab provides a simple test environment for the public methods of your service. You can provide parameters for a method and examine its return value. You can also track and test the different parts of a conversation.

The following image shows how the Test Form tab appears for a service called HelloWorldAsync, which demonstrates a simple conversation:


BEA WebLogic Workshop: HelloWorldAsync.jws Web Service - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://tamram:7001/samples/async/HelloWorldAsync.jws?.EXPLORE=.TEST&.CLEARLOG=true
Go Links
HelloWorldAsync.jws Web Service
Created by BEA WebLogic Workshop
Overview Console Test Form Test XML
http://tamram:7001/ samples/ async/ HelloWorldAsync.jws
Start operations
Message Log Refresh
Log is empty
HelloAsync
HelloAsync starts a conversation
Local intranet

To test the service, click the HelloAsync button. If this method took parameters, you would enter values for them here.

The Test Form page displays information about the service request and response, including the XML message that was returned, as shown in the following image:


BEA WebLogic Workshop: HelloWorldAsync.jws Web Service - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address ://server:7001/samples/async/HelloWorldAsync.jws?.EXPLORE=.TEST&.LOGENTRY=0&.LOGID=1014147002970
Go Links
HelloWorldAsync.jws Web Service
Created by BEA WebLogic Workshop
Overview Console Test Form Test XML
http://tamram:7001/ samples/ async/ HelloWorldAsync.jws
Start operations
Continue this conversation
Message Log Refresh
1014147002970
HelloAsync
Clear Log
Local intranet

Use these links to test the service's methods. If you have already run a method that starts a conversation, you can then test methods that continue or finish the conversation.

This value is the conversation ID, which uniquely identifies this conversation.

Service Request

Submitted at Tue Feb 19 11:30:02 PST 2002

HelloAsync

Information about the request made to the HelloAsync method

```
.EXPLORE = .INVOKE
.EXPLOREOVERRIDE = .TEST
.CONVERSATIONID = 1014147002970
```

Service Response

Submitted at Tue Feb 19 11:30:03 PST 2002

HelloAsync

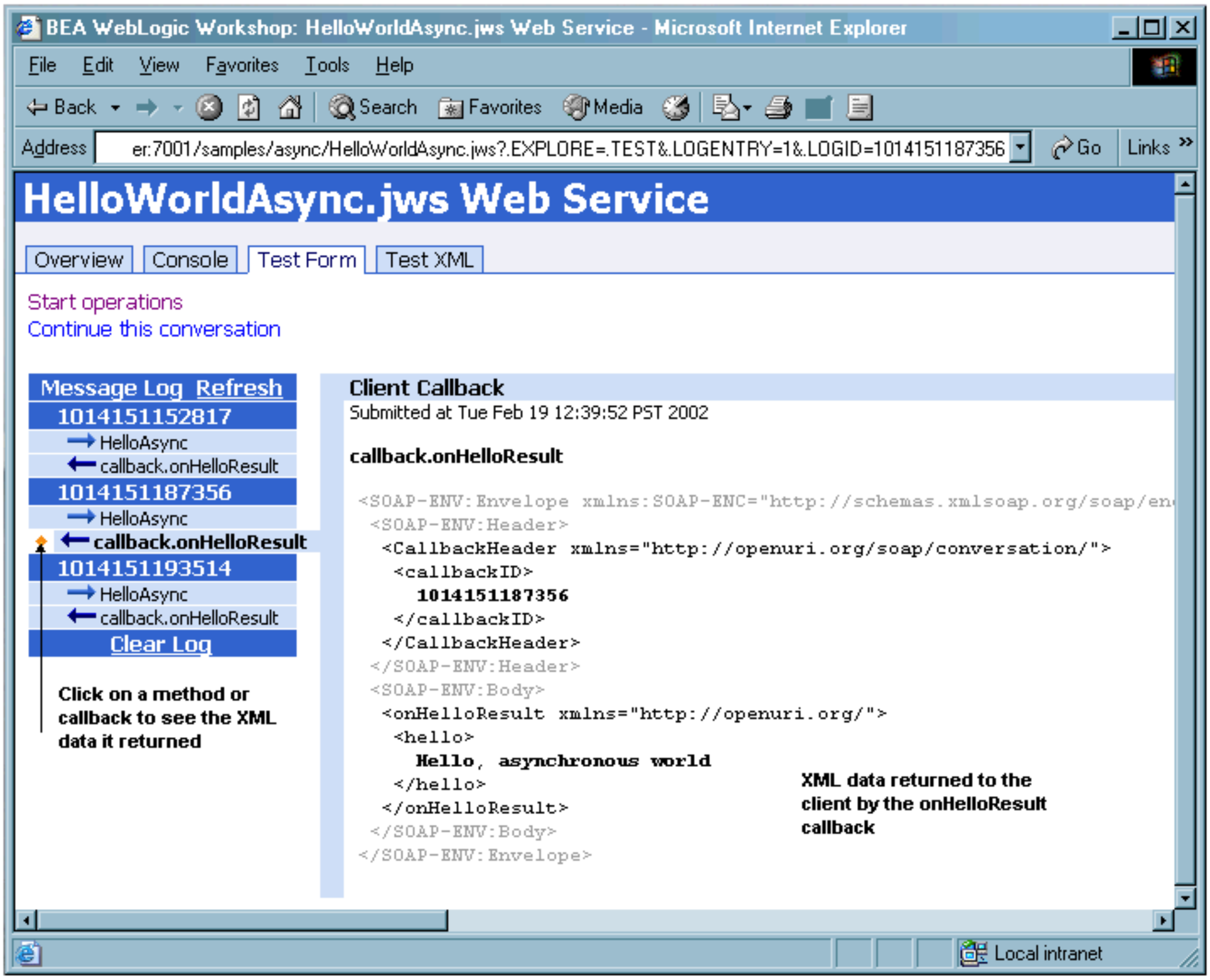
XML data returned by HelloAsync method

```
<HelloAsyncHttpGetOutBodyType xmlns="http://openuri.org/">
</HelloAsyncHttpGetOutBodyType>
```

If your service implements conversations, as the HelloWorldAynch service does, you can use Test View to test the methods that start, continue, or finish a conversation and the callbacks that continue or finish a conversation. You can also test multiple conversations at once.

The conversation ID that appears in the message log uniquely identifies each conversation that is underway. Click on this value to select and work with this conversation. You can view the results for each method or callback that has participated in this conversation by clicking on its name in the list. Click the Refresh link to refresh the message log.

The following image shows Test View with multiple conversations underway:


BEA WebLogic Workshop: HelloWorldAsync.jws Web Service - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address er:7001/samples/async/HelloWorldAsync.jws?.EXPLORE=.TEST&.LOGENTRY=1&.LOGID=1014151187356
Go Links
HelloWorldAsync.jws Web Service
Overview Console Test Form Test XML
Start operations
Continue this conversation
Message Log Refresh
1014151152817
HelloAsync
callback.onHelloResult
1014151187356
HelloAsync
callback.onHelloResult
1014151193514
HelloAsync
callback.onHelloResult
Clear Log
Click on a method or callback to see the XML data it returned
XML data returned to the client by the onHelloResult callback
Local intranet

Client Callback
Submitted at Tue Feb 19 12:39:52 PST 2002

callback.onHelloResult

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/en
 <SOAP-ENV:Header>
  <CallbackHeader xmlns="http://openuri.org/soap/conversation/">
   <callbackID>
     1014151187356
   </callbackID>
  </CallbackHeader>
 </SOAP-ENV:Header>
 <SOAP-ENV:Body>
  <onHelloResult xmlns="http://openuri.org/">
   <hello>
     Hello, asynchronous world
   </hello>
  </onHelloResult>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

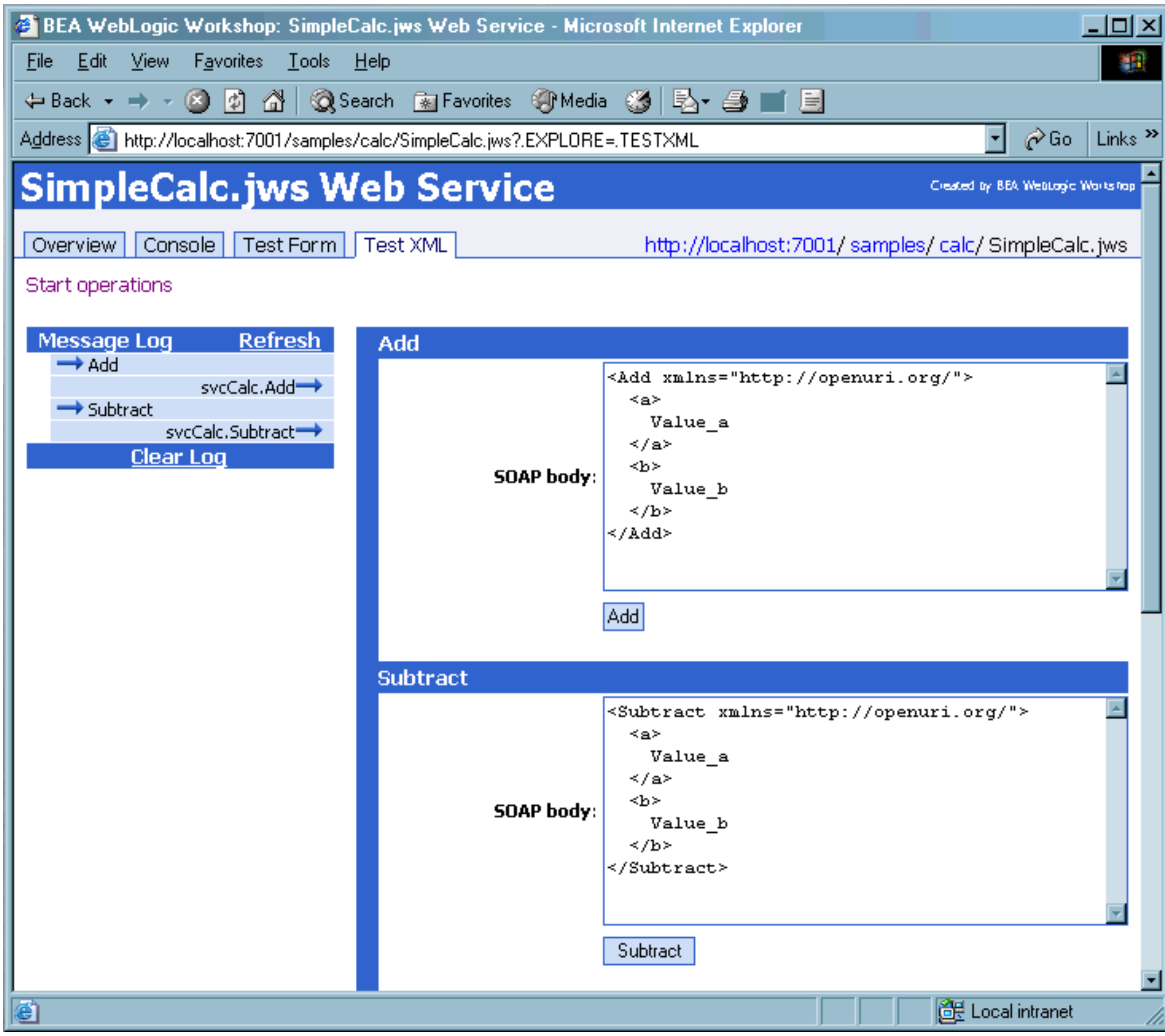
Note: To learn how the comments associated with each method are obtained, and how you can document your web service's methods, see Documenting Web Services.

Test XML Tab

The Test XML tab shows the XML data that is being sent to your service when you test its methods. You can use this page to examine and modify the XML data that is passed to a method of your service.

If your method takes parameters of a data type other than String, you must modify the parameter placeholders in the SOAP body before you click the button to call the method. For example, if your method expects an integer, you must supply a valid integer in the SOAP body.

The following image shows the Test XML tab for a service called SimpleCalc, which performs simple arithmetic calculations:


BEA WebLogic Workshop: SimpleCalc.jws Web Service - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:7001/samples/calc/SimpleCalc.jws?.EXPLORE=.TESTXML
Go Links
SimpleCalc.jws Web Service
Created by BEA WebLogic Workshop
Overview Console Test Form Test XML
http://localhost:7001/ samples/ calc/ SimpleCalc.jws
Start operations
Message Log Refresh
Add
svcCalc.Add
Subtract
svcCalc.Subtract
Clear Log
Add
SOAP body:
Add
Subtract
SOAP body:
Subtract
Local intranet

```
<Add xmlns="http://openuri.org/">
  <a>
    Value_a
  </a>
  <b>
    Value_b
  </b>
</Add>
```

```
<Subtract xmlns="http://openuri.org/">
  <a>
    Value_a
  </a>
  <b>
    Value_b
  </b>
</Subtract>
```

Warnings Tab

The Warnings tab displays warnings that are generated by the compiler when you compile and run your web service. The warnings are not errors, but information about problems your service may encounter as it is currently designed, and suggestions to remedy those problems.

WebLogic Workshop Directory

The WebLogic Workshop Directory contains a sortable list of the files that comprise the different levels of the Project directory. You can use this page to select another service to test, test a control file, and clean up the service.

There are two ways to navigate to the service directory:

- Click on any of the links that make up the service address listed on the upper right−hand corner of the Test View pages. Each part of the path is a link to the directory listing of the files stored in this part of the project directory.
- Click on the See other services in your project link in the upper right−hand corner of the Overview page.

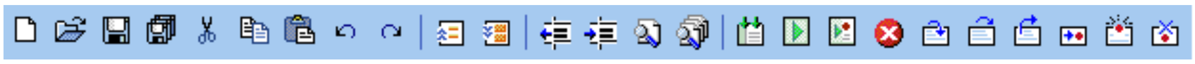
You can use the WebLogic Workshop Directory to:

- See the complete list of files and folders stored at this level.
- Use filters to show files of a particular type such as JWS, Control, XML, ECMASript, and WSDL files.
- Click on a JWS file name and test it.
- Click on a CTRL file, see information about this file, and choose to generate a Test JWS for testing the CTRL.
- Use the Clean All command to delete all generated .java, .class, and directories; erase cached dependency graph information, get rid of all conversation instances, undeploy all beans, and clear logs while leaving your application sources intact.

Related Topics

Web Service Development Cycle

Toolbar



The toolbar provides quick access to the most commonly used development functions of the WebLogic Workshop development environment.

New File

Click the New File button to display the dialog for creating new files.

Open File

Click the Open File button to display a file chooser of files that can be opened

Save

Click the Save button to save the file that is being worked on. You can identify the file that will be saved from the files name on the top right of the Design View / Source View work area.

Save All

Click the Save All button to save all of the files you have open with unsaved changes.

Cut

Click the Cut button to remove the selected text and place it on your clipboard.

Copy

Click the Copy button to and place the selected text on your clipboard.

Paste

Click the Paste button to add the text in your clipboard to your file.

Undo

Click the Undo button to delete the latest change to your web service project.

Redo

Click the Redo button to reinstate any changes which you have removed using the Undo button.

Collapse All Controls

Click the Collapse All Controls button to hide the list of methods displayed underneath their associated controls.

Show All Controls

Click Show All Controls to show the list of methods displayed underneath their associated controls.

Decrease Indent

Click Decrease Indent to move selected text to the right.

Increase Indent

Click Increase Indent to move selected text to the left.

Find

Click Find to search for text and text patterns in the currently opened file.

Find in Files

Click Find in Files to search for text and text patterns in all files of the currently opened project.

Build

Click the Build button to compile the currently opened JWS file into bytecode interpretable by the Java Virtual Machine. Any errors that occur will be displayed in the error pane.

Start

Click the Build button to compile the currently opened JWS file into bytecode interpretable by the Java Virtual Machine. Any errors that occur will be displayed in the Error Pane. If no errors are found a browser window will be launched showing the Test Page where you can test the methods that make up your web service. Note that any breakpoints you have placed in your code will be ignored. To activate any breakpoints in your code, click the Start and Debug button instead of the Start button.

Start and Debug

Click the Build button to compile the currently opened JWS file into bytecode interpretable by the Java Virtual Machine. Any errors that occur will be displayed in the Error Pane. If no errors are found a browser window will be launched showing the Test Page where you can test the methods that make up your web service. Any break points you have placed in your code will be activated. Also debugging information will appear in the main work area providing information about the currently executing web service, such as the values of the local variables and the current position in the call stack.

Stop

Click the Stop button to end the current debugging run of the web service. Clicking Stop ceases execution of your web service and closes the browser window which displays the Test Page.

Step Into / Step Over / Step Out

When a breakpoint is encountered, execution is halted, and you have the option of stepping through the remaining code using the Step Into, Step Over, and Step Out buttons.

Clicking the Step Into button provides the most fine grained display of the executing code. With each click of the Step Into button, the current line of code will be executed, the pointer will move to, but not execute, the next line of code. If the current line of code is a method call, then the point will move to, but not execute, the first line of the called method. To complete execution and return to the original calling method click the Step Out button.

Click the Step Over button to simply execute the current line of code and move to the next line of code within the current method. Any subordinate method call which is encountered on the current line of code is simply executed without displaying the method called.

Click the Step Out button to complete execution of the entire current method and return to the original calling method (if any).

Continue

Click the Continue button to resume execution of code until the next breakpoint is encountered.

Toggle Breakpoint

Click the Toggle Breakpoint button to place a breakpoint on the current selected line of code, if a breakpoint is not already present there. If a breakpoint is present on the currently selected line of code, then clicking the Toggle Breakpoint button will remove the breakpoint.

Clear All Breakpoints

Click the Clear All Breakpoints button to remove all breakpoints from the current file.

Add Database Control Dialog

Use this dialog to add a database control to your web service. A database control provides a Java interface between your web service and a database, allowing you to interact with the database via Java method calls.

For more information about creating Database controls, see Creating a New Database Control.

Step 1

Type the variable name of your database control. When you call database operation methods from your web service, your web service will refer to your database control by this name. It must be a valid Java identifier.

Step 2

If you already have a database CTRL file within your project, choose "Use a Database Control already defined by a CTRL file" and click the Browse button to select the appropriate CTRL file. To finish the dialog, click "Create".

To create a new database CTRL file, choose "Create a new database control to use with this service" and type the name of the new CTRL file. The word "Control" will be automatically appended to the name you enter.

Control Factories

If you make this a control factory, web services can dynamically create multiple instances of this control. To learn more about control factories, see Control Factories: Managing Collections of Controls.

Step 3

If you have chosen to create a new database CTRL file, click the Browse button to select from the data sources available to your database CTRL file. The cgSampleDataSource data source is available for experimentation. To finish the dialog, click "Create".

Related Topics

Controls: Using Resources from a Web Service

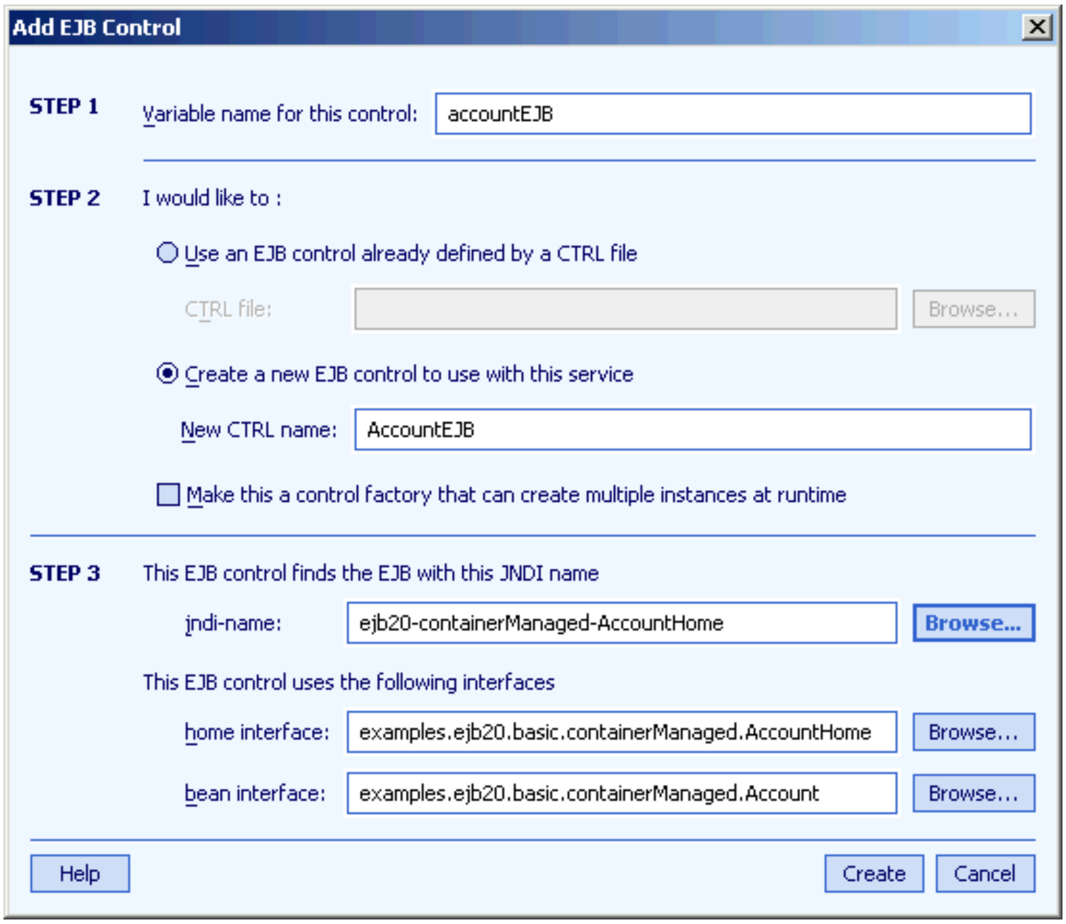
Database Control: Using a Database from Your Web Service

Using a Database Control

Creating a New Database Control

Add EJB Control Dialog

Use this dialog to add a new EJB control to your web service. An EJB control enables a web service to utilize an Enterprise JavaBean.


Add EJB Control

STEP 1 Variable name for this control: accountEJB

STEP 2 I would like to :

( ) Use an EJB control already defined by a CTRL file

CTRL file:

(●) Create a new EJB control to use with this service

New CTRL name: AccountEJB

[ ] Make this a control factory that can create multiple instances at runtime

STEP 3 This EJB control finds the EJB with this JNDI name

jndi-name: ejb20-containerManaged-AccountHome

This EJB control uses the following interfaces

home interface: examples.ejb20.basic.containerManaged.AccountHome

bean interface: examples.ejb20.basic.containerManaged.Account

Step 1: Specify a Name for This Control

Enter a unique name for this EJB control. The name that you choose will be the variable name that you use to refer to this EJB control in your web service's code, so it must be a valid Java identifier.

Step 2: Use an Existing Control, or Create a New Control

To add an existing EJB control to your service, select the option "Use an EJB control already defined by a CTRL file". Type the path and file name, or click the Browse button to locate the CTRL file.

To create a new EJB control, select the option "Create a new EJB control to use with this service," and enter a name for the new control. The word "Control" will be automatically appended to the name you enter. The name you enter plus "Control" must be a valid Java identifier, as it will be used as an interface name.

Control Factories

If you make this a control factory, web services can dynamically create multiple instances of this control. To learn more about control factories, see Control Factories: Managing Collections of Controls.

Control factories are most useful for Timer controls and Service controls.

Step 3: Set Required Attributes for This Control

If you're creating a new EJB control, you must specify the home interface and the bean interface for the EJB that you wish to use. The interfaces must be made available to WebLogic Workshop by placing the target EJB's JAR file or client JAR file in the WEB−INF/lib folder in your WebLogic Workshop project.

You must also specify the JNDI name for the EJB. The Browse button will display a browser listing the EJBs that are registered in the JNDI directory on your development WebLogic Server.

Related Topics

Controls: Using Resources from a Web Service

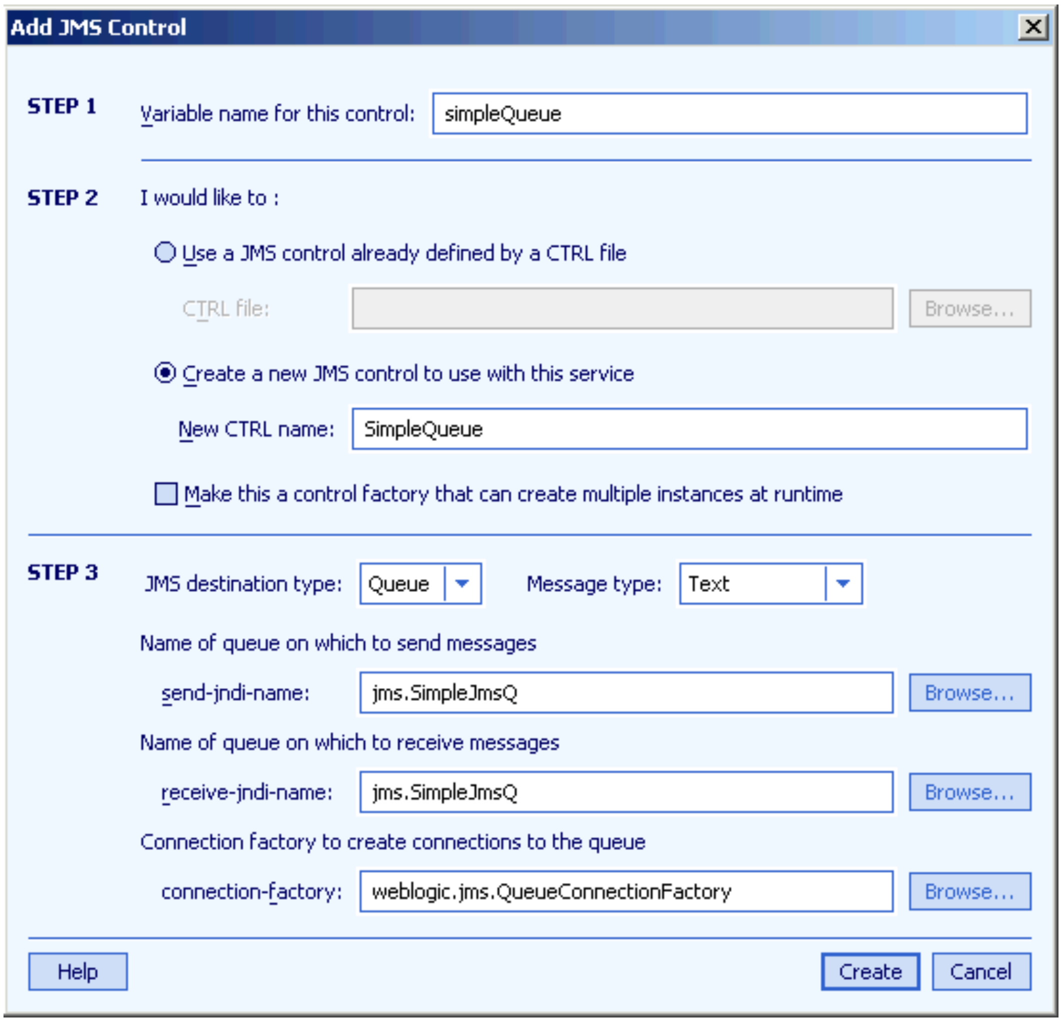
EJB Control: Using an Enterprise Java Bean from Your Web Service

Creating a New EJB Control

Using an EJB Control

Add JMS Control Dialog

Use this dialog to add a new or existing JMS control to your web service. A JMS control enables your service to exchange messages with a Java Messaging Service queue or topic.


Add JMS Control
STEP 1
Variable name for this control:
simpleQueue
STEP 2
I would like to :
( ) Use a JMS control already defined by a CTRL file
CTRL file:
Browse...
(•) Create a new JMS control to use with this service
New CTRL name:
SimpleQueue
[ ] Make this a control factory that can create multiple instances at runtime
STEP 3
JMS destination type:
Queue
Message type:
Text
Name of queue on which to send messages
send-jndi-name:
jms.SimpleJmsQ
Browse...
Name of queue on which to receive messages
receive-jndi-name:
jms.SimpleJmsQ
Browse...
Connection factory to create connections to the queue
connection-factory:
weblogic.jms.QueueConnectionFactory
Browse...
Help
Create
Cancel

Step 1: Specify a Name for This Control

Enter a unique name for this JMS control. The name that you choose will be the variable name that you use to refer to this JMS control in your code.

Step 2: Use an Existing Control, or Create a New Control

To add an existing JMS control to your service, select the option "Use a JMS control already defined by a CTRL file". Type the path and file name, or click the Browse button to locate the CTRL file.

To create a new JMS control, select the option "Create a new JMS control to use with this service," and enter a name for the new control. The word "Control" will be automatically appended to the name you enter.

Control Factories

If you make this a control factory, web services can dynamically create multiple instances of this control. To learn more about control factories, see Control Factories: Managing Collections of Controls.

Step 3: Set Required Attributes for This Control

JMS Destination Type

If you are creating a new JMS control, you must specify whether the control should exchange messages with a JMS queue or topic. The default setting is "queue."

Message Type

A JMS control can process one of four types of messages: Text, XML Map, Object or JMS Message.

Text: messages consist of a single string per message. It is up to the entities communicating via JMS to decipher the meaning of the strings that are passed.

XML Map: messages consist of XML. XML maps are used to translate JMS control method parameters (Java objects) into XML elements in messages. To learn about XML maps, see Handling and Shaping XML Messages with XML Maps.

Object: messages consist of serialized Java objects. It is up to the entities communicating via JMS to use appropriate object types that all communicating entities understand.

JMS Message: messages consist of serialized objects that implement javax.jms.Message.

Destination Name(s)

Next, specify the name of the queue or topic for sending messages in the send−jndi−name attribute, and the name of the queue or topic for receiving messages in the receive−jndi−name attribute. Click the Browse button to select from a list of possible JNDI names defined in WebLogic Server.

Finally, specify the name of the connection factory to create connections to the queue or topic in the connection−factory−jndi−name attribute. Click the Browse button for a list of available connection factory names.

Related Topics

Controls: Using Resources from a Web Service

JMS Control: Using Java Message Service Queues and Topics from Your Web Service

Creating a New JMS Control

Using a JMS Control

Add Service Control Dialog

Use this dialog to add a new Service control to your web service. A Service control provides an interface to another web service, so that your service can invoke the methods of the target service and handle its callbacks.

Step 1: Specify a Name for This Control

Enter a unique name for this Service control. The name that you choose will be the variable name that you use to refer to this Service control in your code.

Step 2: Use an Existing Control, or Create a New Control

To add an existing Service control to your service, select the option "Use a Service control already defined by a CTRL file". Type the path and file name, or click the Browse button to locate the CTRL file.

To create a new Service control, select the option "Create a new Service control from a WSDL," and specify the name of the file containing the WSDL or the URL that points to it. Click the Browse button to search for a local file name. Click the UDDI button to search public or private registries for available web services.

Control Factories

If you make this a control factory, web services can dynamically create multiple instances of this control. To learn more about control factories, see Control Factories: Managing Collections of Controls.

Related Topics

Controls: Using Resources from a Web Service

Service Control: Using Another Web Service

How Do I: Generate a CTRL File?

How Do I: Call One Web Service from Another?

Add Timer Control Dialog

Use this dialog to add a new timer control to your web service.

Step 1: Specify a Name for This Control

Enter a unique name for this timer control. The name that you choose will be the variable name that you use to refer to this timer control in your code.

Step 2: Specify Settings for Attributes of the Timer Control

The timeout attribute specifies the interval until the timer fires for the first time. This value is required, and should include a number followed by a unit designation. For example, a valid setting is "10 s", to specify that the timer fires after ten seconds.

The repeats−every attribute specifies the interval at which the timer fires after the first time. This value is optional, and should also include a number followed by a unit designation.

Control Factories

If you make this a control factory, web services can dynamically create multiple instances of this control. To learn more about control factories, see Control Factories: Managing Collections of Controls.

Related Topics

Controls: Using Resources from a Web Service

Timer Control: Using Timers in Your Web Service

Creating a New Timer Control

Specifying Time on a Timer Control

Autogenerated File Dialog

This dialog appears when you are about to modify an autogenerated control or WSDL file. If you modify and save an autogenerated file, WebLogic Workshop will no longer autogenerate the file when its parent JWS file changes, and the two files will no longer be in synch.

Use caution when you modify an autogenerated file. If you are unsure whether to turn off autogeneration, choose No to maintain the relationship between the two files.

Related Topics

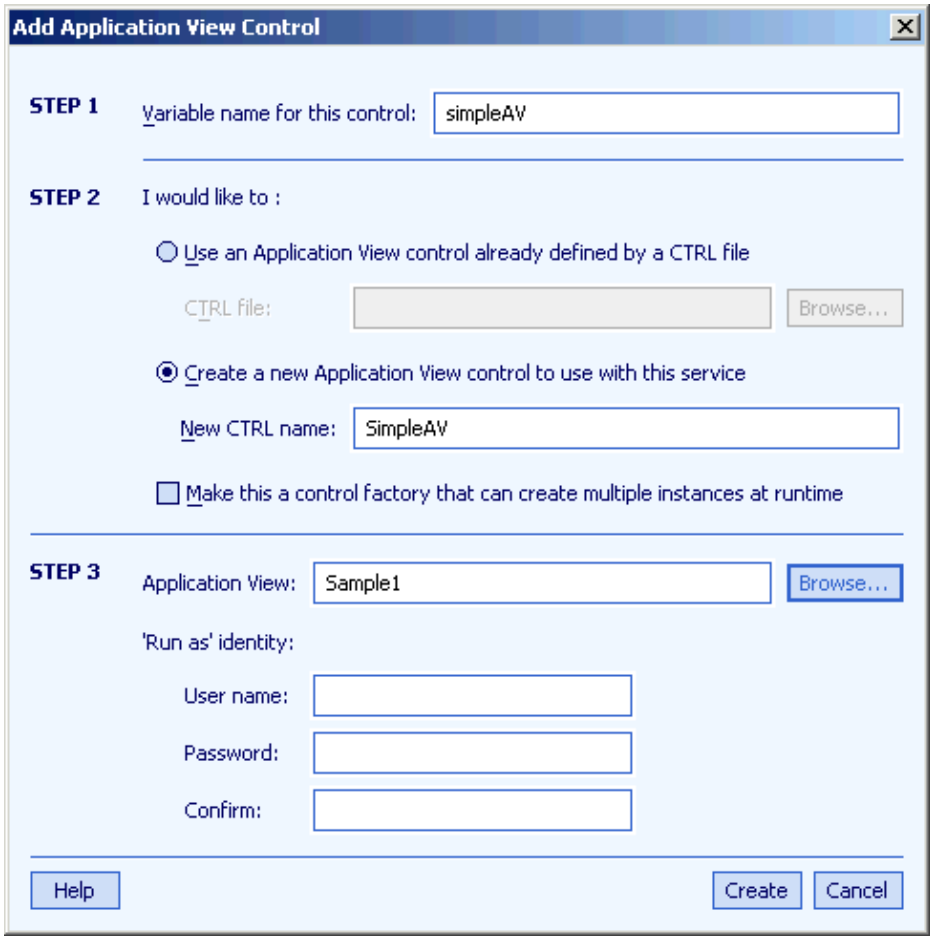
How Do I: Publish a WSDL File for My Web Service?

How Do I: Generate a CTRL File?

Add Application View Control Dialog

Use this dialog to add an Application View control to your web service. An Application View control provides a Java interface between your web service and a WebLogic Integration Application View, allowing you to interact with the Enterprise Information System represented by the Application View via Java method calls.

For more information about creating Application View controls, see Creating a New Application View Control.


Add Application View Control

STEP 1 Variable name for this control: simpleAV

STEP 2 I would like to :

( ) Use an Application View control already defined by a CTRL file

CTRL file:

[✓ Create a new Application View control to use with this service]

New CTRL name: SimpleAV

☐ Make this a control factory that can create multiple instances at runtime

STEP 3 Application View: Sample1

'Run as' identity:

User name:

Password:

Confirm:

Step 1

Type the variable name used to access the new Application View control instance from your web service. When you call Application View control methods from your web service, your web service will refer to the Application View control by this name. The name must be a valid Java identifier.

Step 2

If you already have an Application View control CTRL file within your project, choose Use an Application View control already defined by a CTRL file and click the Browse button to select the appropriate CTRL file. To finish the dialog, click Create.

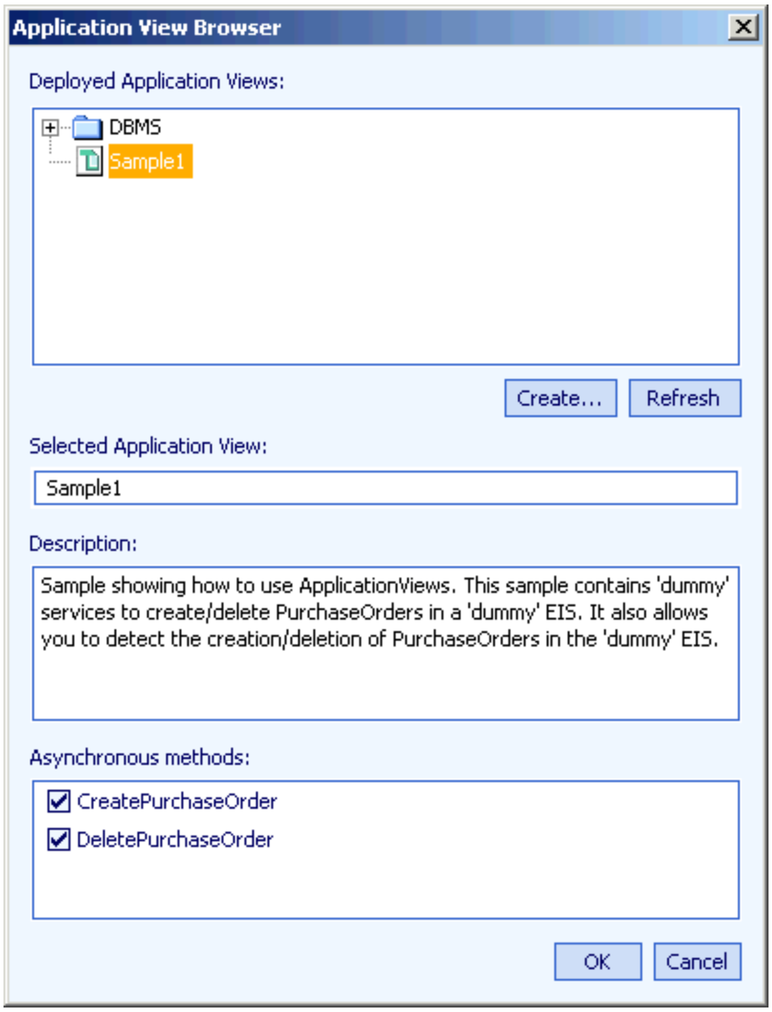
To create a new Application View CTRL file, choose Create a new Application View control to use with this service and type the name of the new CTRL file. The word "Control" is automatically appended to the name you enter.

Control Factories

If you make this a control factory, web services can dynamically create multiple instances of this control. To learn more about control factories, see Control Factories: Managing Collections of Controls.

Step 3

If you have chosen to create a new Application View CTRL file, click the Browse button to select from the deployed Application Views. To finish the dialog, click Create.



Sample showing how to use ApplicationViews. This sample contains 'dummy' services to create/delete PurchaseOrders in a 'dummy' EIS. It also allows you to detect the creation/deletion of PurchaseOrders in the 'dummy' EIS.

- [✓] CreatePurchaseOrder
- [✓] DeletePurchaseOrder

If you select the Browse button in Step 3, the Application View Browser is displayed. It presents the Application Views that are deployed in the current domain. When an Application View is selected, the Application Views description is presented.

If the Application View defines asynchronous services, each one is displayed in the Asynchronous methods area of the dialog. You may choose which methods to use asynchronously. By default, all asynchronous services are selected to be implemented as asynchronous methods.

In Step 3, you may also set the username and password with which the Application View control will access the target Application View.

Related Topics

Controls: Using Resources from a Web Service

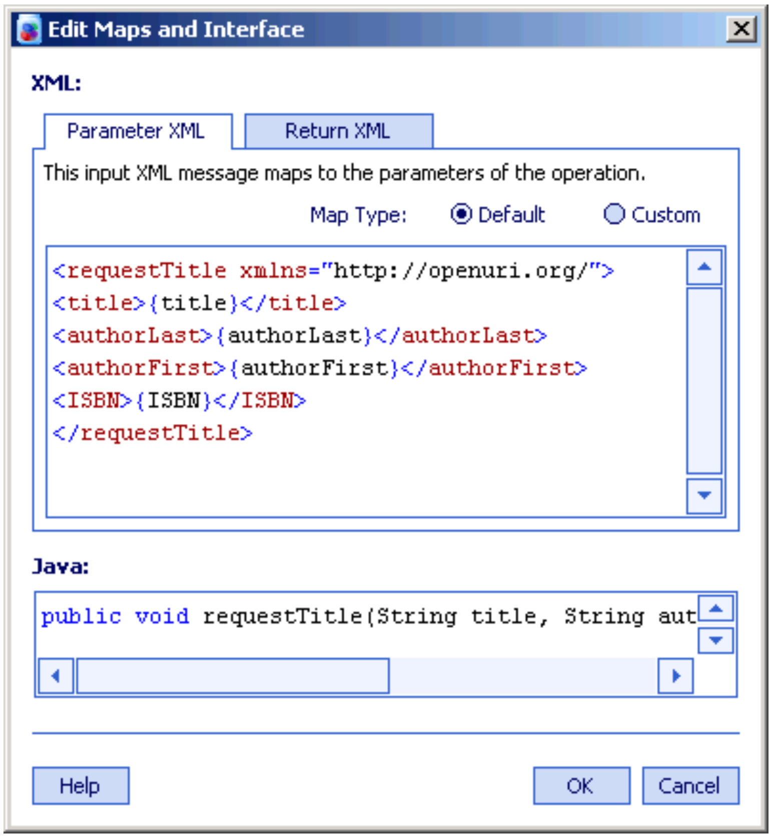
Application View Control: Accessing an Enterprise Application from a Web Service

Using an Application View Control

Creating a New Application View Control

Edit XML Maps and Interface Dialog (General)

Use this dialog to view and edit the declaration and XML maps for a method or callback. Through XML maps, you can define the relationship between values in an incoming or outgoing XML message and method or callback parameters and return values.



This input XML message maps to the parameters of the operation.

Map Type: [✓ Default] Custom

```
<requestTitle xmlns="http://openuri.org/">
<title>{title}</title>
<authorLast>{authorLast}</authorLast>
<authorFirst>{authorFirst}</authorFirst>
<ISBN>{ISBN}</ISBN>
</requestTitle>
```

```
public void requestTitle(String title, String aut
```

XML

Provides a place to specify which map you are editingadov−−>the parameter−xml map or the return−xml map. By default, the XML pane displays a default map representing the natural mapping provided by WebLogic Server. Click in the pane or click Custom to edit the default map, creating a custom XML map.

Note: When you click in the XML pane or click Custom, WebLogic Workshop adds code for the default map to your source code immediately preceding the declaration for this method or callback. This code will reflect edits you make in the Edit Maps and Interface dialog.

Parameter XML

Provides a place to edit the parameter−xml map, which governs mapping between values in method and callback parameters and XML messages.

Return XML

Provides a place to edit the return−xml map, which governs mapping between values in method and callback return values and XML messages.

The return value of a method or callback is referred to using the special name return in the map. If the return value is an Object, you may access fields and methods of the object using standard dot notation. For example, return.field.

Note: The direction of parameter−xml and return−xml maps changes depending on the context.

In interactions between your web service and its clients (the left hand side of Design View) a method arriving at your web service has incoming parameters and an outgoing return value and callbacks have outgoing parameters and an incoming return value.

In interactions between your web service and controls (the right hand side of Design View), methods have outgoing parameters and an incoming return value and callbacks have incoming parameters and an outgoing return value.

Map Type

Choose an option to specify whether to use the natural (default) mapping provided by WebLogic Server or a custom mapping:

- Default adov−−> Choose this to allow WebLogic Server to use the natural mapping to describe the relationship between XML messages and your declaration's parameters and return values.
- Custom adov−−> Choose this to edit the XML map, defining a custom mapping between between XML messages and your declaration's parameters and return values.

Java

Provides a place to edit the declaration for the selected method or callback. Edits to the declaration here will be reflected in your source code.

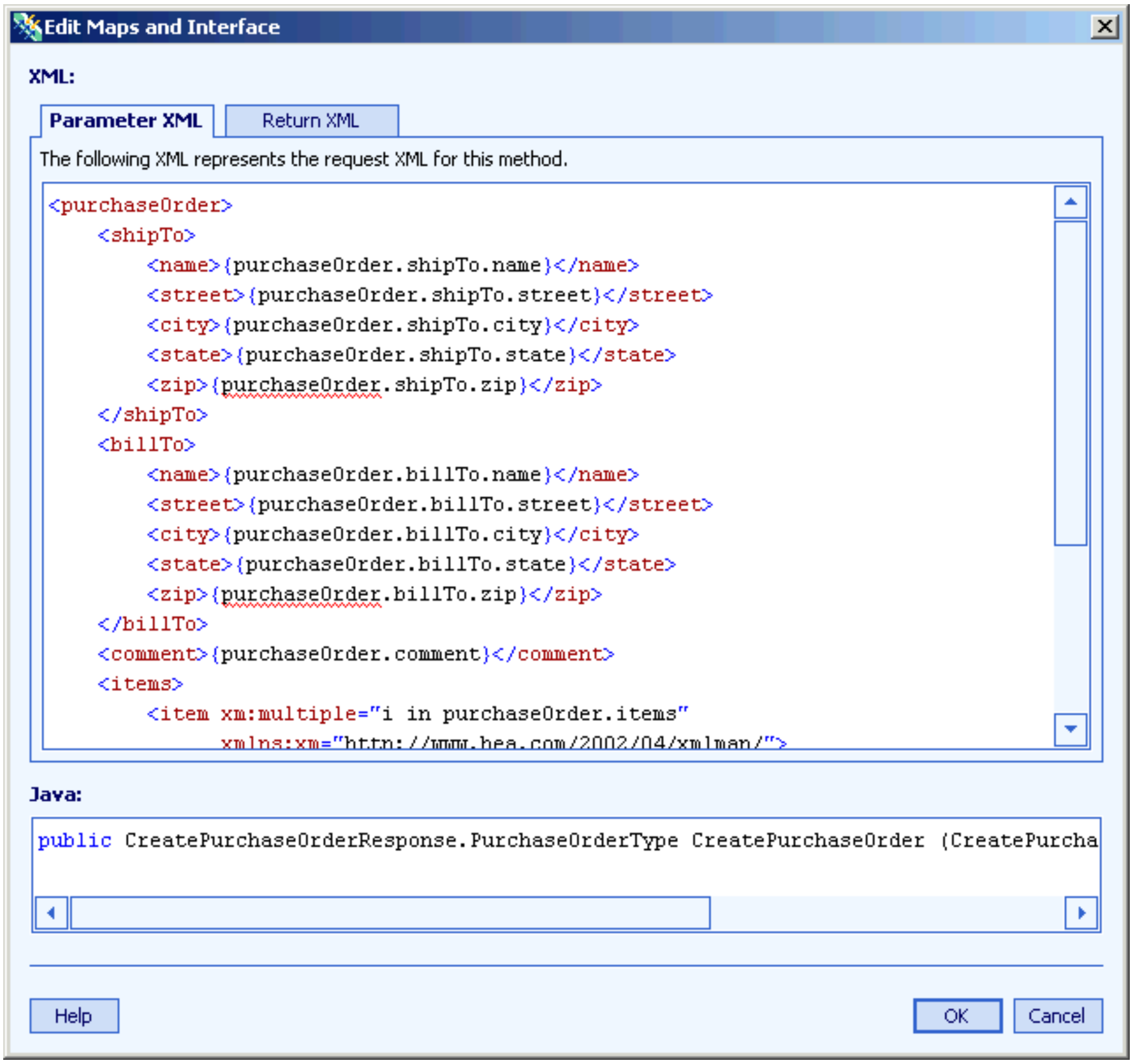
Related Topics

Why Use XML Maps?

How Do I: Add or Edit an XML Map with the Edit Maps and Interface Dialog?

Edit XML Maps and Interface Dialog (AppView Control)

Use this dialog to view and edit the declaration and XML maps for a method or callback of an AppView control. Through XML maps, you can define the relationship between values in an incoming or outgoing XML message and method or callback parameters and return values.



The following XML represents the request XML for this method.

```
<purchaseOrder>
    <shipTo>
        <name>{purchaseOrder.shipTo.name}</name>
        <street>{purchaseOrder.shipTo.street}</street>
        <city>{purchaseOrder.shipTo.city}</city>
        <state>{purchaseOrder.shipTo.state}</state>
        <zip>{purchaseOrder.shipTo.zip}</zip>
    </shipTo>
    <billTo>
        <name>{purchaseOrder.billTo.name}</name>
        <street>{purchaseOrder.billTo.street}</street>
        <city>{purchaseOrder.billTo.city}</city>
        <state>{purchaseOrder.billTo.state}</state>
        <zip>{purchaseOrder.billTo.zip}</zip>
    </billTo>
    <comment>{purchaseOrder.comment}</comment>
    <items>
        <item xm:multiple="i in purchaseOrder.items"
            xmlns:xm="http://www.bea.com/2002/04/xmlmap/">
```

```
public CreatePurchaseOrderResponse.PurchaseOrderType CreatePurchaseOrder (CreatePurcha
```

XML

Provides a place to specify which map you are editingadov−−>the parameter−xml map or the return−xml map.

Parameter XML

Provides a place to edit the parameter−xml map, which governs mapping between values in method and callback parameters and XML messages.

Return XML

Provides a place to edit the return−xml map, which governs mapping between values in method and callback return values and XML messages.

The return value of a method or callback is referred to using the special name return in the map. If the return value is an Object, you may access fields and methods of the object using standard dot notation. For example, return.field.

Note: the direction of parameter−xml and return−xml maps changes depending on the context.

In interactions between your web service and its clients (the left hand side of Design View) a method arriving at your web service has incoming parameters and an outgoing return value and callbacks have outgoing parameters and an incoming return value.

In interactions between your web service and controls (the right hand side of Design View), methods have outgoing parameters and an incoming return value and callbacks have incoming parameters and an outgoing return value.

Java

Provides a place to edit the declaration for the selected method or callback. Edits to the declaration here will be reflected in your source code.

Related Topics

Why Use XML Maps?

How Do I: Add or Edit an XML Map with the Edit Maps and Interface Dialog?

AppView Control: Accessing an Enterprise Application from a Web Service

JMS Map Editor Dialog

Use this dialog to view and edit the function definition, header, property, and message for a method of a JMS control.

XML

In the XML section, you can specify an XML map for the JMS message, header, or property, on the corresponding tab.

Message XML

Provides a place to edit the jms−message XML map. The value of this is an XML document fragment describing an XML document and parameters.

- For XML map applied to a CTRL file method, this tells the control to publish an XML document using the XML map as a template.
- For a CTRL file callback, tells the control to use the XML map to set the callback parameters.

Header XML

Provides a place to edit the jms−header XML map. This maps JMS headers to publish/callback parameters. The value of this is an XML document fragment with the element name as the value and the element text as the value, in the format:

```
<header>
    <header-name>value</header-name>
    <header-name-2>value2</header-name2>
</header>
```

- For a CTRL file method, tells the control to set headers using the XML map as a template.
- For a CTRL callback, tells the control to use the XML map to set the callback parameters.

Property XML

Provides a place to edit the jms−property xml−map. This maps jms user properties to publish/callback parameters. The value of this is an XML document fragment with the element name as the value and the element text as the value, in the format:

```
<property>
    <prop-name>value</prop-name>
    <prop-name-2>value2</prop-name-2>
</property>
```

- For a CTRL file method, tells the control to set properties using the XML map as a template.
- For a CTRL file callback, tells the control to use the XML map to set the callback parameters.

Java

The Java section provides a place to edit the definition for the selected method of the JMS control. Your edits to the method definition here will be reflected in your source code.

Related Topics

Why Use XML Maps?

JMS Control: Using Java Message Service Queues and Topics from your Web Service

New File Dialog

Use this dialog to create a new file within your web service project.

1. Choose Web Service, Java, JavaScript, or Text.

- Choosing Web Service creates a new JWS file (a file with the .jws file extension).
- Choosing Java creates a new JAVA file (a file with the .java file extension).
- Choosing JavaScript creates a new JSX file (a file with the .jsx file extension).
- Choosing Text creates a new TXT file by default; however, you can change the TXT file extension by typing in the text box labeled File extension.

2. In the File name field, type the name of file and click OK to finish the dialogue.

New Project Dialog

Use this dialog to create a new web services project. A project contains various files supporting related web services, such as JWS, JSX, CTRL, and WSDL files.

To create a new project, type the name of the new project and click OK.

The name you choose for the project will become part of the URL used to access web services in the project. For example, a web service named "myWebService" residing in a project named "myProject" is accessed by the URL:

```
http://host:port/myProject/myWebService.jws
```

Related Topics

WebLogic Workshop Projects

New Variable Dialog

Use this dialog to add a new variable to the Java class which constitutes your web service.

Type the name of the variable and choose its data type from the drop−down list provided, then click OK.

Related Topics

Structure of a JWS File

Preferences Dialog

Use the preferences dialog to modify your preferences for the WebLogic Workshop visual development environment. Many of the values that are set in the Preferences Dialog are stored in the Workshop.properties Configuration File.

Display Tab

The Display tab allows you to modify display settings for WebLogic Workshop, such as font face and size and sorting paradigms for the visual development environment.

Editor Tab

The Editor tab allows you to modify settings for the code editor, including spacing, indentation, and code completion options.

Colors Tab

The Colors tab shows the text and background colors for various elements in the visual development environment.

Paths Tab

The Paths tab provides information about the location of the WebLogic Server installation and domain against which you are developing your web service. By default, WebLogic Workshop is configured for development against the WebLogic development server that is integrated with WebLogic Workshop.

If you want to develop your web service against a remote WebLogic Server installation, you can modify the WebLogic development server options, as described below:

- Name: Change the Name option to the name of the remote WebLogic Server. By default Name is set to localhost, which refers to the integrated development server.
- Port: Change the Port option to the port specified for the remote WebLogic Server. By default Port is set to 7001.
- Config directory: The Config directory option refers to the directory containing domains for WebLogic Server. Change the path specified in this directory if you are developing against a remote WebLogic Server installation, so that it points to the domain−containing directory for the remote server. The directory on the remote server must be mounted or mapped to a local directory or drive letter. By default the Config directory option is set to the path where the integrated development server is installed, for example, C:\bea\weblogic700\samples.
- Domain: Change the Domain option if you have configured a domain other than the default for WebLogic Workshop develoment, or if you are developing against a remote server. By default the Domain option is set to Workshop, which is the default domain for WebLogic Workshop development.
- Note that the domains listed in the Domain option correspond to the directories beneath the directory specified in the Config directory option.
- When you choose File−−>Open Project, the list of available projects is determined by the combined settings of the Config directory and Domain options. This list corresponds to the folders beneath the applications directory in the specified domain.

Related Topics

Workshop.properties Configuration File

SQL Editor Dialog

Use this dialog to associate SQL statements with Java methods within a database CTRL file.

When you use this dialog, the changes you make are saved as new or modified methods with @jws:sql tags in the Database control's CTRL file.

SQL

Type a SQL statement to be executed when the Java method (in the window below) is called. Use curly braces to substitute Java method parameters into your SQL statement. See the example and related topics below for syntax details.

Java

Type a Java method declaration, including the its initial modifier (whether the method is public, private, etc.) and its return type. When you call this method from your web service, the SQL statement (in the window above) will be executed.

The possible return types for the Java method depend on what is returned from the database operation.

Most "administrative" operations such as INSERT and UPDATE return an integer indicating the number of rows affected.

SELECT queries may return a single value, a single row (or partial row) or multiple rows (or partial rows). To learn what Jave types are valid when for these database operation results, see the following topics:

- Returning a Single Value from a Database Control Method
- Returning a Single Row from a Database Control Method
- Returning Multiple Rows from a Database Control Method

Example #1

SQL

SELECT * FROM CUSTOMER

JAVA

public java.sql.ResultSet findAllCustomersResultSet()

Example #2

SQL

SELECT CITY FROM CUSTOMER WHERE CUSTID = {key}

JAVA

public String findCustomerCityByID(int key)

Related Topics

Database Control: Using a Database from Your Web Service

Creating a New Database Control

Parameter Substitution in @jws:sql Statements

Mapping Database Field Types to Java Types in the Database Control

@jws:sql Tag

Support

For BEA support resources, see http://support.bea.com/, where you'll find:

- BEA's online support center
- AskBEA, where you can search for answers to your questions about WebLogic Workshop and other BEA products
- BEA NewsWeb, where you can browse or post questions to the WebLogic Workshop newsgroup, weblogic.developer.interest.workshop
- Contact information for phone support

dev2dev

BEA's dev2dev web site offers technical resources for the BEA developer community, including articles written by expert developers, up−to−date product information, white papers, ongoing discussions, support, product downloads, and more.

e−docs

Up−to−date documentation for WebLogic Workshop and other BEA products is available on e−docs.bea.com.

Glossary

application server

An application server is designed to help make it easier for developers to isolate the business logic in their projects (usually through components) and develop three−tier applications. Many application servers also offer additional features such as transaction management, clustering and fail−over, and load balancing.

In the context of WebLogic Workshop, the term application server refers specifically to a Java Application Server, which is an application server that complies with the Java 2 Enterprise Edition (J2EE) platform.

BEA WebLogic Server™, on which WebLogic Workshop is built, is a Java Application Server.

asynchronous

In distributed application architectures like web services, clients invoke methods (or send messages to) servers and servers respond. If the client is blocked from performing other work while waiting for the server to respond, the interaction is referred to as synchronous because the client is synchronized with the server.

If the interaction is designed such that the client can continue performing other work while the server prepares its response, with the server notifying the client when the response is ready, it is referred to as asynchronous.

An asynchronous architecture is useful in event−driven scenarios, in which an event can arrive at any time and the receiver handles the event whenever it arrives.

buffering

You can add a buffer to a method of your service to ensure that it returns to the client immediately, so that the client need not wait for the server to process other requests. Incoming calls to a buffered method are queued so that the server is not overwhelmed with requests.

callback

A callback is a method defined on the client which can be called by your service. Callbacks make it possible to have an asynchronous two−way exchange between a client and a service. For example, if the service performs an operation that takes awhile, the service can immediately acknowledge the client's request with a simple return value, then later use the callback to return the full result of the operation. A callback must participate in a conversation.

callback handler

A callback handler is a method of your service which runs when the service receives the corresponding callback. The method is defined by the control that includes the callback. For example, the timer control provides a callback handler, the onTimeout method, to which you can add code to run when the timer fires.

client (of a web service)

A client makes a request to a web service to return data. The client can be written in any language and running on any platform, so long as it communicates in the manner that the web service expects. Most web services expect to receive requests over an Internet protocol such as HTTP, and they expect those requests to be XML messages formatted according to the SOAP specification.

control

A control is a component that you can incorporate into your web service so that it can communicate with other kinds of applications and components. For example, a database control enables your web service to request data from a database. A Service control makes it easy to call another web service.

control method

A method that allows the web service to invoke the functionality of the data resource.

conversation

A conversation is a sequence of interactions that may occur between a client and a web service. A single instance of a web service conversation has state associated with it that persists for the lifetime of the conversation. At a minimum, this state includes a “conversation ID” which is used to correlate messages coming in and out of the service with a specific conversation instance. This means that if you are processing 1000 credit checks for 1000 people, WebLogic Workshop automatically keeps a unique conversation open for each, and it correlates messages coming in and out with the appropriate conversation.

correlation

Correlation refers to a capability WebLogic Workshop automatically provides to web services. If a web service has many simultaneous clients, it must keep track of which responses go to which clients in response to requests. This includes callbacks: when events occur in the web service that must be passed on to clients via callbacks, WebLogic Workshop's correlation capability routes the callback messages automatically.

CTRL file

A file that contains one or a set of control methods.

deployment

Deployment refers to the act of moving a web service from a development environment to a production environment. Since WebLogic Workshop's runtime is integrated with the WebLogic Server, deployment is very simple for WebLogic Workshop web services: just copy the JWS file (and any supporting files) to the production server and the web service is deployed.

ECMAScript

ECMA is the European Computer Manufacturers Association, the organization that acted as the standards body for ECMAScript.

ECMAScript is the standardized, combined form of the JavaScript and JScript languages. JavaScript was developed by Netscape Communications and JScript was developed by Microsoft Corporation.

lifetime, of web service

A particular instance of a WebLogic Workshop web service exists in the WebLogic Server for a length of time that is controlled by the web service developer. If a web service does not manage conversations, the web service may be created and destroyed each time it is invoked. If the web service manages conversations, each conversation has a lifetime. If no interactions happen in a conversation within a specified timeout interval, the conversation instance will be released. You can also specify, implicitly or explicitly, that a conversation should end when one or more particular methods or callbacks are invoked.

operation

A method that is exposed to a client.

Service control

A Service control makes it easy to communicate with another web service from within your web service. You can create a Service control from any target service's WSDL.

shape (of XML)

XML shape refers to the structure and organization of an XML document, including its hierarchy and order of elements.

SOAP

SOAP, or Simple Object Access Protocol, is a standard set of rules for formatting an XML message so that it can be interpreted by different web services.

SQL

SQL is the acronym for Structured Query Language. See www.sql.org. SQL is a semi−standard language used to communicate with relational databases. SQL statements or queries are used to create, manipulate, query, update and delete tables and records.

synchronous

In simple terms, a method is synchronous if it returns a value. The term comes from the fact that a caller to a synchronous method must wait for the called method to return. The caller is synchronized with the method.

See also asynchrony.

UDDI

Acronym for Universal Description, Discovery and Integration. See www.uddi.org.

web service

An Web Service is a language−independent, platform−independent, self−describing code module that applications can access via a network or the Internet. The application can have the service's location hard−coded or can locate it using UDDI (Universal Description, Discovery, and Integration). Because the service is self−describing, the application can determine which functions are available and how to call them.

well−formed

A quality of an XML document that conforms to XML syntax rules. For a document in use, syntax rules are typically enforced by an XML parser.

WSDL

WSDL is the acronym for Web Services Description Language, the markup language used to describe a web service so that it can be called by diverse clients.

XML map

An XML map correlates the data in an XML message to the parameters and return values of a Java method in your web service. XML maps offer you greater freedom in terms of how you implement the code in your web service, because you are not tied to a particualar XML shape.

Search Tips

This topic includes tips for searching WebLogic Workshop help.

In addition to searching on simple expressions, you can perform more complex searches using Boolean operators and nested expressions, as described in the following sections.

Boolean Operators

You can use the AND, OR, NOT, and NEAR operators to narrow your search by creating a relationship between search terms. The following table shows how to use each of these operators. If no operator is specified, the AND operator is used. For example, the query "asynchronous web service" is equivalent to "asynchronous AND web AND service." If you need to search for a term that contains more than one word, using the NEAR operator rather than the implicit AND operator may provide the most specific results.

Search For	Example	Results
Both terms in the same topic.	asynchronous AND callback	Topics containing both the words "asynchronous" and "callback"
Either term in a topic.	callback OR polling	Topics containing either the word "callback" or the word "polling" or both.
The first term without the second term.	EJB NOT control	Topics containing the word "EJB" but not the word "control"
Both terms in the same topic, close together.	message NEAR buffer	Topics containing the word "message" within eight words of the word "buffer"

Note: The |, &, and ! characters don't work as Boolean operators (you must use OR, AND, and NOT).

Nested Expressions

Nested expressions allow you to create complex searches for information. For example, "asynchronous AND (message NEAR buffer)" finds topics containing the word "asynchronous" as well as the words "message" and "buffer" close together.

The basic rules for searching Help topics using nested expressions are as follows:

- You can use parentheses to nest expressions within a query. The expressions in parentheses are evaluated before the rest of the query.
- Terms within a nested expression are evaluated first. If a query does not contain a nested expression, it is evaluated from left to right.
- You cannot nest expressions more than five levels deep.