BEA WebLogic Workshop™ Help

Version 8.1 SP2
November 2003

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Table of Contents

Table of Contents

Table of Contents

Building Web Services

WebLogic Workshop makes it easy to build enterprise−class web services, even if you're not a J2EE expert.

Topics Included in This Section

Getting Started with Web Services

Describes the technologies underlying web services, discusses the advantages of building web services with WebLogic Workshop, and provides a first introduction to designing and building web services.

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.

Transforming XML Messages With XQuery

Use XQuery expressions to transform one XML shape to another or map XML to Java.

Handling XML with ECMAScript Extensions

Learn how to manipulate XML with the ECMAScript extensions.

Handling and Shaping XML Messages With XML Maps (Deprecated)

XML maps are deprecated in WebLogic Workshop 8.1. You should use XQuery expressions to transform XML instead.

Related Topics

Designing Asynchronous Interfaces

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

Introduction to XQuery Maps

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

Security

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

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.

How Do I...?

Provides step−by−step information for approaching common tasks.

Samples

Explains how to use the web services samples that accompany WebLogic Workshop.

Reference

Provides reference information for building web services.

Getting Started with 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 provide, 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.

WebLogic Workshop makes it easy for you to build and deploy applications, including web services. The following topics offer an overview of web services and the standard technologies they rely on, and help you figure out where to start when building a web service.

Topics Included in This Section

Introduction to Web Service Technologies

Explores the standard technologies underlying web services.

Why Build Web Services With WebLogic Workshop?

Discusses the technical advantages of building web services with WebLogic Workshop.

Building Web Services with WebLogic Workshop

Describes the basic components of a web service built with WebLogic Workshop.

Web Service Development Cycle

Provides an overview of designing, implementing, testing, debugging, and deploying a web service.

XML Strategies and Web Service Design

Describes various strategies for specifying or interpreting XML messages received by web services.

Accessing SOAP Attachments in Web Services

Describes how to receive and return MIME messages as SOAP attachments in web service methods.

Specifying SOAP Handlers for a Web Service

Describes how to define and configure SOAP message handlers for a web service.

Documenting Web Services

Explains how to document various aspects of a web service, such as annotations, formatting, and methods.

WSDL Files: Web Service Descriptions

Discusses how WSDL files are used to describe web service interfaces.

Related Topics

Designing Asynchronous Interfaces

Design robust, asynchronous web services to coordinate long−running operations

Introduction to Web Service Technologies

A web service makes software application resources available over networks using standard technologies. Because web services are based on standard interfaces, they can communicate even if they are running on different operating systems and are written in different languages. For this reason they are an excellent approach for building distributed applications that must incorporate diverse systems over a network.

The following topic outlines the standard technologies that you use to build web services.

Standard Technologies

Web services are able to expose their resources in this generally accessible way because they adhere to recognized standards. A web service:

- Publicly describes its own functionality through a WSDL file
- Communicates with other applications via XML messages, often formatted with SOAP
- Employs a standard network protocol such as HTTP

WSDL Files

The Web Service Description Language (WSDL) is a standard XML format for describing web services. A WSDL file describes a particular web service so that other software applications can interface with it.

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 a common language by which different applications can talk to one another over a network. Most web services communicate via XML. A client sends an XML message containing a request to the web service, and the web service responds with an XML message containing the results of the operation. In most cases 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 web services may communicate with XML messages that are not SOAP−formatted. The types of messages supported by a particular web service are delineated in the service’s WSDL file.

For more information about XML see Introduction to XML.

Network Protocols

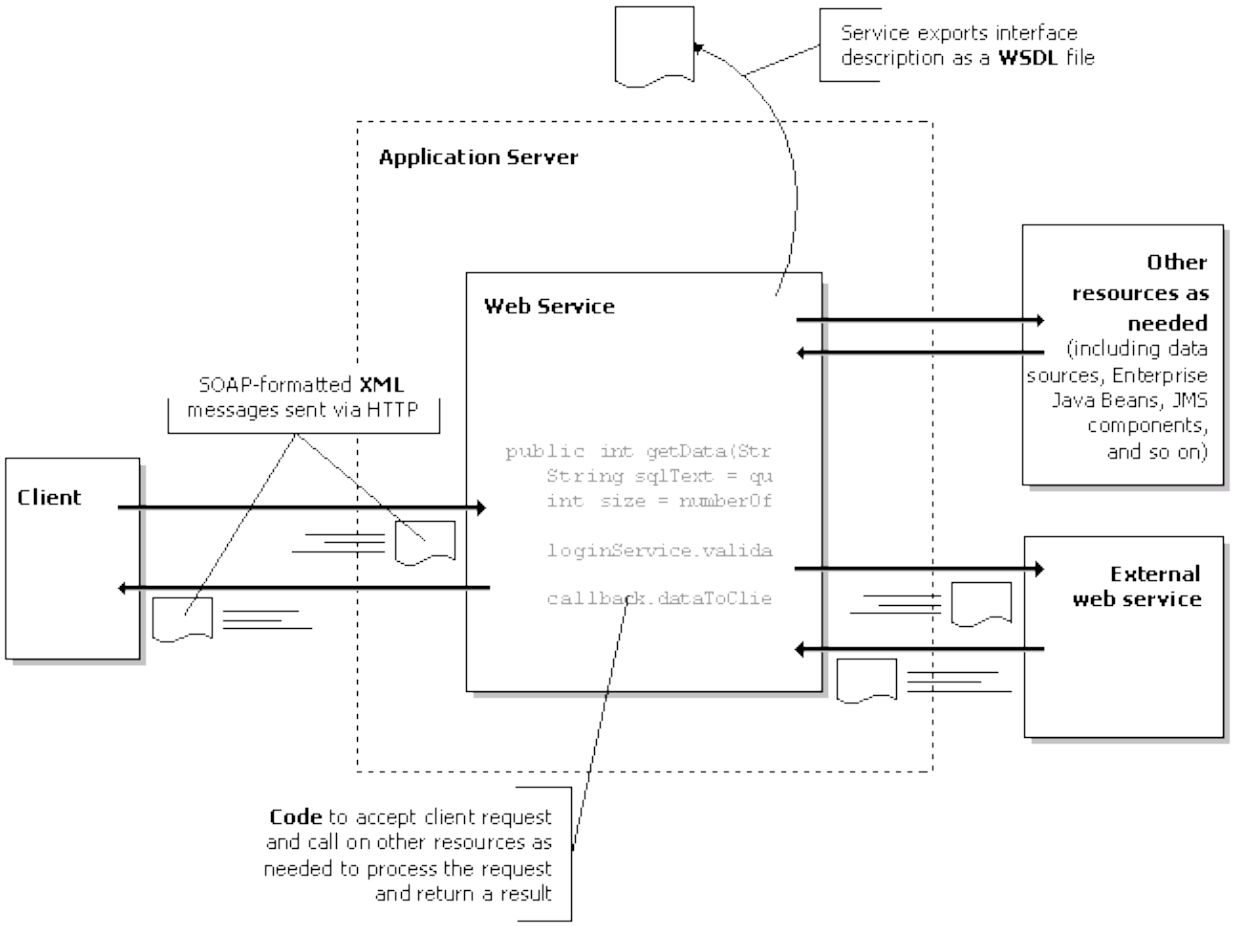
Web services receive requests and send responses using widely used protocols such as HyperText Transfer Protocol (HTTP) and Java Message Service (JMS). A web service may support more than one protocol, and different methods on the web service may support different protocols. The protocols that a web service

supports are published in the WSDL file.

For more information on network protocols, see Protocols and Message Formats.

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.



Related Topics

Building Web Services with WebLogic Workshop

Why Build Web Services with WebLogic Workshop?

Web services promise a new level of integration for the enterprise, without prohibitive cost. Why should you build web services with WebLogic Workshop?

WebLogic Workshop provides a framework for building web services that automatically leverage the power, reliability and scalability of WebLogic Server. The web services that you build with WebLogic Workshop are enterprise−class services, and WebLogic Workshop provides simple controls for connecting to your enterprise resources. At the same time, WebLogic Workshop simplifies the process of creating web services by insulating developers from the low−level implementation details that have traditionally made web service development the domain of sophisticated J2EE developers. With WebLogic Workshop, you can build powerful web services whether you're an application developer or a J2EE expert.

Web services that you build with WebLogic Workshop are asynchronous, loosely−coupled, and reliable. The following sections describe these advantages in greater detail.

Asynchronous

Many business processes take more than a few moments to complete, but traditional architectures make it hard to handle long−running tasks efficiently. WebLogic Workshop helps you architect asynchronous web services easily using conversations and callbacks. Conversations help manage the typical problems in asynchronous messaging, namely correlating messages and managing some information or state between message exchanges. In an ongoing conversation, a web service can notify a client when the results of an operation are ready using a callback.

In addition, WebLogic Workshop supports the use of Java Message Service (JMS) queues as message buffers to ensure that web service messages are not lost regardless of server load. JMS can also be used by WebLogic 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 implementation details of the web service. A web service publishes only an interface, or *public contract*, that determines how users can interact with the web service. As long as the messages and data formats 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 have all the reliability, scalability, and availability you've come to expect from J2EE applications running on WebLogic Server—absolute requirements for web services deployed in the enterprise.

Related Topics

Designing Asynchronous Interfaces

Building Web Services with WebLogic Workshop

You can build enterprise−class web services with WebLogic Workshop. Web services built with WebLogic Workshop employ standard web service technologies: XML, SOAP, and WSDL. WebLogic Workshop simplifies web service development by allowing you to focus on application logic, rather than the complex implementation details traditionally required by these technologies.

The following sections explain the basic concepts that you need to know about to begin building web services with WebLogic Workshop, and point you to more in−depth information about each.

Applications and Projects

You build web services within a Web Service project. A Web Service project corresponds to a J2EE web application. You may build multiple web services within a single project.

Your Web Service project, and other projects you're working on, are contained in an *application*. An application is a set of projects, libraries, and resources that are related in some way. At deployment time, a WebLogic Workshop application is compiled into an J2EE enterprise application (an EAR file).

For more information on applications and projects, see Applications and Projects.

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, typically through the use of one or more custom Java controls that contain the web service's business logic. A JWS file is actually an ordinary Java class file that's treated specially by the WebLogic Workshop runtime. 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.

You design a JWS file in the WebLogic Workshop integrated development environment. You can use Design View to design your service visually, and Source View to modify your Java code directly. Changes you make in Design View are reflected in your code, and vice versa.

Methods and Callbacks

Your web service has a public interface that clients may call. This interface is made up of methods (also called operations) and callbacks. The methods that your web service exposes are called by clients; the callbacks are methods on the client that your web service calls to send information back to the client.

Within your web service code, you may also have non−public methods that are not exposed to clients. These methods perform internal functions in your web service.

The controls that your web service uses also expose methods and callbacks. Your web service functions as a client of the control, calling its methods and implementing its callback handlers.

Java Controls

You can use custom Java controls in your web service to implement the business logic of your web service. Custom Java controls in turn use built−in Java controls to access enterprise resources such as databases, legacy applications, and other web services. In other words, your web service interacts with a custom Java control by calling its control methods and implementing callback handlers for control callbacks, and the custom Java control calls control methods and implements callback handlers for any built−in controls it uses.

WebLogic Workshop provides built−in Java controls for connecting to common resources. Some of the built−in controls provided with WebLogic Workshop include:

- The Web Service control, for calling another web service
- The Timer control, which notifies your web service when a specified period of time has elapsed or when a specified absolute time has been reached
- The EJB control, which provides simplified access to Enterprise Java Beans (EJBs).
- The Database control, which provides simplified access to a relational database
- The JMS control, which makes it easy to send and receive messages via a Java Message Service (JMS) topic or queue.

For more information about built−in controls, see Using WebLogic Built−In Controls. For information on building custom Java controls, see Building Your Own Java Control.

Properties

Most of the elements that make up your web service—methods, callbacks, controls, and the web service itself—have properties that you can set to specify their characteristics. You can set properties in in the Property Editor in the WebLogic Workshop IDE. Each element of your web service has one or more annotations, each with a set of attributes, corresponding to the element's properties in the Property Editor. Properties are stored in your code as special Javadoc tags (annotations beginning with @). Once you are familiar with the annotations, you can edit them directly in Source View if you wish.

Conversations and Asynchronous Communication

Many business processes take time to complete. When a client makes a request from a web service, if the web service doesn't return a response right away, the client may be left waiting for it, unable to continue other operations. Web services that you build with WebLogic Workshop can address this problem by relying on asynchronous communication. In WebLogic Workshop, you implement asynchronous communication for your web service by specifying that methods and callbacks exposed by the web service participate in a conversation.

When a client and web service communicate asynchronously, the web service immediately acknowledges the client's request, then continues processing the request. The client is free to continue performing other work. When the web service completes its operations, it can return results to the client via a callback or a polling interface. All of this work happens within the context of a conversation.

The conversation correlates the client's requests and the service's response by means of a conversation ID, a unique identifier that is generated when the client initiates a conversation with the service. The conversation also keeps track of state−related data for this exchange between client and service.

For more information on conversations, see Designing Conversational Web Services. For more information on building asynchronous web services, see Designing Asynchronous Interfaces.

XML Maps

A web service communicates with clients and with many resources via XML messages. In many cases it is necessary to translate back and forth between XML and the Java method or callback arguments inside your web service. Outgoing method calls need their Java parameters translated into the appropriate XML data types; likewise incoming XML messages need their data translated into the appropriate Java method parameter types. WebLogic Workshop provides a range of facilities to assist you in creating these translations. Simple mappings happen automatically. For more complex translations, you can modify the XML map—the XML schema that WebLogic Workshop uses to translate method parameters and return values. XML Maps support the XQuery language for SQL−like traversal of XML data. If you need procedural control over the translation, you can write XScript to manipulate XML messages. Finally, XMLBeans, a technology used throughout WebLogic Workshop, provide powerful, natural and very easy translation between XML data and Java types.

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

For more information about XQuery in XML Maps, see Introduction to XQuery Maps.

FOr more information about XMLBeans, see Handling XML with XMLBeans.

Related Topics

Designing Asynchronous Interfaces

Working with Java Controls

Web Service Development Cycle

The life cycle of a web service is similar to that of any web application component: design, implement, test, debug and deploy. Designing and building the web service will typically follow design and implementation of the custom Java control(s) that implements the web service's business logic.

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 can accomplish this task in 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 Applications and Projects.
- Add the methods your web service will expose and configure each method's parameters. Invoke Java control methods as desired from each web service method. To lean about Java controls, see Working with Java Controls.
- Add any callbacks your web service will expose and configure each callback's parameters. Implement callback handlers for all callbacks from Java controls that you need to accept. 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 Designing Conversational Web Services.

The set of methods and callbacks you define for your web service is called the *public interface* or *public contract*.

Implementing Web Service Code

WebLogic Workshop recommends that you use one or more custom Java control(s) in your web service to implement the business logic of your web service. Custom Java controls in turn sometimes will need to use built−in Java controls to access enterprise resources such as databases or other web services.

You can implement your web service using both the Design and Source Views. 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. In Source View you can add code to the body of the methods and implement callback handlers for the Java controls.

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 need 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 without Debugging 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).

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.

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 Your Application.

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 application containing your web services 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 Applications.

Related Topics

What Are Web Services?

Building Web Services with WebLogic Workshop

Structure of a JWS File

Working with Java Controls

XML Strategies and Web Service Design

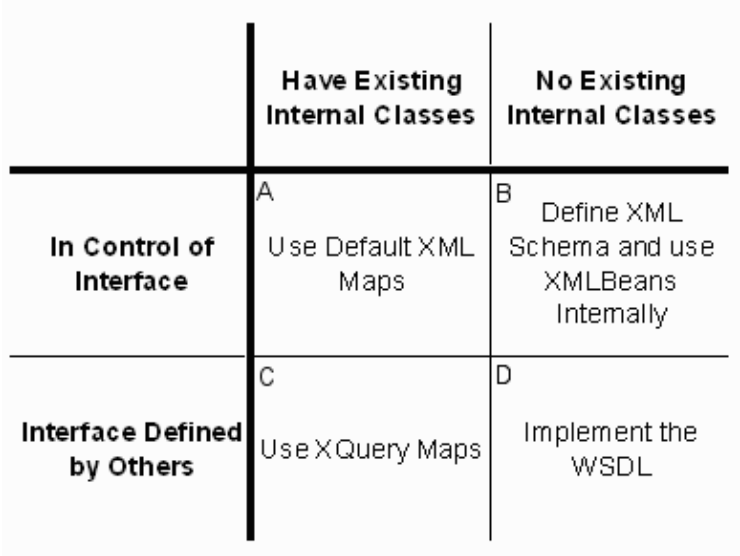
WebLogic Workshop provides several strategies for handling XML messages in web services. Which strategy you should use depends on how the web services you are designing fit into the overall application.

Strategy Criteria

The strategy you should use when implementing a web service depends on two criteria:

1. Is there existing application logic you are exposing? Specifically, are there Java classes you are using internally that are suitable to be exposed as data types in the web service's interface?
2. Will you define the interface to the web service, or is the interface externally defined? For example, is there a document schema your web service must accept?

The following diagram illustrates the matrix of strategies you may use depending on how your situation fits the preceding criteria:



Strategy A: Expose Existing Classes

In this scenario, you have existing business logic and data types you would like to expose as one or more web services, and there are no external restrictions on the interface you expose. Specifically, there are no pre−defined external data types or schema that you must support.

If the existing Java classes are suitable to be exposed in the public interface, the easiest approach in this situation is to use the classes directly as parameters and return values of the methods of your web service(s). WebLogic Workshop will automatically use *default XML maps* to express the data types as XML schema in the web service's WSDL file.

If the internal classes are not suitable to expose as public data types, define new types to be exposed in the public interface and then map the public types to the internal types using XQuery Maps by following the steps in Strategy C below.

Strategy B: Expose XML Schema

In this scenario, there are no existing data structures and no external requirements. The easiest approach is to create XML schema describing the data types you would like to use in the public interface. Place these schema in the Schemas project in your application and use the resulting XMLBeans types for the parameters of your web service's methods. WebLogic Workshop will use your schemas for the data types in the web service's WSDL file.

Strategy C: Use XQuery Maps

In this scenario, the message formats for your web service's methods are specified externally but you have existing internal data types that you wish to use. To accomplish this, you need to use XQuery Maps to map the data in the incoming messages to fields in your internal data types. You can do this by following these steps:

1. Define your web service's methods using the internal Java types as parameters
2. Create an XML schema describing the incoming message format for each method (or use existing schemas if appropriate)
3. Add the schema(s) to the Schemas project in your application
4. For a method on the web service, in Design View, double−click on the method arrow to bring up the XQuery Mapper (or select the "..." button on the parameter−xml property's schema−element attribute in the Property Editor).
5. Select the Choose button next to the XML Message Schema text box.
6. Choose a schema type to use as the incoming message format for the method.
7. Use the XQuery Mapper to designate how to map data from the incoming messages to the Java types used as method parameters.

Note: When you specify a schema−element as the incoming message format for a web service method, you are defining the complete message format for that method. If you want to share a single data type across multiple methods you must define a different outermost element for each method, each of which contains the shared data type. WebLogic Workshop maps incoming XML messages to web service operations based on the complete message schema, so message to method relationships must be one−to−one.

To learn more about XQuery Maps, see Introduction to XQuery Maps.

Strategy D: Implement a Pre−defined WSDL

In this scenario, the data types in your web service's public interface are externally specified (in a WSDL file) and you have no existing internal types. In this case, you may use the data types defined in the WSDL directly, as follows:

1. Copy the WSDL file to the Schemas project in your application. The schemas defined in the WSDL file will be compiled into XMLBeans.
2. Use the XMLBeans types directly as the arguments to your web service's methods.

Related Topics

None.

Accessing SOAP Attachments in Web Services

The SOAP Specification defines SOAP attachments as a way to include MIME data in or associated with SOAP messages. Some web service designs use SOAP attachments to pass typed files between a client and a web service. This topic describes how WebLogic Workshop web services can send and receive MIME messages as SOAP attachments.

WebLogic Workshop's SOAP attachment support is based on the SOAP Messages with Attachments specification.

WebLogic Workshop supports SOAP attachments only for web service method invocations that arrive via HTTP. Specifically, in order to support SOAP attachments a web service method must support the http−soap or http−soap12 protocols. You specify the protocols a web service supports using the @jws:protocol annotation.

SOAP attachments are held in memory, and the size of the SOAP attachment supported depends on the amount of memory available on the WebLogic Server machine.

What is MIME?

MIME stands for Multipurpose Internet Mail Extensions. As the name implies, MIME was originally developed to standardize the format and identification of email attachments. Specifically, MIME messages can contain text, images, audio, video, or other application−specific data. The standardized MIME type system has found uses in many other applications besides email. Whenever you use a Download button on a web page or play a media file by clicking on a link in a browser, MIME is used by the browser or other application to identify the type of the linked file and determine how to interact with that resource.

To learn more about MIME, including MIME content types, see RFC 1341.

The Java API includes the javax.activation package that can decode MIME messages and provide application access to the operations that are possible on each type of content.

Handling SOAP Attachments in a Web Service

Handling SOAP attachments in a WebLogic Workshop web service is easy. When you want to access MIME SOAP attachments from a web service method, just reference the javax.activation.DataHandler class in the method's signature. See the examples that follow.

Example: Receiving SOAP Attachments in a Web Service Method

To specify that a web service method expects exactly one SOAP attachment, include an argument of type javax.activation.DataHandler in the method's signature as show below in a web service named EchoAttachmentService.jws:

```
import javax.activation.DataHandler;

public class EchoAttachmentService implements com.bea.jws.WebService
{
  static final long serialVersionUID = 1L;
```

```
  /**
  * @common:operation
  * @jws:protocol form-get="false" form-post="false"
  */
  public String echoAttachment(DataHandler dh)
  {
    return("***Service received DataHandler of type: " + dh.getContentType());
  }
}
```

The preceding example returns a String containing the content type of whatever MIME message it receives as an attachment. Examples of MIME content types are text/plain, text/html and application/octet-stream. There are many other registered MIME content types.

In the preceding example, the form-get and form-post attributes of the @jws:protocol annotation have been set to false. The FROM GET and FORM POST operations have no way to convey SOAP attachments. To access a web service that accepts SOAP attachments, clients must send raw SOAP messages (via some supported transport protocol, typically HTTP or JMS).

To specify that a web service expects multiple SOAP attachments, include an argument of type javax.activation.DataHandler[] in the method's signature.

Example: Returning SOAP Attachments from a Web Service Method

To specify that a web service method returns one or more SOAP attachments to the client, declare the method's return value as javax.activation.DataHandler or javax.activation.DataHandler[].

The FileRetrieverService web service below demonstrates a web service method whose return value is a MIME message packaged as a SOAP attachment:

```
import javax.activation.DataHandler;
import javax.activation.FileDataSource;

public class FileRetrieverService implements com.bea.jws.WebService
{
  static final long serialVersionUID = 1L;

  /**
  * @common:operation
  * @jws:protocol form-get="false" form-post="false"
  */
  public DataHandler getFileAsAttachment(String filename)
  {
    return(new DataHandler(new FileDataSource(filename)));
  }
}
```

The getFileAsAttachment method returns the named file as a MIME message SOAP attachment.

You may also return an array of MIME message SOAP attachments from a web service method by declaring the method's return value to be of type javax.activation.DataHandler[].

Tested Interoperation Scenarios

The following interoperation scenarios have been tested:

- WebLogic Workshop web services can exchange one or more SOAP attachments.
- WebLogic Workshop web services and WebLogic Server (servicegen) web services can exchange single SOAP attachments but not multiple attachments (DataHandler[] in the web service method signature).
- WebLogic Workshop web services and Apache Axis web services can exchange single SOAP attachments but not multiple attachments. You must develop the Apache Axis client using Axis utilities directly as opposed to building the client automatically from the WebLogic Workshop web service's WSDL file.
- WebLogic Workshop successfully interoperates with the SOAP attachment test scenarios that make up the SOAP Builders Round 4 interoperability test suite.

Related Topics

@jws:protocol Annotation

javax.activation.DataHandler (at java.sun.com)

Specifying SOAP Handlers for a Web Service

When designing a web service that accepts or returns SOAP messages, you may specify that the SOAP messages should be processed by *message handlers*. Handlers on incoming messages are invoked before the message is delivered to the web service operation, and handlers on outgoing messages are invoked after the web service operation has completed. The web service itself is unaware of the presence of handlers. SOAP message handlers are sometimes referred to as *interceptors*.

SOAP message handlers can be used for a variety of purposes. Examples include message logging, audit trails, and message transformations. Message handlers should *not* be used to implement security or encryption or to manage the redirection of messages; those facilities are provided by other mechanisms.

This topic contains the following sections:

JAX−RPC and SOAP Handlers

The standard Java interface to web services is called JAX−RPC, which stands for Java API for XML−Based Remote Procedure Calls. The term *remote procedure call*, or RPC, is a historical term that describes an application running on one machine invoking a procedure (a method) on another machine. Invoking a web service method is one form of a remote procedure call. The JAX−RPC API provides a Java interface that can be used by both a web service implementation and its clients to manipulate the underlying XML messages without having to be concerned with the details of the XML messages.

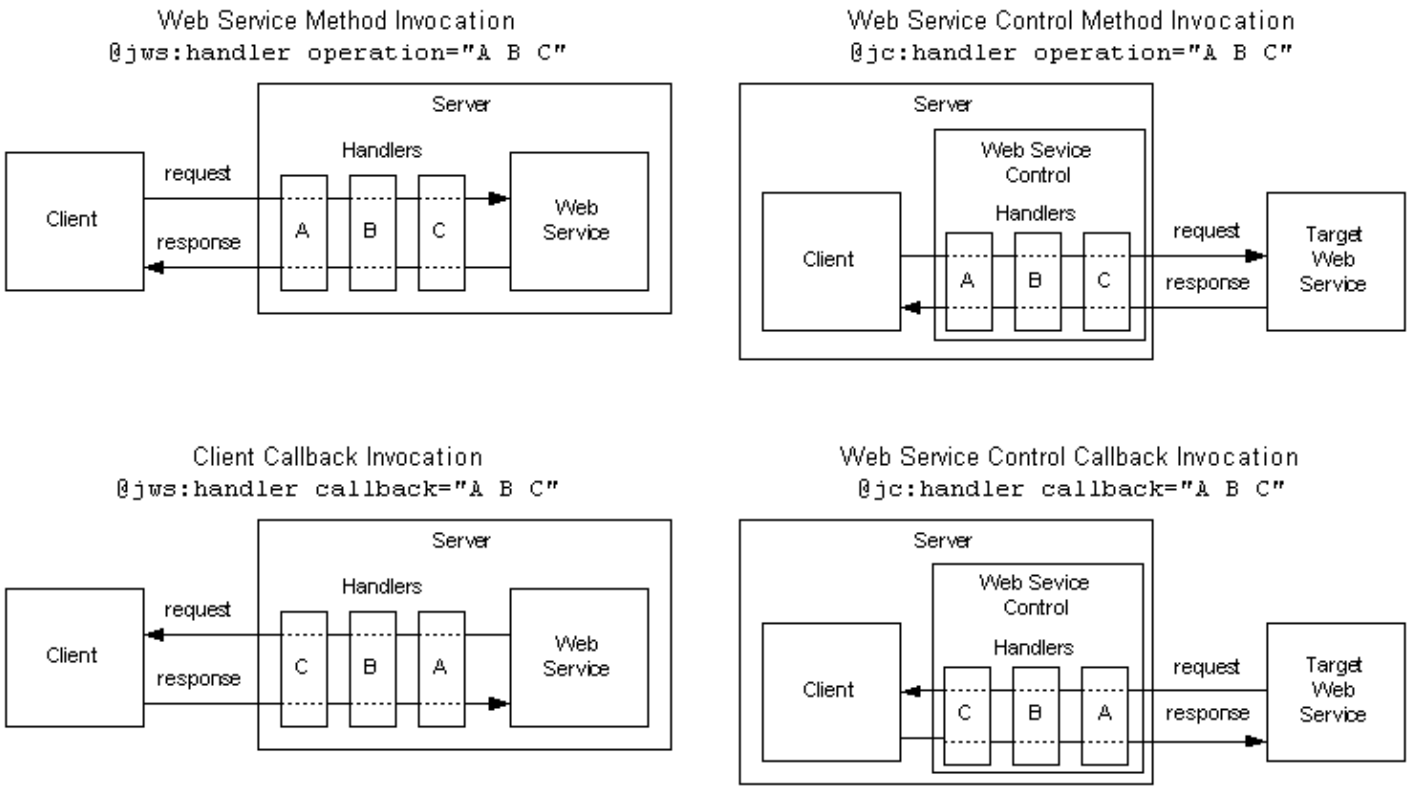
WebLogic Workshop leverages the classes defined in the message handler portion of the JAX−RPC API; specifically, the javax.xml.rpc.handler package. WebLogic Workshop provides annotations that allow you to easily specify the message handler(s) you would like to associate with a web service or Web Service control. The @jws:handler annotation is used to specify message handlers for JWSs, and the @jc:handler annotation is used to configure message handlers in Web Service control JCXs. The WebLogic Workshop runtime uses the annotations in a JWS or JCX file to automatically manage the message handlers.

Where Handlers Fit in the Message Path

Each web service or Web Service control may have a set of associated handlers.

The left column of the diagram below illustrates a message scenario in which the @jws:handler annotation is applied to the Web Service in the diagram. Invocation of a web service operation consists of an incoming *request* message followed by an outgoing *response* message. The scenario illustrates the case of three message handlers, A, B and C, specified in that order. Note the sequence in which incoming and outgoing messages encounter the handlers.

In WebLogic Workshop, web services may also define and invoke callbacks. A callback is a message to the client *initiated by the JWS*. In the case of a callback, the web service invokes a callback on the client by sending an outgoing request message and subsequently receiving an incoming response message.



The right column of the preceding diagram illustrates use of message handlers on a Web Service control. In this case, your code (a web service, business process or page flow) is the Client, and you are calling the Target Web Service via a Web Service control. Note the sequence in which incoming and outgoing messages encounter the handlers, and how it differs in the method invocation and callback cases.

JAX−RPC Specification

If you want to know more about the JAX−RPC Specification, visit the Java API for XML−Based RPC (JAX−RPC) page at java.sun.com.

The Handler Interface

The JAX−RPC API message handler interface is javax.xml.rpc.handler.Handler. When you design a handler, you define your own Java class that implements the javax.xml.rpc.handler.Handler interface. The Handler interface defines the following core methods:

init(HandlerInfo)

The init method allows access to initialization parameters that are obtained from the handler configuration file, described below.

handleRequest(SOAPMessageContext)

The handleRequest method of a handler is invoked whenever a request message passes through the handler. In the case of a web service method invocation, this is an incoming message. In the case of a callback, this is an outgoing message.

handleResponse(SOAPMessageContext)

The handleResponse method of a handler is invoked whenever a response message passes through the handler. In the case of a web service method invocation, this is an outgoing message. In the case of a callback, this is an incoming message.

handleFault(SOAPMessageContext)

The handleFault method of a handler is invoked whenever a SOAP Fault passes through the handler.

The javax.xml.rpc.handler.Handler API specifies that the handleXxx methods take a javax.xml.rpc.handler.MessageContext as their argument type. In WebLogic Workshop, the arguments will in fact always be of the more specific type javax.xml.rpc.handler.soap.SOAPMessageContext. The SOAPMessageContext class provides access to all parts of the SOAP message.

weblogic.webservices.GenericHandler

The JAX−RPC API provides a generic handler from which you may derive your own handler classes. The JAX−RPC generic handler class is javax.xml.rpc.handler.GenericHandler.

WebLogic Server also provides a generic handler class that you may use as the basis for new handlers you create. The class is weblogic.webservices.GenericHandler. To learn more about the weblogic.webservices.GenericHandler class, see Creating SOAP Message Handlers to Intercept the SOAP Message.

Handler Chains

You may specify that more than one handler is associated with a particular web service. When multiple handlers are associated with one web service, the handlers form a *handler chain*. You may specify a handler chain using a handler configuration file, as is described in Specifying Handler Chains in a Handler Configuration File, below.

Note: the JAX−RPC API defines the javax.xml.rpc.handler.HandlerChain and javax.xml.rpc.handler.HandlerRegistry interfaces, which allow fine control over handler chains and the order of invocation of individual handlers in a chain. WebLogic Workshop 8.1, like most J2EE−based web service implementations, does not support the HandlerChain and HandlerRegistry interfaces.

Handler Annotations

In WebLogic Workshop, you associate message handlers with a web service using the @jws:handler annotation and you associate message handlers with a Web Service control using the @jc:handler annotation.

The @jws:handler annotation is applied to the class in a JWS file. The @jc:handler annotation is applied to the interface declaration in a Web Service control JCX file.

The @jws:handler and @jc:handler annotations have identical attributes:

operation="<*handler−specification*>"

specifies the handler(s) that will be called on requests and responses to and from all operations of the web service or web service control that accept SOAP messages.

callback="<*handler−specification*>"

specifies the handler(s) that will be called on requests and responses to and from all callbacks of the web service or web service control that accept SOAP messages.

file="<*handler−config−file*>"

specifies an optional configuration file that defines handler chains and handler initialization data.

Specifying Handlers Directly

You may specify handlers directly by setting the operation or callback attribute to a space−separated list of handler class names as shown in the following example:

```
package handler;

/**
 * @jws:handler operation="handler.ConsoleLoggingHandler"
 */
public class HandlerExample1 implements com.bea.jws.WebService
{
    static final long serialVersionUID = 1L;

    /** @common:operation */
    public String echoString(String inputString)
    {
        return inputString;
    }
}
```

The list of handlers specified in the operation attribute applies to all methods of the web service that accept SOAP messages.

Note: it is possible, using the @jws:protocol annotation, to configure methods or callbacks that do not accept SOAP messages. SOAP message handlers will not be applied to methods that do not accept SOAP messages.

To specify that more than one handler should be invoked, supply the list of handlers in a space−separated list:

@jws:handler operation="handler.ConsoleLoggingHandler somepackage.SomeOtherHandler"

See Where Handlers Fit in the Message Path, above, for examples of the order of handler invocations in various scenarios.

Specifying Handler Chains in a Handler Configuration File

As an alternative to specifying the handlers directly as the value of the operation or callback attribute, you may configure named handler chains in a *handler configuration file* and reference the handler chain names as the value of the operation or callback attribute. An example handler configuration file is shown below:

```
<?xml version="1.0" encoding="UTF−8"?>
<hc:wlw−handler−config xmlns:hc="http://www.bea.com/2003/03/wlw/handler/config/">
    <hc:handler−chain name="LoggingHandler">
        <hc:handler handler−name="Logger" handler−class="handler.ConsoleLoggingHandler"/>
        <hc:handler handler−name="Auditor" handler−class="handler.AuditHandler"/>
    </hc:handler−chain>
</hc:wlw−handler−config>
```

Note: a <handler−chain> with no handlers is valid. It provides a means to remove handlers without changing the referencing source file.

The preceding handler configuration file defines a handler chain named LoggingHandler. The example web service that follows assumes the preceding XML is saved in a file named consoleLoggingConfig.xml and references the LoggingHandler chain as the value of the operation attribute:

```
package handler;

/**
 * @jws:handler operation="LoggingHandler" file="consoleLoggingConfig.xml"
 */
public class HandlerExample2 implements com.bea.jws.WebService
{
    static final long serialVersionUID = 1L;

    /** @common:operation */
    public String echoString(String inputString)
    {
        return inputString;
    }
}
```

The handler chain specified in the operation or callback attribute applies to all methods or callbacks of the web service that accept SOAP messages. See Where Handlers Fit in the Message Path, above, for examples of the order of handler invocations in various scenarios.

Note: it is possible, using the @jws:protocol annotation, to configure methods or callbacks that do not accept SOAP messages. SOAP message handlers will not be applied to methods that do not accept SOAP messages.

Each <handler> element in a handler configuration file may also specify initialization parameters for the handler in <init−param> elements as shown in the following example:

```
<hc:handler handler−name="MyHandler" handler−class="MyHandler"/>
    <hc:init−param>
        <hc:description>First Param</hc:description>
        <hc:param−name>param1</hc:param−name>
        <hc:param−value>value1</hc:param−value>
    </hc:init−param>
</hc:handler>
```

All defined parameters are delivered to the handler's init method when it is invoked.

The handler configuration file approach has one advantage over the direct handler specification approach: you can reconfigure a handler chain simply by modifying the handler configuration file and redeploying the application − there is no need to recompile the application (unless you add or modify handler classes).

An Example Message Handler

The example below shows the code for the ConsoleLoggingHandler referenced in the previous examples:

```
package handler;

import java.util.Iterator;
import javax.xml.rpc.handler.MessageContext;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.soap.SOAPElement;

// for readability, using weblogic API instead of javax.xml.rpc.handler.GenericHandler
// See http://edocs.beasys.com/wls/docs81/webserv/interceptors.html#1060763
import weblogic.webservice.GenericHandler;

/**
 * Purpose: Log all messages to the Server console
 */
public class ConsoleLoggingHandler extends GenericHandler
{
    /**
     * Handles incoming web service requests and outgoing callback requests
     */
    public boolean handleRequest(MessageContext mc)
    {
        logSoapMessage(mc, "handleRequest");
        return true;
    }
    /**
     * Handles outgoing web service responses and incoming callback responses
     */
    public boolean handleResponse(MessageContext mc)
    {
        this.logSoapMessage(mc, "handleResponse");
        return true;
    }

    /**
     * Handles SOAP Faults that may occur during message processing
     */
    public boolean handleFault(MessageContext mc)
```

```
    {
        this.logSoapMessage(mc, "handleFault");
        return true;
    }

    /**
     * Log the message to the server console using System.out
     */
    protected void logSoapMessage(MessageContext mc, String eventType)
    {
        try
        {
            System.out.println("*****************************");
            System.out.println("Event: "+eventType);
            System.out.println("Endpoint Method: " + getMethodName(mc));
            System.out.println("Soap message is: \n " +
                com.bea.wlw.runtime.jws.soap.util.SAAJUtil.SOAPPart2String(
                    ((SOAPMessageContext)mc).getMessage() ) + "\n");
            System.out.println("*****************************");
        }
        catch( Exception e )
        {
            e.printStackTrace();
        }
    }

    /**
     * Get the method Name from a SOAP Payload.
     */
    protected String getMethodName(MessageContext mc)
    {
        String operationName = null;

        try
        {
           SOAPMessageContext messageContext = (SOAPMessageContext) mc;

           // assume the operation name is the first element
           // after SOAP:Body element
           Iterator i = messageContext.
               getMessage().getSOAPPart().getEnvelope().getBody().getChildElements();
           while ( i.hasNext() )
           {
               Object obj = i.next();
               if(obj instanceof SOAPElement)
               {
                   SOAPElement e = (SOAPElement) obj;
                   operationName = e.getElementName().getLocalName();
                   break;
               }
           }
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
        return operationName;
    }
}
```

If you use the weblogic.webservices.GenericHandler class as the base class for your handler, as the ConsoleLoggingHandler does, you do not need to implement methods (such as init()) that you do not need to customize.

Message Handlers and Web Service Controls

To call an external web service from WebLogic Workshop, you use a Web Service control. To configure message handlers for messages that pass between your application and the external web service, you specify the @jc:handler annotation on the interface defined in the Web Service control's JCX file.

The example below illustrates the @jc:handler annotation in the TargetServiceControl.jcx file that defines a Web Service control for communication with the TargetService external web service:

```
/**
 * @jc:location http-url="TargetService.jws" jms-url="TargetService.jws"
 * @jc:wsdl file="#TargetServiceWsdl"
 * @jc:handler callback="handler.ConsoleLoggingHandler" operation="handler.ConsoleLoggingHand
 */
public interface TargetServiceControl extends com.bea.control.ControlExtension, com.bea.cont
{
    ...
}
```

Implementation Details

Message Handlers Not Supported Via HTTP GET

WebLogic Workshop 8.1 does not support SOAP message handlers on messages that arrive via HTTP GET.

When testing WebLogic Workshop web services with message handlers via Test View, remember that the Test Form page uses HTTP GET and the Test XML page uses HTTP POST. Your message handlers will not be invoked if you submit web service requests via HTTP GET from the Test Form page. Use the Test XML page instead.

Message Handler Limitations

All J2EE-based web service implementations that support SOAP message handlers place limitations on the types of operations that message handlers may perform on messages. This is because the web services are implemented as container-managed objects such as EJBs, and certain types of operations on SOAP messages will interfere with the operation of the container.

Below is a non-exhaustive list of things you can *not* do in a message handler:

- You cannot encrypt the message body of an incoming web service method invocation message. If you do so, the WebLogic Workshop runtime will not be able to dispatch the message to the web service's implementation.
- You cannot change the conversation ID in the SOAP message header (if present).
- You cannot change the name of the target method in the SOAP message. The target method name is typically the name of the top-level element in the SOAP:Body.

WebLogic Workshop provides container−based security outside the context of message handlers. To learn more, see WebLogic Workshop Security Overview.

Related Topics

Web Service Control

@jws:handler Annotation

@jc:handler Annotation

Generating SOAP Faults from a Web Service

For web services that use SOAP as their message format, the proper way to inform a client about errors encountered while processing an invocation message is to return a *SOAP fault*. A SOAP fault element may contain one or more *detail* objects describing the problem encountered.

In a WebLogic Workshop web service, there are three ways for you to create a SOAP fault and return it to the client:

- You can construct and throw an exception of type com.bea.jws.SoapFaultException using an XMLBeans type that represents the entire SOAP fault element. This approach gives you full control over the content of the fault. The XMLBeans type that you use depends on which fault type (from the WSDL file) you want to throw.
- You can construct an exception of type com.bea.jws.SoapFaultException and add fault detail objects as desired. This provides a simple way to build a fault that has complex types in the fault detail.
- You can construct and throw an exception of type javax.xml.rpc.soap.SOAPFaultException. This approach is provided for compatibility purposes. This exception type is specific to the SOAP 1.1 envelope and can only be thrown while processing a SOAP 1.1 request. You can determine the SOAP version of the current request from within a web service method by calling the getProtocol() method of com.bea.control.JwsContext.

SOAP Faults, Details and XMLBeans

The WSDL file for a web service specifies the types of SOAP faults that may be returned by the web service, including the particular object types that may be included in the SOAP fault detail. WSDL files you receive from external sources and implement as WebLogic Workshop web services may include SOAP fault definitions.

The easiest way to handle SOAP fault detail types is with XMLBeans. If you add a WSDL file to the Schemas project of your application, all of the types defined in the WSDL's schema, including any SOAP fault detail types, will be compiled into XMLBeans. You can use the generated XMLBeans types to create SOAP fault detail objects and use the com.bea.jws.SOAPFaultException API to include the detail objects in SOAP faults you throw.

Related Topics

How Do I: Handle Errors In a Web Service?

JwsContext Interface

com.bea.jws.SoapFaultException

Documenting Web Services

The web services you build can 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 a web service are included in the WSDL that is generated for that web service. 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 topic 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 bold text (<b> tag), italics (<i> tag), lists (<ul> and <ol> tags with member <li> tags), etc.

Javadoc Annotations and Comment Structure

If Javadoc annotations (tokens beginning with @) are present in a comment, they must be located at the end of the comment. Any text following a Javadoc annotation (and preceding another annotation) is considered arguments to the annotation. If you include additional commentary after the Javadoc annotations in a Javadoc comment, the WebLogic Workshop compiler will generate errors about inappropriate arguments to annotations.

For example, the following Javadoc comments 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 attribute 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 attribute 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 annotations within comments. Notice that the comment discusses the @jws:conversation−lifetime annotation. If the annotation were located at the start of the line, it would be interpreted as an annotation (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 annotation form being interpreted as an active annotation.

Class Comment

A Javadoc comment that immediately precedes a class is called the *class comment*. Test View will display the class comment on the Overview Page.

Method comments are also included in the WSDL file generated from a JWS file.

Method Comment

A Javadoc comment that immediately precedes a method 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 the WSDL file generated from a JWS file.

Related Topics

Introduction to Java

Structure of a JWS File

WSDL Files: Web Service Descriptions

Structure of a JWS File

A JWS file contains a Java Web Service. This topic is an overview of the JWS file format.

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 annotations specific to WebLogic Workshop that enable or access features in the WebLogic Workshop runtime.
- 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 the WebLogic Workshop runtime.

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

Inner Classes

Methods

Internal Methods

Package Statement

The package statement specifies the Java package in which the classes and objects in this file reside. 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 Application pane, WebLogic Workshop will update the package name in the moved file. However, you may 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 directly in your Java class, 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 com.bea.control.TimerControl;  // TimerControl API
import com.bea.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 implements com.bea.jws.WebService
{
    ...
}
```

The implements com.bea.jws.WebService clause indicates that this class is a web service. The WebService Java interface is a marker interface only, it declares no methods.

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. These are variables whose scope is at the level of the class. Member variables declared in a JWS file are valid and are persisted as long as the instance of the class exists. If the web service is conversational, member variables are persisted per−conversation−instance for as long as each conversation exists.

```
public class MyService implements com.bea.jws.WebService
{
    int myMemberVariable;
```

```
    ...
}
```

Non−annotated 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 Designing Conversational Web Services.

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 @jc:control annotation, the member variable represents a Java control. Java controls provide convenient interfaces to resources such as databases, Enterprise JavaBeans (EJBs), other web services or arbitrary business logic in custom Java controls.

```
public class MyService implements com.bea.jws.WebService
{
    int myMemberVariable;
    /**
     *  @jc:control
     */
    TimerControl delayTimer;
    ...
}
```

For more information on controls, see Working with Java Controls.

JwsContext Object

Each JWS file may have a special member variable of type com.bea.control.JwsContext annotated with the @common:context annotation.

```
public class MyService implements com.bea.jws.WebService
{
    int myMemberVariable;
    /**
     *  @common:context
     */
    com.bea.controlJwsContext context;
    ...
}
```

This instance of the JwsContext object may be used to access aspects of the web service'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 Web Service control instance is not populated. This code will throw a NullPointerException at runtime.

```
public class MyService implements com.bea.jws.WebService
{
    /** @common:control */
    private SomeServiceControl someService;
    ...
    String serviceResult = someService.getResult();
}
```

The following code is correct:

```
public class MyService implements com.bea.jws.WebService
{
    /** @common:control */
    private SomeServiceControl someService;
    ...
    String serviceResult;
    /**
     * @common:operation
     */
    public void someMethod()
    {
        ...
        serviceResult = someService.getResult();
        ...
    }
}
```

Callback Interface

When you add a callback to your service in Design View, the signature for the callback method becomes a member of the 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. A Callback interface is automatically added to a web service's JWS file when a callback is added in Design View.

```
public class MyService implements com.bea.jws.WebService
{
    int myMemberVariable;

    /**
     *  @common:control
     */
    TimerControl delayTimer;

    public static interface Callback extends com.bea.control.ServiceControl
    {
        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 JCX files, you may edit it manually if you wish.

The callback interface is included in the WSDL file for your web 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. Callbacks are a feature of asynchronous web services. To learn more about asynchronous web services, see Designing Asynchronous Interfaces.

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
- mirror of a database record structure, into which query results can be mapped

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 Introduction to XQuery Maps.

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 its result to an object of that class.

For more information on using a database from your service, see Database Control.

The example below demonstrates the declaration of Customer as an inner class of the MyService class.

```
public class MyService implements com.bea.jws.WebService
{
    int myMemberVariable;

    /**
     *  @common:control
     */
    TimerControl delayTimer;

    public static interface Callback extends com.bea.control.ServiceControl
    {
        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 @common:operation annotation. 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 implements com.bea.jws.WebService
{
    int myMemberVariable;

    /**
     *  @common:control
     */
    TimerControl delayTimer;

    public static interface Callback extends com.bea.control.ServiceControl
    {
        public void myCallback(String stringArg);
    }
    public Callback callback;

    public class Customer
    {
        public int custid;
        public String name;
        public String emailAddress;
    }

    /**
     *  @common:operation
     */
    public String myMethod(String stringArg)
    {
        ...
    }
   ...
}
```

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 implements com.bea.jws.WebService
{
```

```
    int myMemberVariable;

    /**
     *  @common:control
     */
    TimerControl delayTimer;

    public static interface Callback extends com.bea.control.ServiceControl
    {
        public void myCallback(String stringArg);
    }
    public Callback callback;

    public class Customer
    {
        public int custid;
        public String name;
        public String emailAddress;
    }

    /**
     *  @common: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 specified 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. A WSDL description allows a client to utilize a web service's capabilities without knowledge of the implementation details of the web service.

Contents of a WSDL File

A WSDL file contains 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
- The individual methods names and signatures (WSDL refers to methods as *operations)*
- The protocols and message formats allowed for each method
- The URLs used to access the web service

Imported WSDL Files

When you want to use an external web service from within 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 Web Service control. The Web Service control may then be used from your application like any other WebLogic Workshop Java control.

Some web service tools produce WSDL files that do not contain an *XML declaration*. WebLogic Workshop requires that all XML files contain an XML declaration. The XML declaration is just the first line of an XML file of the following form:

```
<?xml version="1.0" encoding="utf−8" ?>
```

If you receive a WSDL file that does not contain an XML declaration, you must add a declaration to the file using a text editor before you can use the WSDL file in WebLogic Workshop.

Note that the encoding attribute is not required. If an encoding attribute is not present, the default encoding is utf−8.

To learn more about producing a Web Service control file from a WSDL file, see Creating a Web Service Control from a WSDL File.

To learn more about JCX files, see JCX Files: Implementing Controls.

If the web service you wish to use was built with WebLogic Workshop, you can create a Web Service control JCX file directly from the web service's JWS file.

Note: You should not use a Web Service control to invoke a web service that resides in the same application. Invoking a web service via a Web Service control means marshalling the method parameters into a SOAP message on the calling end and unmarshalling the SOAP message on the receiving end, then again for the method return value. This is very inefficient when the invocation is local. You would usually be tempted to invoke one web service from another if the called web service included business logic you want to access from the calling web service.

In general, you should place business logic in custom Java controls instead of in web services. This allows you to access the business logic from various contexts in the application (web services, other controls, page flows) without incurring the cost of data marshalling and unmarshalling. Web Service controls should *only* be used to invoke web services that are truly external to your application.

To learn more about Web Service controls, see Web Service Control.

Generating 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: Generate a WSDL File?.

If your clients are web services built with WebLogic Workshop, they can use a Web Service control JCX file generated directly from your web service's JWS file.

Related Topics

JCX Files: Extending Controls

Applications and Projects

Web Service Control

W3C WSDL Specification

Versioning Web Services

This topic provides information you may need for versioning web services created in WebLogic Workshop in such a way that they will continue to support clients without disruption.

There are two areas to consider in regard to versioning a web service:

- The public interface, as described by the web service's WSDL file
- The web service implementation, including its conversational state

Versioning the Public Interface

You can add new operations to the public interface of a web service, but you cannot change or remove existing operations without potentially disrupting clients that rely on the web service. By adding new operations and making them known to clients, you can gradually shift clients over to a new set of operations, but you should leave the original operations intact for backward compatibility.

Versioning the Implementation

WebLogic Workshop uses Java serialization to persist the state associated with web service requests and conversations. For this reason, the primary requirement for supporting versioned implementations is maintaining backward serialization compatibility. Specifically, it must be possible to load state into the current version of a class that was stored using any prior versions of that class. Compatible changes are described in the Java Serialization Specification and include (but are not limited to):

- Changing the implementation of a method
- Adding new methods
- Adding new instance variables

Types of changes that are not version compatible include (but are not limited to):

- Changing the type of a variable
- Moving a class within an inheritance hierarchy

You should avoid making these kinds of changes if you want to maintain backward compatibility.

This model is known as "hot fix" versioning because while there can be multiple prior versions of an implementation, there can only be one current version of an implementation. Deploying an updated version of a class creates the new current version. At that point, all conversational instances that receive requests for that service will use the current version, regardless of which version was used to process any prior requests.

Web services created in WebLogic Workshop include a default field, serialVersionUID, for tracking binary compatiblity between web service versions. This value remains constant as long as subsequent revisions of the web service maintain binary compatibility. This value is used by the Java virtual machine, and you should not modify it. Should WebLogic Workshop detect that a class does not have a serialVersionUID field but is being serialized as part of a request to a web service or conversation state, WebLogic Workshop generates a warning message.

Related Topics

None.

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:

- ***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−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***.
- ***http−soap12***: Messages are SOAP−formatted according to the SOAP 1.2 protocol and delivered over HTTP. The default namespace for the SOAP 1.2 envelope is http://www.w3.org/2002/12/soap−envelope. By default ***http−soap12*** is set to ***false***.
- ***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−soap12***: Messages are SOAP 1.2−formatted and delivered over a JMS queue. The default namespace for the SOAP 1.2 envelope is http://www.w3.org/2002/12/soap−envelope. By default ***jms−soap12*** 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

You may configure a WebLogic Workshop web service or web service method to accept requests via JMS (Java Message Service) using the jms−soap, jms−soap12 or jms−xml attributes of the @jws:protocol annotation. To build a client that sends messages to a WebLogic Workshop web service over a JMS queue, you need to configure the client to:

1. Instantiate a javax.naming.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	The context factory for the JMS service
PROVIDER_URL	Host URL, using the t3 protocol (e.g., "t3://localhost:7001")

2. Connect to the queue named jws.queue. (This queue is already configured on WebLogic Server, you do not need to create it.)
3. Depending on the web service's @jws:protocol settings, format request messages as XML or SOAP in a javax.jms.TextMessage object. If the web service is conversational, see Conversing with Non−Workshop Clients to learn how to include conversation headers in the request messages.

   Each message must also have a String property named ***URI*** that identifies the target web service. For example, if the URL to the web service is http://localhost:7001/WebServices/jms/JMSService.jws, specify the ***URI*** property as /WebServices/jms/JMSService.jws.
4. Send the message.

Note: the target Web Service must not be a conversational web service.

SOAP Encodings

The SOAP specification allows two types of SOAP messages, called "document/literal" and "SOAP RPC". The soap−style attribute of the @jws:protocol annotation determines which SOAP format a web service or web service method accepts.

In both SOAP formats, the top−level element of the SOAP body must be exactly the namespace−qualified name of the target web service method. Method parameters are specified by child elements.

The following example shows invocation of the Hello method (of some web service) using the SOAP document/literal syntax. Note that the <name> parameter has no data type specification.

```
<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:Body>
    <Hello xmlns="http://www.openuri.org/">
      <name>John</name>
    </Hello>
  </SOAP−ENV:Body>
</SOAP−ENV:Envelope>
```

Contrast with the following example, which invokes the same method using the SOAP RPC syntax. Note that the <name> parameter's data type is specified by the attribute xsi:type="xsd:string".

```
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:Body SOAP−ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <Hello xmlns="http://www.openuri.org/">
      <name xsi:type="xsd:string">John</name>
    </Hello>
  </SOAP−ENV:Body>
</SOAP−ENV:Envelope>
```

The preceding examples are not intended to explain the complex topic of SOAP encodings. To learn more about SOAP encodings, see the SOAP specifications at www.w3.org or consult a book on SOAP. You can also consult the target web service's WSDL file for an exact description of the messages expected by each of the web service's methods.

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?

Sample: JMS−XML Protocol

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. These properties are described int the following list:

- ***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.

Related Topics

None

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. If a method or callback in your web service calls methods in another control, WebLogic Workshop also includes those methods in the same transaction. If one of these controls calls a method in another control, Web Logic Workshop includes that control and so on. This topic explains basic transactional behavior in WebLogic Workshop.

The Transactional Context

Included within the transactional context are the start and finish methods of a conversation, 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. If a call to a method that starts a conversation fails, the transaction rolls back and the conversation is not initiated. If a call to a method that ends a conversation fails, the transaction rolls back and the conversation is not completed.

Any operations performed by a Web Service 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 Web Service control, the operation performed by the Web Service control will not be rolled back.

If a method on a JCX control is message−buffered, 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 the control, the method will never be called. This holds true for any control that is a JCX file. To ensure that a method participates in the transactional context, set the enable attribute of the method's @common:message−buffer annotation to **true**. Note that a buffered method must return void.

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:

```
/**
 * @common:operation
 * @jws:conversation phase="finish"
 */
public void finish()
{
    try
```

```
    {
         bank.cancelAnalysis()
    }
    catch (ProxyException pe)
    {
        // Log the error, but don't rethrow.
    }
}
```

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.

Some third−party controls can participate in transactions and interact with transactional resources; others cannot. Check with your control vendor if you need to understand how a third−party control behaves with respect to transactions.

Initiating Transactions from a Page Flow

When a page flow calls a method in a Java control, that control begins an implicit transaction. When that method returns execution to the page flow, the control commits the operations performed by that method and the transaction completes. This is a little different from the way web service methods handle transactions.Since web service methods begin and complete transactions, any controls that the Web Service uses will execute in the same transaction. Page flows do not begin and complete transactions. Therefore, there needs to be a way for a page flow action method to group multiple method calls into the same transaction. Fortunately, you can use the Java Transaction API (JTA) to begin and complete transactions in a Page Flow Action method. If that Action method calls a method in another control, that control will not create a new transaction. Instead, it will execute within the Page Flow's transaction. For more information on how to do this, see Introducing Transactions.

Transactions and Connection Pools

WebLogic Workshop uses a built−in connection pool, cgPool, to track conversation state. If you create a Database control that uses a data source created on a different connection pool, you may get an error that stating that you already have a transactional context created in the first pool (the one that Workshop is using to track state), and you cannot use data sources from two different connection pools within the same transaction. You can solve this problem by using a transactional (XA) driver, but doing so may add unnecessary overhead to your application. Instead, you can share a single connection pool across multiple data sources.

To share a connection pool across multiple data sources, you can modify the properties of cgPool to point to your application database (the one on which the Database control is based). Alternately, you can create a new connection pool that points to your application database, and modify cgDataSource, which is the data source in cgPool that's used to manage WebLogic Workshop's internal state, to use your new connection pool. This ensures that WebLogic Workshop's internal state tracking and your application are using the same connection pool, and transactions will automatically be coordinated across the two. An application designed in this manner will run faster than one that uses an XA driver for the same task.

Handling Transaction Time−out

By default, the implicit transaction initiated by a web service times out after five minutes. If your web service performs a database operation that takes longer than five minutes, the transaction may time out before the operation is complete. One way to solve this problem is to use a data source that is not configured to be transactional.

Another approach is to change the interval allowed before the transaction times out. This can be done by setting the value of the <transaction−timeout> element in the wlw−config.xml file.

The third way to handle transaction time−out has been deprecated since the SP2 release. This technique should not be used when writing new code. You can change the time−out interval by changing the property that specifies the transaction timeout interval. The following code snippet shows how to set this property:

```
import javax.transaction.Transaction;
import weblogic.transaction.TxHelper;
import weblogic.transaction.internal.TransactionImpl;
...
Transaction tx = TxHelper.getTransaction();
if (tx instanceof TransactionImpl)
    {
    ((TransactionImpl)tx).setProperty(weblogic.transaction.internal.Constants.TX_TIMEOUT_SECS_P
                    new Integer(nSeconds));
    }
```

For more information, see the following topics on e−docs.bea.com:

Managing Transactions

Programming Weblogic JTA

Related Topics

Transaction Basics

Transforming XML Messages with XQuery Maps

You can use XQuery maps to reshape the XML messages that web services send and receive.

Topics Included in This Section

Introduction to XQuery Maps

Learn about what XQuery maps are and when to use them.

Using ECMAScript From XQuery Maps

Learn about using ECMAScript to extend XQuery maps.

Related Topics

Handling XML With ECMAScript Extensions

XQuery Map Samples

Introduction to XQuery Maps

XQuery maps provide a way for you to reshape the XML messages that web services send or receive. You use an XQuery map as a kind of translation layer between a web service method or callback and the outside world (its client, or a resource). Applying an XQuery map allows you to change the shape of a message dictated by the WSDL into the shape you want to use in your java program.

Note: In previous versions of WebLogic Workshop, you did this kind of translation with an XML map. XML map functionality is deprecated as of WebLogic Platform 8.1; you should use XQuery maps for future development.

As you might imagine, an XQuery map indicates how data from one XML shape maps to the parts of another shape by using XQuery expressions. XQuery is a powerful syntax for specifying particular parts of an XML instance. At the simple end, XQuery is expressed in a simple, path−like syntax that may select (in the SQL sense) elements of a particular name. At the more complex end, XQuery supports expressions that declare variables, contain loop structures, and construct new XML shapes with the data they retrieve.

Note: Your XQuery map relies on the schema you compile by placing an XSD file in a schema project. Keep in mind that if you change the target namespace for a schema used by an XQuery map, you must also change the web service's xmlns property so that its values correspond to the new schema. Also, you will need to update an yXQuery expressions that use the namespace so that they match as well.

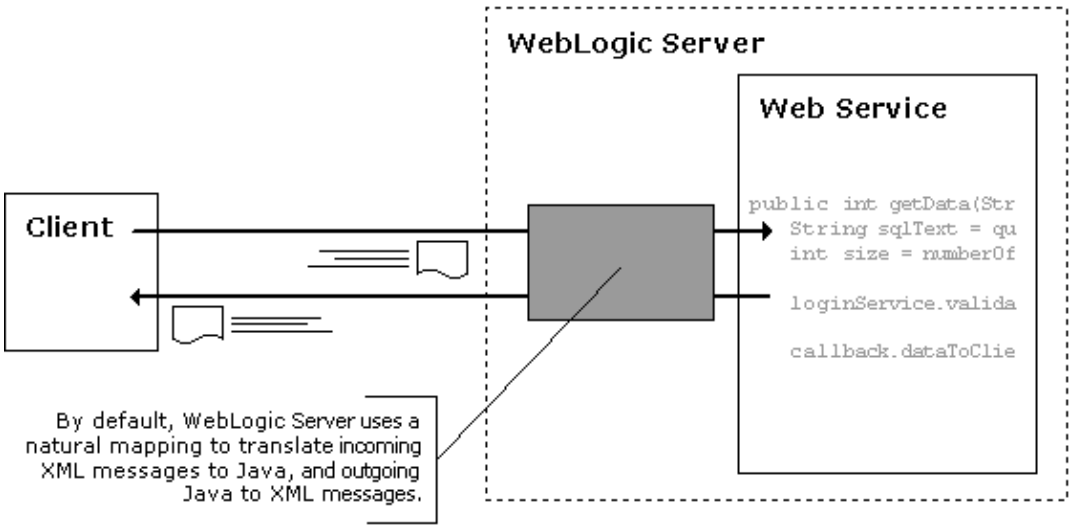
You can get assistance creating an XQuery map by using the XQuery map dialog (although you can also type one manually). For more information, see How Do I: Add or Edit an XQuery Map with the XQuery Mapper Dialog?

When to Use an XQuery Map on a Web Service

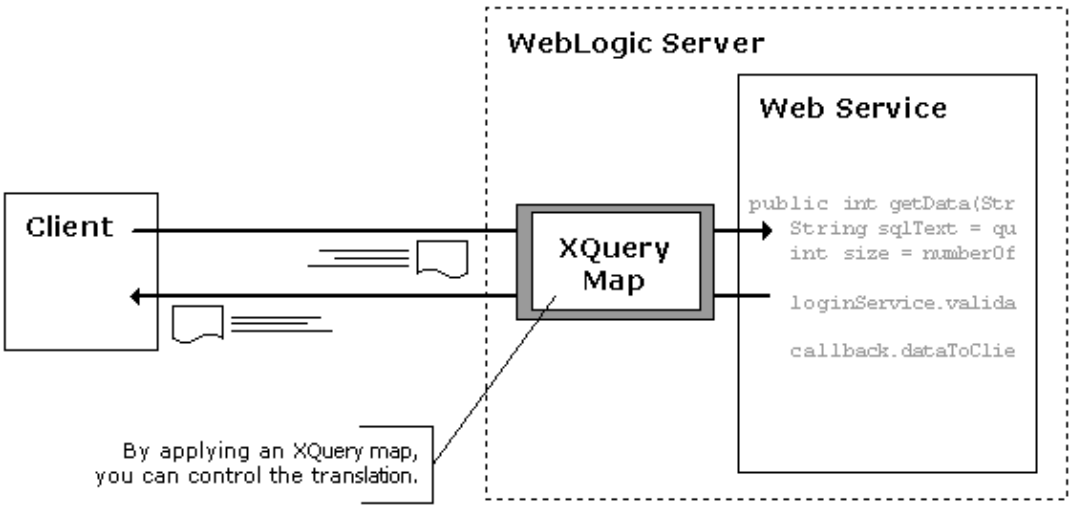
You may want to apply an XQuery map if:

- You want to receive a particular XML shape from clients, but your web service operations don't support it. Through an XQuery map, you can anticipate the desired message shape, but leave your Java operation signature unchanged. The map translates from one to the other.
- You want to provide a layer for loose coupling between your web service and its clients. Again, the XML shape represented by the XQuery map is used for your web service's public face to the world. You can change the underlying Java code without breaking the contract with clients.

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—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—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.

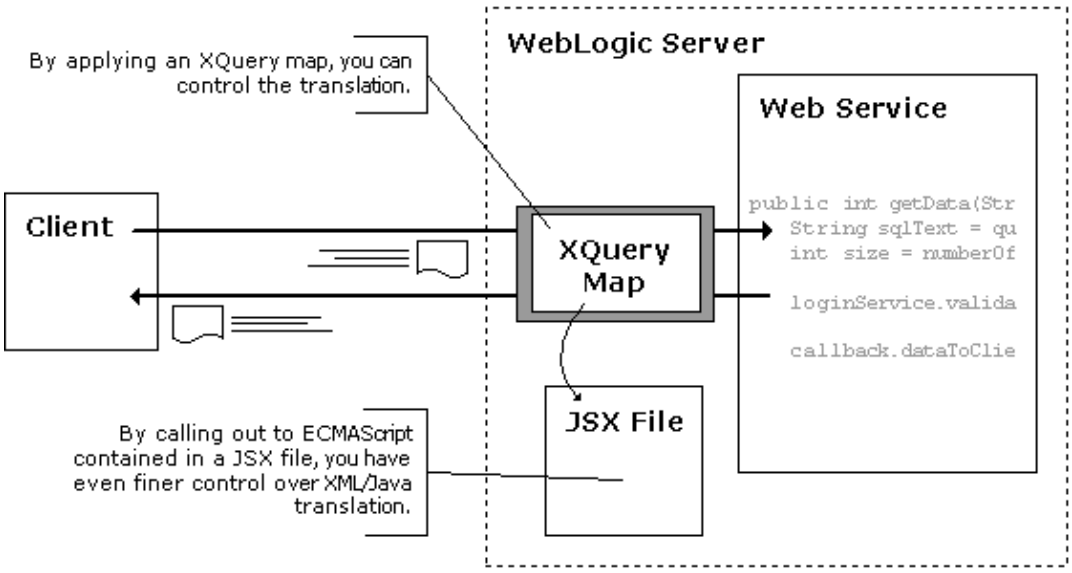
Related Topics

How Do I: Add or Edit an XQuery Map with the XQuery Mapper Dialog?

XQuery Map Samples

Using ECMAScript From XQuery 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.



You store an ECMAScript function for mapping in a JSX file. From your XQuery map you point to the function. A map with a reference to a script function might look like this example:

```
/**
 * @common:operation
 * @jws:parameter-xml schema-element="ns0:emp" xquery::
 * declare namespace ns0="http://openuri.org/test"
 * declare namespace ns1="http://www.openuri.org/"
 * <ns1:submitPerson>
 *     {xqueryMap.TestScript.getPerson($input)}
 * </ns1:submitPerson>
 * ::
 */
public String submitPerson(String personName, int personAge)
```

The text in bold is the call to script. The script file is TestScript (in the xqueryMap folder), and the function being called is getPerson. The $input parameter represents, as it does elsewhere in maps, the XML document; because this is a parameter−xml map on a method, $input represents the incoming XML.

In other words, the incoming XML is passed to the script function. The function is responsible for handling the XML and extracting the portion needed. In this simple example, the function simply constructs the XML that corresponds to the method parameters; it inserts values from the incoming XML into the XML is it constructing, as shown here:

```
function getPerson(xml)
{
    /*
     * Create a variable that contains the XML that contains values to be
     * passed to the method parameters. Use <> and </> to enclose what is
     * actually an XML fragment. Access values from the incoming XML with the
```

```
     * dot operator, enclosing those expressions in {} to indicate
     * where their return values should be inserted.
     */
    var responseXml =
        <>
            <test:personAge xmlns:test='http://www.openuri.org/'>{xml.person.age}</test:personA
            <test:personName xmlns:test='http://www.openuri.org/'>{xml.person.name}</test:perso
        </>;
    return responseXml;
}
```

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:

```
<ns1:submitPerson>
    {xqueryMap.TestScript.getPerson($input)}
    <ns1:personAge>{data($input/ns0:person/ns0:age)}</ns1:personAge>
</ns1:submitPerson>
```

Related Topics

Introduction to XQuery 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.

Converting Java Types in Script with Variant

ECMAScript provides a Variant type that you can use to convert and specify Java types in script.

Removing Elements and Attributes with the delete Operator

You can use the delete operator to remove specified elements and attributes.

Using XMLBeans Types in Script Functions

You can use XMLBeans schema types, such as those generated by compiling schema, in an ECMAScript function.

Using XQuery Within Script Functions

You can write an ECMAScript function that contains XQuery instead of script, then call that function from an XQuery map or Java code.

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.

Using ECMAScript From XQuery Maps

Just as with XML maps you create, ECMAScript you write for mapping acts as a translation layer.

Functions for Manipulating XML

Offers a reference on the functions available with extended ECMAScript.

Creating and Using XML Variables

Introduces the XML and XMLList data types, and shows how you can create XML variables.

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.

Summary of ECMAScript Language Extensions

Lists the ECMAScript operators and functions designed for handling XML.

Related Topics

Handling and Shaping XML Messages with XML Maps

XQuery Map Samples

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">
           <firstname>Armin</firstname>
            <lastname>HooHaw</lastname>
            <age>91</age>
    </employee>;
```

When you want to retrieve all of an element's children, you can use the .* operator. For example, the following code would return the firstname, lastname, and age elements:

```
children = xmlEmployees.employees.employee[0].*;
```

Your code could loop through the returned children with code such as the following, printing the value for each:

```
for (i = 0; i < children.length; i++)
{
    currentChild = children[i];
    if (currentChild.getName() == "firstname")
    {
        print("first name is " + currentChild);
    }
    else if (currentAttr.getName() == "lastname")
    {
        print("last name is " + currentChild);
    }
}
```

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—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];
```

When elements have names that conflict with ECMAScript keywords, you can use the following syntax instead..

```
/*
```

```
 * 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 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" dept="Sales">
        <firstname>John</firstname>
        <lastname>Walton</lastname>
        <age>25</age>
    </employee>
    <employee id="222222222" dept="Human Resources">
        <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;
```

You can also retrieve all of an element's attributes with the @* operator. For example, the following line would return all of the attributes for the first employee element.

```
var attrs = xmlEmployees.employees.employee[0].@*;
```

You could loop through the returned attributes with code such as the following, printing the value for each:

```
for (i = 0; i < attrs.length; i++)
{
    currentAttr = attrs[i];
    if (currentAttr.getName() == "id")
    {
        print("id attribute's value is " + currentAttr);
    }
    else if (currentAttr.getName() == "dept")
    {
        print("dept attribute's value is " + currentAttr);
    }
}
```

Note: You can also access namespaced attributes, as described in Filtering By Namespace.

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—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 variable of the Namespace type to query the XML for elements contained in a specified namespace.

The syntax for declaring a Namespace variable is as follows:

```
var nsVariable = new Namespace("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 = nsVariable::elementName1.nsVariable::elementName2;
```

```
value = ["nsVariable:elementName1"]["nsVariable: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 in the document
 * above. When accessing the XML, the myco variable will correspond to the
 * namespace URI in the way that the "inc" prefix above corresponds to the
 * namespace in the XML instance.
 */
var myco = new Namespace("http://openuri.org/");

/*
 * Create a new variable to hold the <employee> element containing John.
 * Access the elements by using the namespace variable
 * in double-colons in place of the prefix-colon combination used
 * in the XML literal above.
 */
var xmlName = xmlEmps.myco::employees.myco::employee[0];
```

Note that you can also access namespaced attributes in the same way. In other words, you can use @inc::id to access the inc:id attribute in the example, as shown here:

```
/* Create a new variable to hold the <employee> node containing John. */
var xmlName = xmlEmps.myco::employee.attribute(@inc::id);
```

Or like this:

```
var xmlName = xmlEmps.myco::employee.@inc::id;
```

Related Topics

Accessing Attributes With the @ Operator

Accessing Element Children With the . Operator

Inserting Elements With the += Operator

Using the += operator, you can insert an XML element at the end of a list of sibling elements. (Note that this operator is equivalent to using the appendChild function described in Functions for Manipulating XML.)

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 at the end of the list */
xmlEmployees..employee[1] +=
        <employee id = "333333333">
            <firstname>Rob</firstname>
            <lastname>Petrie</lastname>
            <age>34</age>
        </employee>;
```

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> element at the end of the list by omitting a specific reference to a particular element (as in the preceding example). To insert an element 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><firstname>Bruce</firstname><lastname>Wa
```

You might also find the appendChild function useful when combining XML elements. For more information, see appendChild function.

Related Topics

appendChild function

Converting Java Types in Script with Variant

ECMAScript provides a Variant type that you can use to convert and specify Java types in script. In script, this type is typically at work invisibly.

This is particularly useful in cases where your script code makes a call to an overloaded method of a Java class. Unlike Java, ECMAScript is a dynamically typed language. This means that you don't specify a type when you declare script variables, allowing the script interpreter to specify the type at run time.

You can specify particular Java types in script by using a casting syntax. At run time, the interpreter converts variables and values to the Java types you specify. For example, the Java Math class provides four max methods that return the larger of their two arguments. Each max method takes number pairs of different types: double, float, int, or long. In script, to specify the method that takes two int values, you could write the following:

```
var greaterNumber = java.lang.Math.max((int)200, (int)100);
```

When you use casting in script to convert a type, the resulting type is a Variant that is automatically converted when it's used. In other words, the following two lines are equivalent:

```
x = (int)10;
x = new Variant(10).changeType(Variant.INTEGER_TYPE);
```

One of the most useful conversions supported by Variant is between date formats. For example, the following conversions are all supported because Variant is the underlying type when you are casting in script.

```
javaDate = new java.util.Date(currentMilliSeconds);
// Convert to the XmlDate type provided with XMLBeans.
xmlDate = (XmlDate)javaDate;
// Convert to an SQL Date.
sqlDate = (java.sql.Date)javaDate;
```

You can get the Java type by using the Variant class's getTypeName method, as shown here:

```
var someDouble = (double)10.0;

// Print the type's name: "double"
print(someDouble.getTypeName());
```

Methods of the Variant Type

The following lists the methods exposed by Variant instances you create. In keeping with the Variant type's role in helping with conversion, these methods let you convert from one type to another, and to determine what type the Variant variable currently is.

```
public Object getValue ();

public String toString ();
```

Methods for Discovering the Type

```
public int getType ();
```

- Returns one of the constants described later in this topic.

```
public boolean isUndefined ();

public boolean isNull ();

public boolean isEmpty ();
```

- Returns true if this type is null or undefined.

```
public boolean isBoolean ();

public boolean isAnyNumber ();

public boolean isAnyDate ();

public boolean isAnyTime ();

public boolean isPrimitive ();
```

- Returns true if this type is a boolean, byte, short, int, long, double, real, or char

```
public boolean isAnyInteger ();
```

- Returns true if this type is a byte, short, int, or long

```
public boolean isAnyFloatingPoint ();
```

- Retuns true if this type is a double or real

```
public boolean isAnyString ();
```

- Returns true if this type is a char or string

Methods for Converting from One Type to Another

Changing the Current Variable's Type

```
public void changeType (int type) throws ClassNotFoundException;
```

- The *type* parameter should be one of the constants described later in this topic. For example, the following lines declare a String variable, then change its type to an int (with a value of 12345):
  ```
  x = (String)"12345";
  y = x.changeType(Variant.INTEGER_TYPE);
  ```

```
public void changeType (Class newClass) throws ClassNotFoundException;

public void changeType (String newClass) throws ClassNotFoundException;
```

Getting Another Type from the Current Variable

```
public Boolean getAsBoolean ();

public Byte getAsByte ();

public Character getAsCharacter ();

public Double getAsDouble ();

public Float getAsFloat ();

public Integer getAsInteger ();

public Long getAsLong ();

public Number getAsNumber ();

public Object getAsObject ();

public Short getAsShort ();

public String getAsString ();

public Object getAsArray ();

public java.util.Date getAsJavaDate ();

public IJsxObject getAsJsxDate ();

public XmlDate getAsXmlDate ();

public java.sql.Date getAsSQLDate ();

public XmlDateTime getAsXmlDateTime ();

public java.sql.Time getAsSQLTime ();

public IJsxObject getAsJsxObject ();
```

Setting the Current Variable to a New Type and Value

```
public void setBoolean (Boolean value);

public void setByte (Byte value);

public void setCharacter (Character value);

public void setDouble (Double value);

public void setFloat (Float value);

public void setInteger (Integer value);

public void setLong (Long value);
```

```
public void setNumber (Number value);

public void setObject (Object value);

public void setShort (Short value);

public void setString (String value);

public void setArray (Object value);

public void setJsxObject (Scriptable value);

public void setJavaDate (java.util.Date value);

public void setJsxDate (IJsxObject value);

public void setXmlDate (XmlDate value);

public void setSQLDate (java.sql.Date value);

public void setXmlDateTime (XmlDateTime value);

public void setSQLTime (java.sql.Time value);
```

Constants Representing Specific Types

The following constant represent specific types handled by Variant. When you call the getType method to discover a variable's current type, the method returns the int value associated with one of these constants. Likewise, when calling the changeType(int) method, you pass one of these as the parameter to indicate which type to change the variable to.

```
UNDEFINED_TYPE = 0;

NULL_VALUE = 1;

BOOLEAN_TYPE = 2;

BYTE_TYPE = 3;

HARACTER_TYPE = 4;

DOUBLE_TYPE = 5;

FLOAT_TYPE = 6;

INTEGER_TYPE = 7;

LONG_TYPE = 8;

NUMBER_TYPE = 9;

OBJECT_TYPE = 10;

SHORT_TYPE = 11;
```

```
STRING_TYPE = 12;

ARRAY_TYPE = 13;

JAVADATE_TYPE = 14;

JSXDATE_TYPE = 15;

SQLDATE_TYPE = 16;

XMLDATE_TYPE = 17;

JSXOBJECT_TYPE = 18;

XMLOBJECT_ANY = 19;
```

- Represents untyped XML

```
XMLOBJECT_COMPLEX = 20;
```

- Represents a schema (XSD) complex type

```
SQLTIME_TYPE = 21;

XMLDATETIME_TYPE = 22;
```

Related Topics

None.

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

Using XMLBeans Types in Script Functions

You can use XMLBeans schema types, such as those generated by compiling schema, in an ECMAScript function.

You create an instance of an XMLBeans type by casting an XML value, as shown here:

```
var poXml = (PurchaseOrder)<purchase−order xmlns="http://openuri.org/easypo">
    <customer>
        <name>Gladys Kravitz</name>
        <address>Anytown, PA</address>
    </customer>
    <date>2001−12−17T09:30:47−05:00</date>
    <line−item>
        <description>Burnham's Celestial Handbook, Vol 1</description>
        <per−unit−ounces>5</per−unit−ounces>
        <price>21.79</price>
        <quantity>2</quantity>
    </line−item>
    <shipper>
        <name>UPS</name>
        <per−ounce−rate>0.74</per−ounce−rate>
    </shipper>
</purchase−order>;
```

Field−Style Accessors

When you use XMLBeans types in script, the script interpreter supports field−style syntax for accessing child elements. This is an alternative to using the get and set methods exposed by these types. For example, a schema that describes the following XML would provide a PurchaseOrder object you could use to access the purchase−order element's customer, date, line−item, and shipper elements.

In Java code, you would get the value of the customer element with the PurchaseOrder object's getLineItem method. In script, you could do it in this way (note that JavaBeans casing rules apply):

```
var lineItemXml = poXml.lineItem;
var description = lineItemXml.description;
```

You can set XML values with the same field−style syntax.

```
poXml.customer.name = "Darren Stevens";
```

Related Topics

Getting Started with XMLBeans

Importing Java Classes to ECMAScript with the import Statement

Using XQuery Within Script Functions

You can write an ECMAScript function that contains XQuery instead of script, then call that function from an XQuery map or Java code. The function may contain only the XQuery expression.

A script function that contains XQuery must be annotated with a language:body annotation that indicates that the language of the function's body is XQuery. For example, the following function contains an XQuery function that selects the zip elements from the incoming XML, then adds them as child elements of a newly created element, zip−list.

```
/**
 * @language:body type="XQuery"
 */
function selectZipsNewDoc(xml)
{
    declare namespace xq="http://openuri.org/bea/samples/workshop/xmlmap/consolidateAddress"
    declare namespace ns="http://www.openuri.org/"
    let $e := $xml/ns:selectZipsResponse/xq:employees
    return
        <zip−list>
            {for $z in $e/xq:employee/xq:address/xq:zip
            return $z}
        </zip−list>
}
```

Briefly, here's what this function does:

- Takes a single parameter for XML passed in from an XQuery map.
- Declares two namespaces that are used in the XQuery expression. The namespace represented by the xq prefix belongs with the XML containing employee information; the namespace represented by the ns prefix belongs with the XML representing the web service operation (selectZipsResponse).
- Declares an "e" variable to represent the employees root element.
- Constructs a zip−list element for use in the return value.
- Declares a "z" variable to represent the zip elements, looping through and returning the zip elements as children of the zip−list element.

An XQuery map that calls this function might look like this:

```
/**
 * @common:operation
 * @jws:return−xml xquery::
 *   declare namespace ns0="http://www.openuri.org/"
 *   declare namespace ns1="http://openuri.org/bea/samples/workshop/xmlmap/consolidateAddress"
 *   <ns1:employees>
 *     {xmlBeans.xquery.EmpQueryScripts.selectZipsNewDoc($input)}
 *   </ns1:employees>
 *   ::
 * schema−element="ns0:employees"
 */
public XmlObject selectZips(XmlObject xml)
{
    return xml;
}
```

Notice that the call to the script function (in bold) must be nested within another element. With the map and script, the selectZips method's return value looks like this:

```
<con:employees xmlns:con="http://openuri.org/bea/samples/workshop/xmlmap/consolidateAddress">
    <zip-list>
        <ns0:zip xmlns:ns0="http://openuri.org/bea/samples/workshop/xmlmap/consolidateAddress">
        <ns1:zip xmlns:ns1="http://openuri.org/bea/samples/workshop/xmlmap/consolidateAddress">
        <ns2:zip xmlns:ns2="http://openuri.org/bea/samples/workshop/xmlmap/consolidateAddress">
        <ns3:zip xmlns:ns3="http://openuri.org/bea/samples/workshop/xmlmap/consolidateAddress">
        <ns4:zip xmlns:ns4="http://openuri.org/bea/samples/workshop/xmlmap/consolidateAddress">
        <ns5:zip xmlns:ns5="http://openuri.org/bea/samples/workshop/xmlmap/consolidateAddress">
    </zip-list>
</con:employees>
```

Related Topics

None.

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 Java, 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—children().length < 4—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 ECMAScript.

The functions described in this topic are exposed by the four types provided with ECMAScript: XML, XMLList, Namespace, and XMLAttribute. For those functions that apply to multiple types, the function's specifics may differ slightly for each type. For more information on the four types, see Creating and Using XML Variables.

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];
```

Functions Exposed by ECMAScript Types for XML

Many of the following functions are exposed by multiple types. For more information on the types themselves, see Creating and Using XML Variables.

Function	Description	**Applies To**
xml.appendChild(*newChild*)	Inserts a new child node after the existing children of the XML value.	XML, XMLList
xml.attribute(*attributeName*)	Returns the value of the specified attribute as an XMLAttribute.	XML, XMLList
xml.attributes()	Returns a list of attributes for the specified element as an XMLAttribute array.	XML, XMLList
xml.child(*childIndex*)	Returns the XML at the 0−based ordinal position specified by *childIndex*.	XML, XMLList
xml.childIndex()	Returns the 0−based ordinal position of the XML value within its parent.	XML, XMLList
xml.children()	Returns a list of the element's children.	XML, XMLList
xml.comments([*booleanToLookDeep*])	Returns a list of comments from *xml's* point in the document.	XML, XMLList
xml.copy()	Returns a copy of the specified element.	XML, XMLList
xml.cursor()	Returns an XmlCursor instance for the current element.	XML, XMLList
xml.document()	Returns *xml* as a document.	XML, XMLList
xml.getName()	Return the local name for *xml*.	XML, XMLAttribute
xml.getValue()	Returns the value (content) of *xml*.	XML, XMLList, XMLAttribute

xml.innerXML(*newContent*)	Replaces the entire contents of the XML value with new content.	XML, XMLList
xml.isComment()	Returns true if *xml* is an XML comment.	XML, XMLList
xml.isDocument()	Returns true if *xml* represents the XML document; otherwise, false.	XML, XMLList
xml.isProcessingInstruction()	Returns true if *xml* is an XML processing instruction.	XML, XMLList
xml.length	Returns the length of a list XML elements.	XML, XMLList
xml.namespaceURI()	Returns a string representing the namespace URI associated with *xml*.	XML, XMLList, XMLAttribute, Namespace
xml.namespaces()	Returns an array of Namespace objects that represents any xmlns attribute (with or without a prefix) declared in the entire XML document.	XML, XMLList
xml.parent()	Returns the parent of the element.	XML, XMLList, XMLAttribute
xml.prependChild(*newChild*)	Inserts a new child node before the existing children of the XML value.	XML, XMLList
xml.processingInstructions([*piName*, *ignoreSpaces*])	Returns a list of all the XML processing instructions (PIs) from this element's point in the document.	XML, XMLList
xml.setValue(*newValue*)	Sets the value (content) of *xml* to *newValue*.	XML, XMLList
xml.tagName()	Returns the name of the element tag.	XML, XMLList
xml.text()	Returns a string containing the value of all XML properties of *xml* that are of type string.	XML, XMLList
xml.thisXML	Specifies the current XML, such as XML returned from an expression.	XML, XMLList
xml.toString()	Returns the element and its content as a string.	XML, XMLList, XMLAttribute, Namespace
xml.toXMLString()	Returns an XML encoded string representation of *xml*.	XML, XMLList
xml.xpath(*xPathExpression*)	Evaluates the XPath expression using the XML value as the context node.	XML, XMLList

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>;
```

xml.appendChild(*newChild*)

Inserts a new child element (specified by *newChild*) after the existing children of *xml*.

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>);
```

xml.attribute(*attributeName*)

Returns the value of *attributeName* as an XMLAttribute.

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.

xml.attributes()

Returns a list of attributes for *xml* as an XMLAttribute array.

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();
```

xml.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);
```

xml.childIndex()

Returns the ordinal position (within its parent) of *xml*.

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();
```

xml.children()

Returns an XMLList containing all of the children of *xml*.

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();
```

xml.comments([*booleanToLookDeep*])

Returns a list of comments from *xml's* point in the document.

This method can take an optional argument (true or false) signaling to look deep. If no argument is specified, or if false is specified, only comments which are a sibling to *xml* are returned. Otherwise, all comments from this point down the entire depth of the tree are returned.

xml.copy()

Returns a deep copy of *xml*.

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;
```

xml.cursor()

Returns an XmlCursor instance for *xml*.

For more information on the XmlCursor interface, see Navigating XML with Cursors.

xml.document()

Returns *xml* as a document.

You can think of a "document" as the containing entity for *xml*. The document function is especially useful when your script is executing with the literalIsDocument environment variable set to false (the default).

For example, you could insert a new top−level element to the XML above with code such as this:

```
xmlEmployees.document().volunteers = <volunteer/>;
```

The result would be the following:

```
<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>
<volunteers>
    <volunteer/>
</volunteers>
```

xml.getName()

Return the local name for *xml*.

xml.getValue()

Returns the value (content) of *xml*.

xml.isComment()

Returns true if *xml* is an XML comment.

xml.isDocument()

Returns true if *xml* represents the XML document; otherwise, false.

When working with XML in script, you can specify an option to indicate whether the XML variable should represent the XML document or the XML's root element. That environment variable is literalIsDocument, as described in Setting Environment Attributes for XML in ECMAScript.

For example, consider the following assignment:

```
var myXml = <name><first>Joe</first><last>Schmoe</last></name>;
```

With literalIsDocument set to false (the default), the code would look like this:

```
var firstName = myXml.first;
```

With literalIsDocument set to true, code to retrieve the first name might look like this:

```
var firstName = myXml.name.first;
```

The following would also return true:

```
myXml.isDocument();
```

xml.innerXML(*newContent*)

Replaces the entire contents of *xml* 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></>);
```

xml.isProcessingInstruction()

Returns true if *xml* is an XML processing instruction.

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;
```

xml.namespaces()

Returns an array of Namespace objects that represents any xmlns attribute (with or without a prefix) declared in the entire XML document.

xml.namespaceURI()

Returns a string representing the namespace URI associated with *xml*.

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();
```

xml.parent()

Returns the parent of *xml*.

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();
```

xml.prependChild(*newChild*)

Inserts *newChild* before the existing children of *xml*.

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>);
```

xml.processingInstructions([*booleanToLookDeep*])

Returns a list of all the XML processing instructions (PIs) from this element's point in the document.

This method can take an optional argument (true or false) signaling to look deep. If no argument is specified, or if false is specified, only PIs which are a sibling to the current element are returned. Otherwise, all PIs from this point down the entire depth of the tree are returned.

xml.setValue(*newValue*)

Sets the value (content) of *xml* to *newValue*.

xml.tagName()

Returns the name of the tag for *xml*.

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();
```

xml.thisXML

Specifies the current XML, such as XML returned from an expression.

For more information, see Specifying the Current XML with the thisXML Property.

xml.text()

Returns a string containing the value of all XML properties of *xml* 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();
```

xml.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();
```

xml.toXMLString()

Returns an XML encoded string representation of *xml*.

The following example returns a string representation of the first <employee> element.

```
var johnText = xmlEmployees..employee[0].toXMLString();
```

xml.xpath(*xPathExpression*)

Evaluates the XPath expression (specified by *xpathExpression*) using *xml* 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

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 Object

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—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 Object

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 XML and XMLList

- You can specify whether the variable name corresponds to the container for the XML, or to the XML's root.

  As of WebLogic Platform 8.1, an XML variable represents the root of the XML assigned to it. Compare the following two lines of code, which could be used to access the XML above.

  ```
  /* This is how it's done by default. */
  var johnFirstName = myXML.employee.firstname;

  /*
   * You can use an environment variable to specify that access
   * should work this way.
   */
  var johnFirstName = myXML.employees.employee.firstname;
  ```
- The script interpreter does not support literal XML values that do not contain at least one element.

Namespace

You declare a namespace variable when you want to refer to a namespace in code that accesses XML. For example, you can declare a namespace variable in this way:

```
var myNS = new Namespace("http://openuri.org/bea/examples");
```

This code associates the myNS variable with the namespace URI. Code to access XML belonging to this namespace could look like this:

```
var firstName = myXml.myNS::employee.myNS::firstname;
```

For more information, see Filtering By Namespace. The Namespace class also provides two methods you might find useful, as described in Functions for Manipulating XML.

XMLAttribute

When your code retrieve an attribute, the type returned is an XMLAttribute. For the most part, you use the XMLAttribute type implicitly, rather than by declaring it.

However, the XMLAttribute class provides several methods that you might find useful, as described in Functions for Manipulating XML.

Related Topics

Functions for Manipulating XML

Accessing Element Children With the . Operator

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—Specifies the number of spaces each child of your XML will be indented when converting the value to a string. This includes when printing to the console, when calling toString or toXMLString.

Usage: XML.environment.@prettyIndent = 4;

Default: 2.

prettyPrint—Specifies whether the your XML should be arranged with indents and line breaks when converting the value to a string. This includes when printing to the console, when calling toString or toXMLString.

Usage: XML.environment.@prettyPrint = false;

Default: true.

ignoreWhitespace—Specifies whether whitespace characters at the beginning or end of any textnode should be ignored when the XML is being parsed by the script interpreter.

Usage: XML.environment.@ignoreWhitespace = false;

Default: true.

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 is less than or equal to 0x20 (including control characters)

ignoreComments—Specifies whether XML comments should be ignored when the XML is being parsed.

Usage: XML.environment.@ignoreComments = false;

Default: true.

ignoreProcessingInstructions—Specifies whether XML processing instructions should be ignored when the XML is being parsed.

Usage: XML.environment.@ignoreProcessingInstructions = false;

Default: true.

literalIsDocument—Specifies whether the variable to which XML is bound represents the XML document or the XML's root.

Usage: XML.environment.@literalIsDocument = true;

Default: false.

For example, consider the following assignment:

```
var myXml = <name><first>Joe</first><last>Schmoe</last></name>;
```

With literalIsDocument set to true, code to retrieve the first name might look like this:

```
var firstName = myXml.name.first;
```

With literalIsDocument set to false (the default), the code would look like this:

```
var firstName = myXml.first;
```

Related Topics

None.

Type Support in ECMAScript

When you add an XQuery map that references an ECMAScript function, data is converted between native ECMAScript types and native Java types. For example, consider an XQuery 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 while mapping an incoming XML message to Java.

Note that the other case—from Java to XML—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 ECMAScript. There are no limitations when there is no type conversion, which means 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 type: 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 types: 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 arrays: 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

None.

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 any one of the attributes belonging to the element on the left side of the operator. See Accessing Attributes With the @ Operator.
@*	Provides access to all 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 any one of immediate child elements contained by the element on left side of the operator. See Accessing Element Children With the . Operator.
.*	Provides access to all 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.
<> and </>	Enclose XML that has no root, making it possible to assign the enclosed XML to a variable even though it is not well−formed. See Creating and Using XML Variables.
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	DEPRECATED: Use the Namespace type instead. 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	**Applies To**
xml.appendChild(*newChild*)	Inserts a new child node after the existing children of the XML value.	XML, XMLList
xml.attribute(*attributeName*)	Returns the value of the specified attribute as an XMLAttribute.	XML, XMLList
xml.attributes()	Returns a list of attributes for the specified element as an XMLAttribute array.	XML, XMLList
xml.child(*childIndex*)	Returns the XML at the 0−based ordinal position specified by *childIndex*.	XML, XMLList
xml.childIndex()	Returns the 0−based ordinal position of the XML value within its parent.	XML, XMLList
xml.children()	Returns a list of the element's children.	XML, XMLList
xml.comments([*booleanToLookDeep*])	Returns a list of comments from *xml's* point in the document.	XML, XMLList
xml.copy()	Returns a copy of the specified element.	XML, XMLList
xml.cursor()	Returns an XmlCursor instance for the current element.	XML, XMLList
xml.document()	Returns *xml* as a document.	XML, XMLList
xml.getName()	Return the local name for *xml*.	XML, XMLAttribute
xml.getValue()	Returns the value (content) of *xml*.	XML, XMLList, XMLAttribute
xml.innerXML(*newContent*)	Replaces the entire contents of the XML value with new content.	XML, XMLList
xml.isComment()	Returns true if *xml* is an XML comment.	XML, XMLList
xml.isDocument()	Returns true if *xml* represents the XML document; otherwise, false.	XML, XMLList
xml.isProcessingInstruction()	Returns true if *xml* is an XML processing instruction.	XML, XMLList
xml.length	Returns the length of a list XML elements.	XML, XMLList
xml.namespaceURI()	Returns a string representing the namespace URI associated with *xml*.	XML, XMLList, XMLAttribute, Namespace
xml.namespaces()	Returns an array of Namespace objects	XML, XMLList

	that represents any xmlns attribute (with or without a prefix) declared in the entire XML document.	
xml.parent()	Returns the parent of the element.	XML, XMLList, XMLAttribute
xml.prependChild(*newChild*)	Inserts a new child node before the existing children of the XML value.	XML, XMLList
xml.processingInstructions([*piName*])	Returns a list of all the XML processing instructions (PIs) from this element's point in the document.	XML, XMLList
xml.setValue(*newValue*)	Sets the value (content) of *xml* to *newValue*.	XML, XMLList
xml.tagName()	Returns the name of the element tag.	XML, XMLList
xml.text()	Returns a string containing the value of all XML properties of *xml* that are of type string.	XML, XMLList
xml.thisXML	Specifies the current XML, such as XML returned from an expression.	XML, XMLList
xml.toString()	Returns the element and its content as a string.	XML, XMLList, XMLAttribute, Namespace
xml.toXMLString()	Returns an XML encoded string representation of *xml*.	XML, XMLList
xml.xpath(*xPathExpression*)	Evaluates the XPath expression using the XML value as the context node.	XML, XMLList

Related Topics

Handling XML with ECMAScript Extensions

Deprecated. *XML Maps are deprecated as of the WebLogic Platform 8.1 release. For new code, use XQuery maps. For more information, see Introduction to XQuery Maps.*

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—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.

Related Topics

@jws:parameter−xml Tag

@jws:return−xml Tag

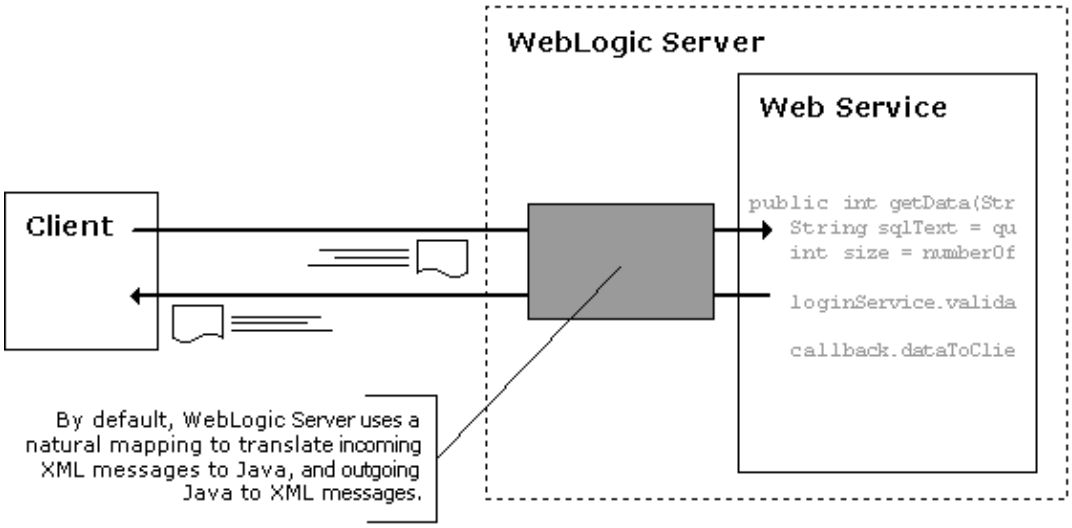
XML Map Tag Reference

Introduction to XML

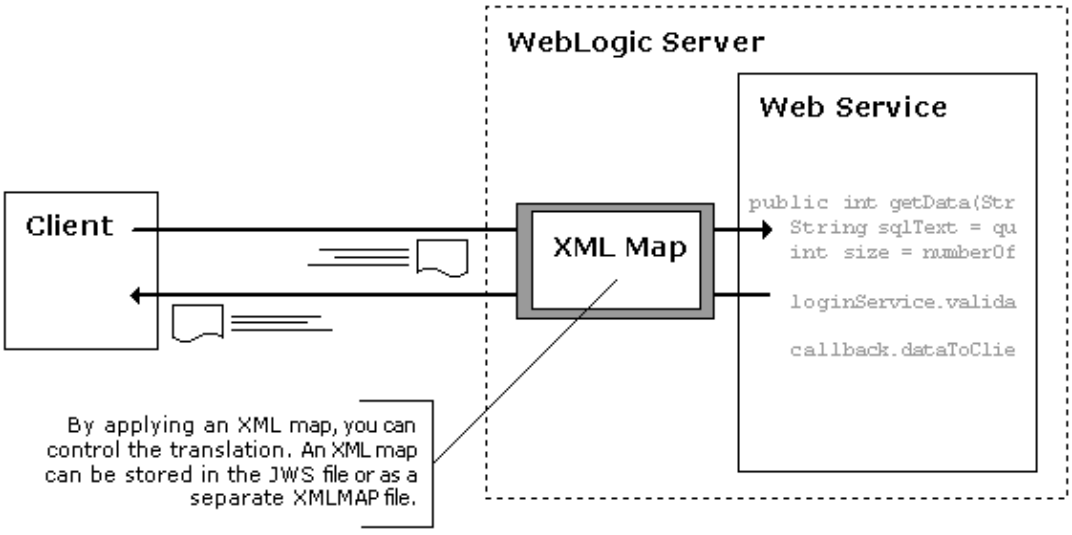
Deprecated. *XML Maps are deprecated as of the WebLogic Platform 8.1 release. For new code, use XQuery maps. For more information, see Introduction to XQuery Maps.*

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—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—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.

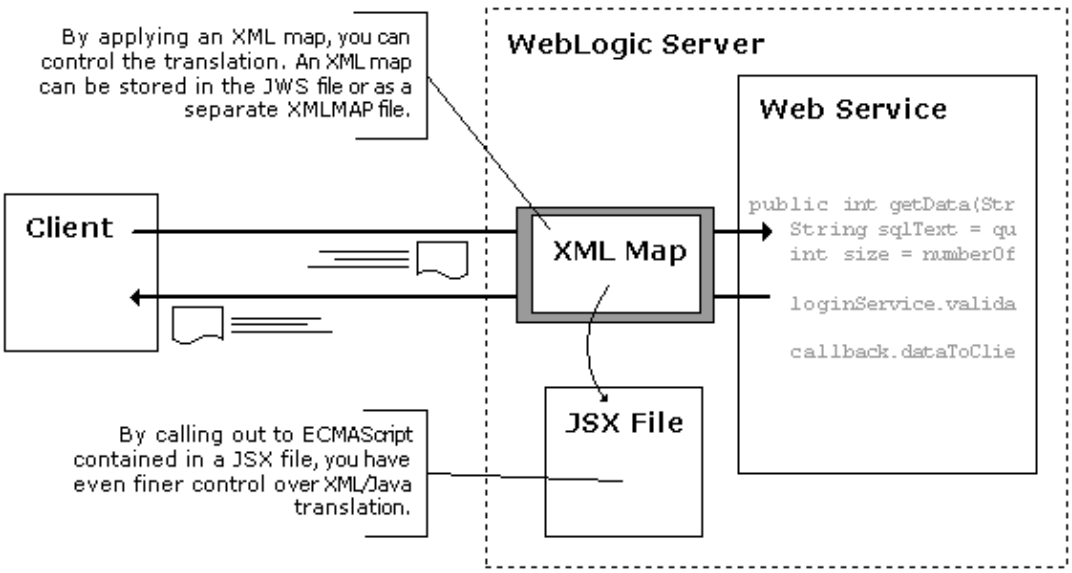
More specifically, through XML maps you can:

- 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.



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—or an example—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

Deprecated. *XML Maps are deprecated as of the WebLogic Platform 8.1 release. For new code, use XQuery maps. For more information, see Introduction to XQuery 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.

Where Can XML Maps Be Used?

Describes on which methods and callbacks XML maps can be used.

Applying parameter−xml and return−xml Maps

Describes when to use these maps and their syntax.

Type Support in XML Maps

Lists the Java types supported by XML maps and gives examples of how they are translated into schema types.

Creating Reusable XML Maps

Introduces XMLMAP files, which contain standalone XML maps.

Related Topics

Getting Started with Script for Mapping

How Do I: Add or Edit an XML Map with the Edit Maps and Interface Dialog?

Handling and Shaping XML Messages with XML Maps

Deprecated. *XML Maps are deprecated as of the WebLogic Platform 8.1 release. For new code, use XQuery maps. For more information, see Introduction to XQuery 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 JCX 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—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.

```
/**
 * @common: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—<searchRequest>, <productName>, serialNumber, <return>, and so on—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:

```
/**
 * @common: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 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.

Maps in Source Code and Maps 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

Deprecated. *XML Maps are deprecated as of the WebLogic Platform 8.1 release. For new code, use XQuery maps. For more information, see Introduction to XQuery Maps.*

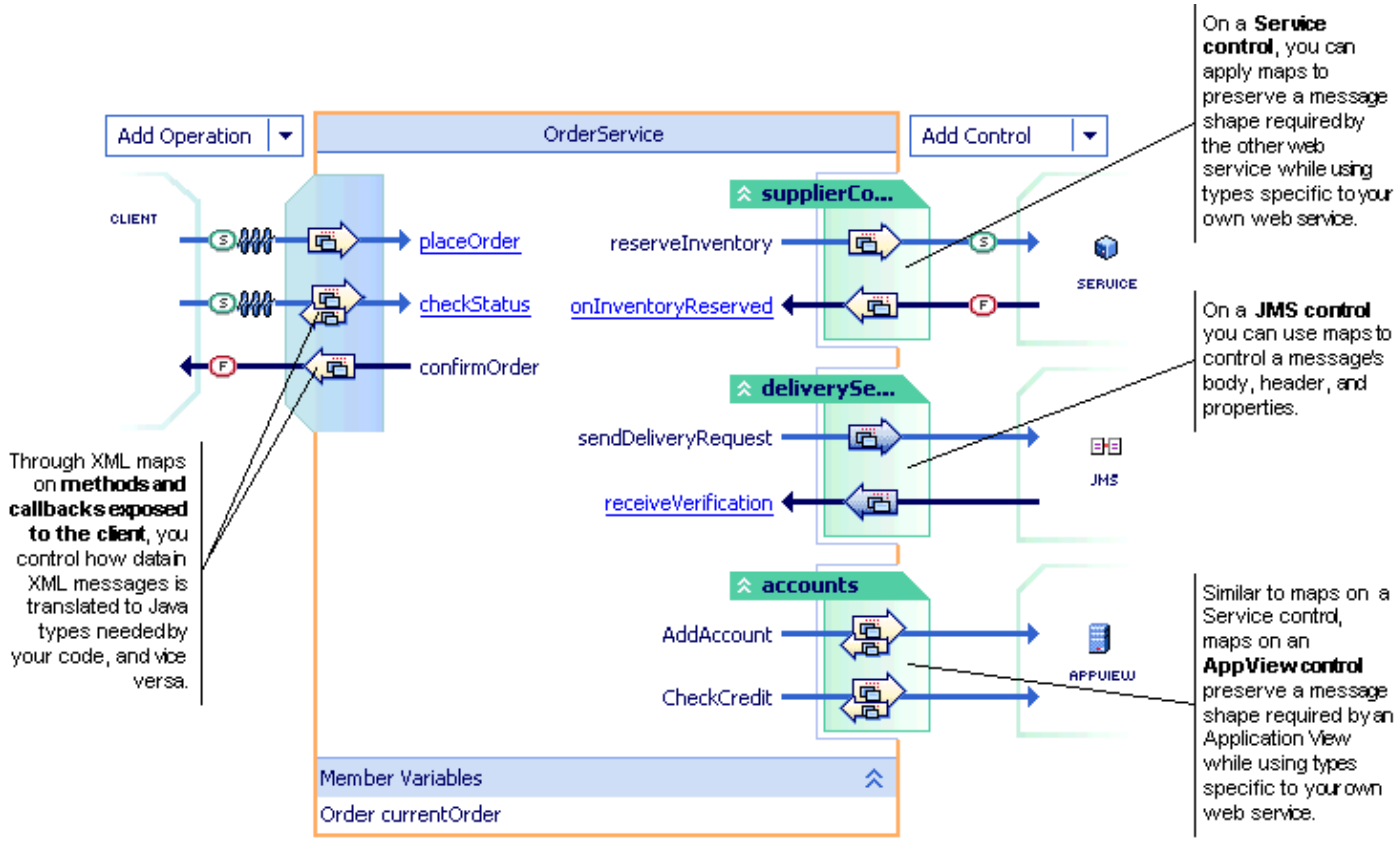
Where Can XML Maps Be Used?

You can apply XML maps to a method or callback exposed to clients, as well as to the methods and callbacks of a Web Service control, a JMS control, and an Application View control.

In all cases, XML maps are a way to ensure loose coupling between your web service and the client or resource with which it is communicating. While topics in this section (beginning with Handling and Shaping XML Messages with XML Maps) describe maps primarily from the client−facing perspective, nearly all of the principles mentioned apply also in the case of maps on controls. For links to more specific information on using maps with controls, use the following links:

- Manipulating JMS Message Headers and Message Properties in a JMS Control
- Defining Java to XML Translation with XML Maps

  Note that XML maps works with the AppView control just as they do with the Web Service control.



Related Topics

Why Use XML Maps?

Deprecated. *XML Maps are deprecated as of the WebLogic Platform 8.1 release. For new code, use XQuery maps. For more information, see Introduction to XQuery Maps.*

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 (JCX 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 Annotation and @jws:return−xml Annotation.

```
/**
 * @common: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.

Related Topics

How Do XML Maps Work?

How Do I: Add or Edit an XML Map with the Edit Maps and Interface Dialog?

Deprecated. *XML Maps are deprecated as of the WebLogic Platform 8.1 release. For new code, use XQuery maps. For more information, see Introduction to XQuery Maps.*

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 type (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:

- String
- boolean
- byte
- short
- int
- long
- Single−precision float
- Double−precision float
- Date
- Typed arrays
- Lists
- Structure
- JavaBeans with public get and set method
- XML element
- XML document fragment

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>250</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" ty
    </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 xmlns:xsd="http://www.w3.org/2001/XMLSchema" xsi:type="xsd:string">second<
    <anyType xmlns:xsd="http://www.w3.org/2001/XMLSchema" xsi:type="xsd:int">71</anyTy
</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"
    </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: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:stri
        <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

***Deprecated**. XML Maps are deprecated as of the WebLogic Platform 8.1 release. For new code, use XQuery maps. For more information, see Introduction to XQuery Maps.*

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—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.

```
/*

* @common: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>

Deprecated. *XML Maps are deprecated as of the WebLogic Platform 8.1 release. For new code, use XQuery maps. For more information, see Introduction to XQuery Maps.*

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.

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.

Namespaces in XML Maps

Shows how to declare a namespace in an XML map.

Related Topics

Why Use XML Maps?

How Do XML Maps Work?

Deprecated. *XML Maps are deprecated as of the WebLogic Platform 8.1 release. For new code, use XQuery maps. For more information, see Introduction to XQuery Maps.*

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.

```
/**
 * @common: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

Deprecated. *XML Maps are deprecated as of the WebLogic Platform 8.1 release. For new code, use XQuery maps. For more information, see Introduction to XQuery Maps.*

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—perhaps choosing from among them iteratively—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.

```
/**
 * @common: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 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.

```
/**
 * @common: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

Deprecated. *XML Maps are deprecated as of the WebLogic Platform 8.1 release. For new code, use XQuery maps. For more information, see Introduction to XQuery Maps.*

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(){
          return title;
      }
      public void setTitle(String newTitle){
          title = newTitle;
      }
      public boolean isInPrint
      public setInPrint(boolean f)
  }
  ```

Related Topics

Matching XML Shapes

Database Control

Making Simple Substitutions Using Curly Braces

Deprecated. *XML Maps are deprecated as of the WebLogic Platform 8.1 release. For new code, use XQuery maps. For more information, see Introduction to XQuery Maps.*

Accessing Data Structures and Fields in Return Values

When you are creating a return−xml map with a return type that contains structured data (such as an array), public fields, or 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 you use the word "return" to indicate that the return value should be used. This example parses the data members of an object to individual XML elements.

```
/*
 * @common: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
     *  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

Making Simple Substitutions Using Curly Braces

Deprecated. *XML Maps are deprecated as of the WebLogic Platform 8.1 release. For new code, use XQuery maps. For more information, see Introduction to XQuery Maps.*

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 member of the customerData structure. Because it is declared in the <address> element, the new variable is available to the <street> and <zip> elements, which are its children.

```
/**
 * @common:operation
 * @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);
}
```

For reference information on <xm:bind>, see <xm:bind> Attribute.

Related Topics

Matching XML Shapes

Binding to Java Data Structure Members

<xm:bind> Attribute

Deprecated. *XML Maps are deprecated as of the WebLogic Platform 8.1 release. For new code, use XQuery maps. For more information, see Introduction to XQuery Maps.*

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

Deprecated. *XML Maps are deprecated as of the WebLogic Platform 8.1 release. For new code, use XQuery maps. For more information, see Introduction to XQuery Maps.*

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"—<biblio:value> and <xm:value—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.)

```
/*
 * @common: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

Deprecated. *XML Maps are deprecated as of the WebLogic Platform 8.1 release. For new code, use XQuery maps. For more information, see Introduction to XQuery Maps.*

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.

A Few Things to Remember About XML Maps and Script

Lists a few guidelines that may be useful when using XML maps and script.

Related Topics

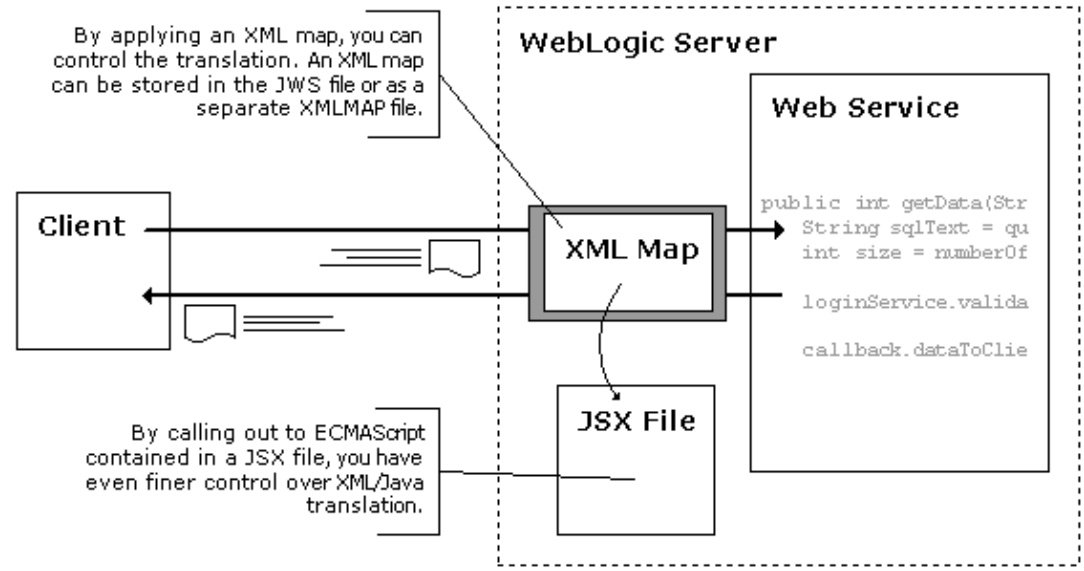
Why Use XML Maps?

Getting Started with XML Maps

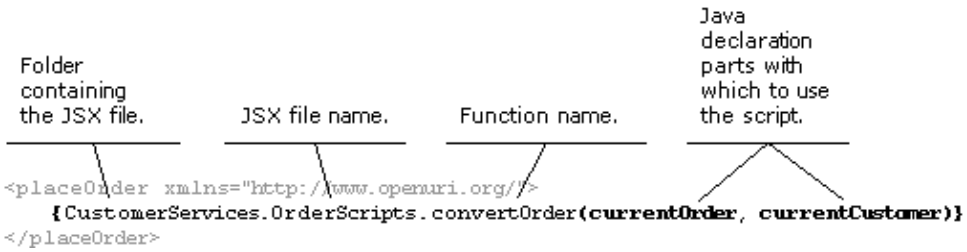
Deprecated. *XML Maps are deprecated as of the WebLogic Platform 8.1 release. For new code, use XQuery maps. For more information, see Introduction to XQuery Maps.*

Using Script Functions From XML Maps

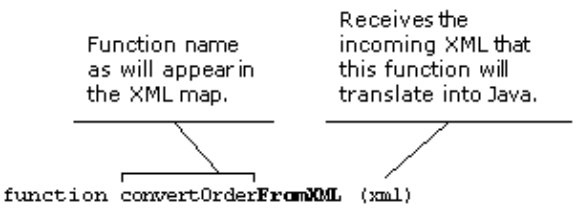
Just as with XML maps you create, ECMAScript you write for mapping acts as a translation layer. For example, in a map designed to translate an incoming XML message to Java, you can include a reference to a script function that is designed to do a calculation, separate or combine pieces of data, and so 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.



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:



A script function for mapping is designed to map an incoming XML message to Java *or* to map Java to outgoing XML—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 convertOrder***FromXML***. At run time, WebLogic Server invokes the function when translating data from XML. Here is what the function's actual declaration looks like:

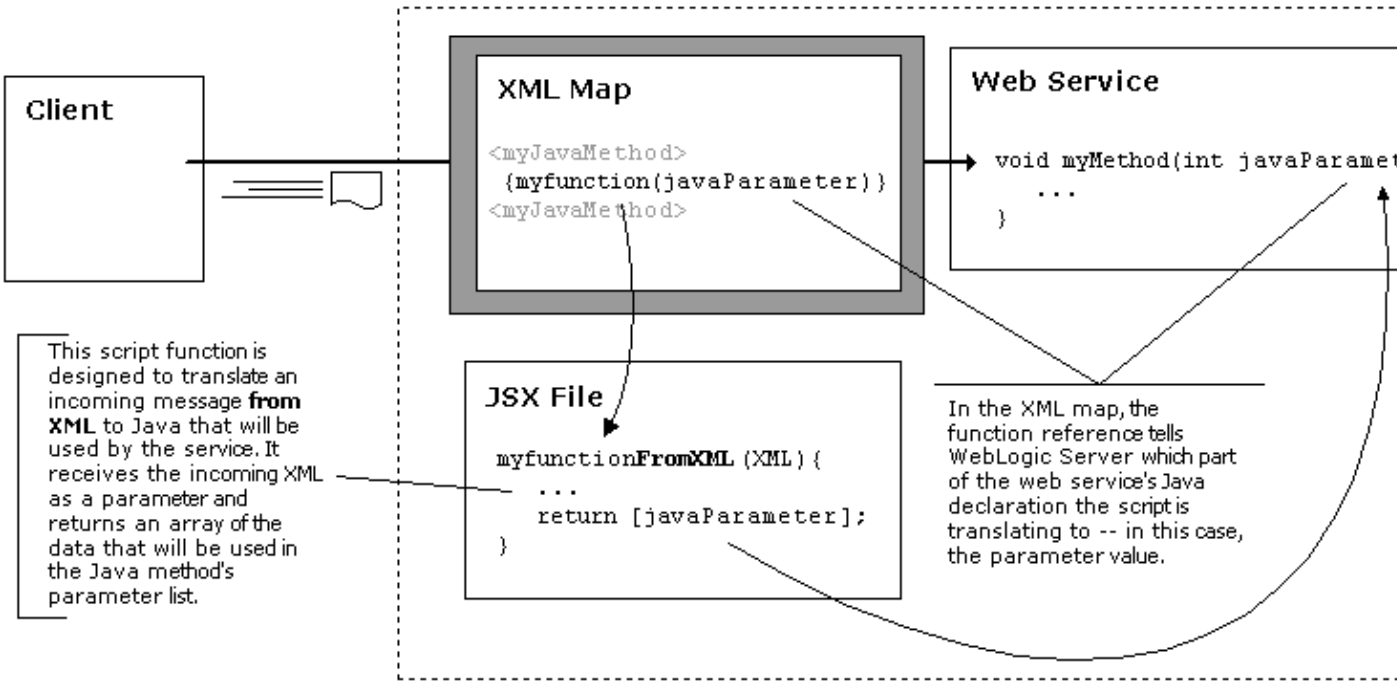


Notice anything unusual? You probably noticed that the single xml parameter in this function declaration doesn't match the items in parenthesis (currentOrder and currentCustomer) in the convertOrder example above. In the example, the items in parenthesis tell WebLogic Server which parts of the Java declaration 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—say, to map it to an outgoing XML message—the item in the parenthesis would simply be return. The function itself is more straightforward:

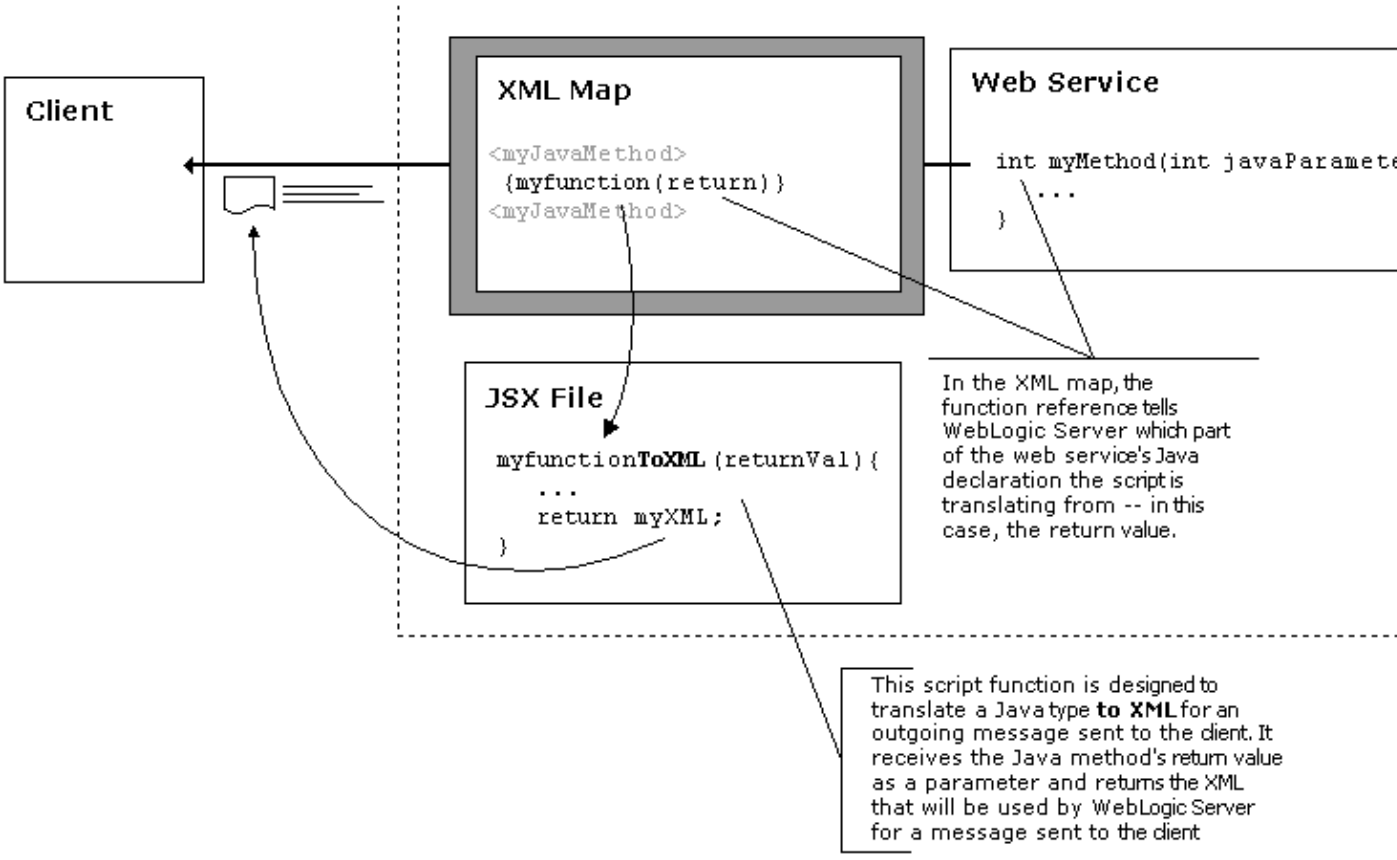
- If the function maps XML to Java, the parameter is XML and the function's return value is an array containing the types to be fed to the Java code.
- If the function maps Java to XML, its parameter is the set of types (one or more) from the Java declaration and its return value is the outgoing XML.

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



Using Script to Map the Return Value on a JWS Method



Related Topics

Why Use XML Maps?

Deprecated. *XML Maps are deprecated as of the WebLogic Platform 8.1 release. For new code, use XQuery maps. For more information, see Introduction to XQuery 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

Keep in mind the following information when you use 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—one is for use with ECMAScript keywords and the other is for use with function names. The following example illustrates both workarounds.

```
/*
 * 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>
```

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 Web 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.
 *
 * @common: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?