BEA WebLogic Workshop™ Help

Version 8.1 SP4
December 2004

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Table of Contents

Designing Asynchronous Interfaces

Interactions between software components can be *synchronous* or *asynchronous*. An interaction is synchronous if the caller of a method must wait for the method's work to complete before the caller can continue its processing. An interaction is asynchronous if the called method returns immediately, allowing the caller to continue its processing without delay. An asynchronous interaction typically initiates a computation but does not wait for the result to be available, which means it must provide some way for the caller to obtain the results of the computation at a later time.

The distributed nature of web applications introduces unpredictable and sometimes very long latencies, which means it may take an operation a long time to complete. If a business process executing over the network involves human interaction at the back end, an operation can take on the order of days. If all interactions over the web were synchronous, clients with pending operations could consume resources on their host systems for unacceptably long periods of time.

WebLogic Workshop provides tools that make it easy for you to build *asynchronous* web services and Java controls that don't require clients to block execution while waiting for results. WebLogic Workshop provides multiple approaches for returning results to your web services' and Java controls' clients; you can choose the one that best suits each situation.

This section describes how to build asynchronous interfaces.

Topics Included in This Section

Getting Started: Using Asynchrony to Enable Long−Running Operations

Describes when and how to build asynchronous web services and Java controls by using callbacks and buffering.

Designing Conversational Web Services

Discusses how to maintain state for an asynchronous web service using conversations.

Best Practices for Designing Asynchronous Interfaces

Describes a number of design principles to consider when building web services and Java controls.

Related Topics

Building Web Services

Working with Java Controls

How Do I...

Samples

Getting Started: Using Asynchrony to Enable Long−Running Operations

WebLogic Workshop provides tools that make it easy for you to build asynchronous web services and asynchronous Java controls. This section introduces the concept of a *callback*, which is one of the critical design components when building an asynchronous web service or Java control, and describes the requirements for using callbacks. In addition, this section describes how to add buffering to methods and callbacks to enable the handling of high−volume traffic.

Topics Included in This Section

Introduction to Asynchronous Interfaces

Introduces asynchronous interfaces in web services and Java controls, and explains when and how to implement them.

Introduction to Asynchronous Java Controls

Introduces special issues surrounding use of asynchrony in Java controls.

Using Callbacks to Notify Clients of Events

Describes how to use callbacks to notify a client of your web service or Java control that some event has occurred.

Using Buffering to Create Asynchronous Methods and Callbacks

Explains how buffering methods and callbacks can help your web services and Java controls handle high−volume loads.

Samples

Conversation.jws

Related Topics

Designing Conversational Web Services

Best Practices for Designing Asynchronous Interfaces

Introduction to Asynchronous Interfaces

Web applications, including web services, typically use the Hypertext Transport Protocol (HTTP) to provide communication between a client and a server (the application). HTTP is a *request−response protocol*. In a request−response protocol, each operation consists of a request message sent from the client to a server followed by a response message returned from the server to the client. The server must always send a response for the operation to complete successfully. Such requests are called *synchronous* because during the request the client is synchronized with the server; the client cannot continue processing until the server responds or the request times out (the client may *time out* if a response is not received within a specific period of time).

In a web application, some of the operations the application performs may be *long−running*. If an operation involves human interaction such as approval by a loan officer of a bank, the operation could take days to complete. It would be a poor design if individual request−response cycles were allowed to span days; such requests would unnecessarily engage resources on both the client and server hosts.

With WebLogic Workshop, you can design your application to be *asynchronous*, which means that the application notifies the client when the response is ready. This allows the client to continue performing other work while the application completes the requested operation. It also keeps each request−response interaction between the client and application as short as possible.

To create an asynchronous web service, you provide one or more methods that accept requests from clients that *begin* an operation but do not wait for the operation to complete. Such methods typically return immediately, supplying the response portion of the initial request−response interaction but *not* supplying the actual result of the requested operation. In an asynchronous interface, you also provide a mechanism for the client to obtain the results of the long−running operation when the results are ready. There are two ways to accomplish this:

- Implement methods that initiate requests and define callbacks to send results.
- Implement methods that initiate requests, methods that return request status (for example, "pending" or "complete") and methods the return results. This approach is referred to as a *polling interface*.

Using Callbacks

When you define a callback for a web service, you are defining a message for the web service to send to the client that notifies the client of an event that has occurred in your web service. In this design, the client first calls the web service with a request. This request call typically returns immediately (completing the first request−response interaction), meaning that the client does not have to wait for the operation to be completed. The client can now continue doing other tasks. When the web service or Java control has finished processing the client's request, it sends a callback, that is, it sends a message back to the client notifying it that the request has been processed and/or providing the results. Note that a callback constitutes a second request−response interaction in which the *request* (not the response) is sent *to the client*. To learn more about the callback mechanism, see Using Callbacks to Notify Clients of Events.

To use a callback, two requirements must be met. First, if a web service defines a callback the web service must be conversational. Conversational web services keep track of the originator of a request and can therefore send the callback to the appropriate caller. Secondly, the client must be capable of receiving and interpreting the callback. If the callback is defined by a web service, then in essence the client must itself be a web service since it must be capable of receiving messages. It must also be capable of correlating an incoming message with a previous request that it initiated. To learn more about conversations, see Designing

Conversational Web Services.

Using Polling

When the client of a web service or Java control is not conversational, as is the case for web pages and non−conversational web services, callbacks cannot be used to notify the client of request completion. In addition, if the client of your web service resides on a host that rejects unsolicited incoming traffic or is protected by firewalls, the host will reject callbacks because callbacks are, by nature, unsolicited, and the client will not receive the callback. To handle these scenarios, web services and Java controls must provide a polling interface. In a polling interface, the client first calls the web service or Java control to that an operation be initiated. This request call is synchronous but typically returns immediately, meaning that the client does not have to wait for the operation to be completed. The client can now continue doing other tasks, but must periodically call the web service or Java control to check the status of its pending request. When a periodic check shows that the request has been completed, the client then calls the web service or Java control to obtain the result. To learn more about implementing a polling interface, see Using Polling as an Alternative to Callbacks.

Polling and callbacks are two different mechanisms to achieve asynchrony. Unless you are absolutely certain that the clients of your web service or Java control will always require only one of these mechanisms, you may want to implement both approaches in order to handle all situations. Doing so provides the convenience of callbacks to those clients who can handle them, and a polling interface for clients who cannot accept callbacks. For more information, see Best Practices for Designing Asynchronous Interfaces.

Buffered Methods and Callbacks

When a conversational web service's request method is called by multiple clients at the same time, a client might have to wait until its call, however short, has been processed by the web service. In addition, when a web service has completed processing multiple requests and needs to send multiple callback messages, it must wait for a message to be processed by the client before it can send the next callback. For a conversational web service, and especially for a web service receiving high−volume traffic, it is recommended to always add buffers to the web service's methods and callbacks. When clients call a buffered method, the call is stored in a buffer and the client does not have to wait for the web service to handle the call. When a web service sends a callback, the message is stored in a buffer and the web service does not have to wait until the client processes the callback. To learn more about asynchronous methods and callbacks, see Using Buffering to Create Asynchronous Methods and Callbacks.

It's important to distinguish between an *asynchronous web service* and an *asynchronous method*. A web service can be designed to be asynchronous without using asynchronous methods. For example, a web service can provide a synchronous request operation that returns an acknowledgement but not the result, and later calls a synchronous callback that sends the result and receives an acknowledgement from the client. Both the method and the callback are synchronous, but the web service is asynchronous.

Related Topics

Building Web Services with WebLogic Workshop

Introduction to Asynchronous Java Controls

Before reading this topic, you should understand the general concepts related to asynchronous interfaces. To learn about asynchronous interfaces, see Introduction to Asynchronous Interfaces.

In WebLogic Workshop, Java controls can have the same asynchronous behavior as web services. However, the behavior of a Java control is dependent on the nature of the Java control's client.

To understand this concept, think about how Java controls are used by their clients: the client is a web service, a page flow or another Java control and the Java control instance is declared as a member variable of the client. This means that the Java control instance cannot have a lifetime that exceeds that of its client.

Using Java Controls from Web Services

WebLogic Workshop web services can be *conversational*. Conversations allow you to control the lifetime of an instance of a web service. Since the lifetime of member variables of a web service is the same as the lifetime of the instance of the web service, Java controls references by a web service also have the same lifetime. Therefore, if a Java control's client is a web service the Java control may freely use callbacks to communicate with its client because the client is guaranteed to exist if the Java control instance exists. Note that the Java control's client must still explicitly implement handlers for each Java control callback it wishes to receive.

To learn more about conversations and WebLogic Workshop web services, see Designing Conversational Web Services.

Using Java Controls from Page Flows

A page flow may also declare a Java control member variable. However, page flows ultimately represent web pages, and web pages operate on a strictly request−response basis from the user's browser. There is no way to "push" information to the browser with a browser−originated request initiated by or on behalf of the user. Since data cannot be "pushed" to the browser, it doesn't make sense to "push" data to a page flow, which is what a callback would represent. Therefore, page flows are not capable of receiving callbacks from a Java control.

However, it is possible to use an intermediary Java control with a polling interface to provide an "adapter" between a page flow and a Java control that uses callbacks. See Example 2, below, but substitute page flow for the non−conversational web service in the example.

With page flows, the limiting issue is not conversational lifetime as it is for web services. The limiting issue with page flows is the inability to "push" information out to the browser. The scenario in Example 2 below hides the "push" (the callback) in Java control A, which presents only a "pull" interface to its client.

To learn more about polling interfaces, see Using Polling as an Alternative to Callbacks.

Using Java Controls from Other Java Controls

Whether callbacks can be sent form one Java control to another depends on the ultimate client at the end of the chain of Java controls. This is probably best illustrated by examples:

Example 1

Assume that you have a conversational web service that uses Java control A, which in turn uses Java control B. The lifetime of the Java control A instance is the same as the lifetime as the web service's conversation, and the lifetime of the Java control B instance is the same as that of Java control A (everything has the same lifetime: that of the web service's conversation). In this situation Java control B may use callbacks to communicate with Java control A, and Java control A may use callbacks to communicate with the web service.

Example 2

Assume the same situation as Example 1, but with a non−conversational web service. Since the web service is non−conversational, the lifetime of an instance of the web service spans only a single invocation of any of the web service's methods. Therefore the lifetime of the web services member variables, including its instance of Java control A, is the same, and the lifetime of the member variables of Java control A, including Java control B, is also the same. Java control A cannot use callbacks to the web service because once the web service method completes Java control A no longer exists. However, if Java control A were to provide a polling interface to be used by the web service, then Java control B may use callbacks to communicate with Java control A. Here is a scenario:

1. A web service client calls a web service method and an instance of the web service is created.
2. An instance of Java control A is created because it is a member variable of the web service.
3. An instance of Java control B is created because it is a member variable of Java control A.
4. The web service method calls the "start request" method of a polling interface provided by Java control A.
5. The "start request" method of Java control A calls a "start request" method of Java control B that will eventually result in a callback to Java control A.
6. The web service method repeatedly calls a "check status" method of Java control A. The method returns false until the result is available.
7. When Java control B's work is completed, it calls a callback of Java control A.
8. The callback handler of Java control A stores the result.
9. The next time the web service calls the "check status" method of Java control A, it returns true.
10. The web service calls a "get result" method on Java control A.
11. The web service returns the result as the return value of the method invoked in step 1.
12. The web service instance is released, causing the release of the instance of Java control A and in turn the release of the instance of Java control B.

Note that step 6 is not advisable if Java control B's work will take a long time. The initial request to the web service in step 1 may time out.

Related Topics

Building Web Services with WebLogic Workshop

Designing Conversational Web Services

Using Polling as an Alternative to Callbacks

Using Callbacks to Notify Clients of Events

Callbacks notify a client (caller) of your web service or Java control that some event has occurred. For example, you can notify a client when the results of that client's request are ready, or when the client’s request cannot be fulfilled.

You can easily add callbacks to the interface of your web service or custom Java control using the Add Callback action in Design View. However, you should be aware that doing so places demands on the client that not all potential clients will be able to meet. This topic describes callbacks and explains some of the things to keep in mind when working with them.

Understanding Callbacks: Messages to the Client

When you add a method to a web service or custom Java control, you implement a message that a client can send to the web service or Java control to perform some operation. When you add a callback to a web service or custom Java control, you are *defining* a message that the web service or Java control can send *to the client* notifying the client of an event that has occurred. So adding a method and defining a callback are completely symmetrical processes, with opposite recipients.

WebLogic Server allows a web service to receive XML and SOAP messages, and routes them automatically to your web service. In order to receive callbacks, the calling client must be operating in an environment that provides the same services. This typically means the client is a web service running on a web server. If the client does not meet these requirements, it is likely not capable of receiving callbacks from your web service. The same requirement applies for Java controls that may use XML and SOAP messaging, such as a web service control, and Java controls whose Java method and callbacks are triggered directly.

The notion that callbacks are messages to the client is important if you want to apply XML maps to callback parameters or return values. The parameters to a callback go into the outgoing message to the client, and the return value is converted from the resulting incoming message from the client. This can seem strange because programmers typically associate all parameters with incoming data and return values with outgoing data.

Maintaining Message Formats

For web services, as well as Java controls using XML messaging, the protocol and message format used for callbacks is always the same as the protocol and message format used by the conversation start method that initiated the current conversation. If you attempt to override the protocol or message format of a callback, an error is thrown.

Related Topics

Introduction to Asynchronous Interfaces

Designing Conversational Web Services

Using Polling as an Alternative to Callbacks

Using Buffering to Create Asynchronous Methods and Callbacks

When a conversational web service's request method is called by multiple clients at the same time, a client might have to wait until its call, however short, has been processed by the web service. In addition, when a web service has completed processing multiple requests and needs to send multiple callback messages, it must wait for a message to be processed by the client before it can send the next callback.

For a conversational web service, and especially for a web service receiving high−volume traffic, it is recommended to always add buffers to the web service's methods and callbacks. When clients call a buffered method, the call is stored in a buffer and the client does not have to wait for the web service to handle the call. When a web service sends a callback, the message is stored in a buffer and the web service does not have to wait until the client processes the callback. You can add a message buffer to any method or callback, provided that the method or callback's parameters are serializable and the return type is void. You can also specify when and whether to retry an invocation of the buffer if it fails.

This topic explains how to add a message buffer to a method or callback of your web service or Java control, how to use message buffers with methods and callbacks, how to make parameters serializable, and how to control retry behavior.

Adding a Message Buffer

You can only add message buffers to methods and callbacks whose parameters are serializable and whose return type is void. When a method or callback with a message buffer is invoked, the message representing the method or callback invocation is placed in a queue to be delivered to the web service, Java control, or client when it is available. Since the queue has no information about the semantics of the message or the destination software component, it cannot return anything meaningful to the caller. And since the queue must always return void, any methods or callbacks that are buffered must return void.

Although you can add message buffers to the methods and callbacks of any web service, it is important to understand what happens when you add message buffers. The queues that provide the buffering always exist on your end of the wire. WebLogic Server has no way to cause configuration actions like message buffering in a remote client's infrastructure. The representation of a web service in Design View reinforces this by drawing the message buffer "spring" icons on the end of the wire closest to the web service. For Java controls, you add message buffers to the methods as you design the Java control, but you add message buffers to the callbacks when you add an instance of this control to a web service (or another Java control), reflecting that you define in the web service how to handle a Java control's callback.

To Add a Message Buffer to a Web Service

1. In the Design View of the Web Service, select the method or callback you want to buffer.
2. In the ***Properties*** pane, navigate to the ***message−buffer*** property.
3. Set the ***enabled*** attribute to true. WebLogic Workshop adds the Javadoc tag @common:message−buffer enabled−true to the Javadoc comment preceding the method or callback in the source code.

   Note: In previous releases, the @common:message−buffer tag was known as the @jws:message−buffer tag. This tag is still supported for backward compatibility.

To Add a Message Buffer to a Java Control's Method

1. In the Design View of the Java control, select the method you want to buffer.
2. In the ***Properties*** pane, navigate to the ***message−buffer*** property.
3. Set the ***enabled*** attribute to true. WebLogic Workshop adds the Javadoc tag @common:message−buffer enabled−true to the Javadoc comment preceding the method or callback in the source code.

To Add a Message Buffer to a Java Control's Callback

1. In the Design View of the Web Service, click the name of the Java control callback you want to buffer. WebLogic Workshop goes to the Web Service's code in Source View that implements the callback.
2. In the ***Properties*** pane, navigate to the ***message−buffer*** property.
3. Set the enabled attribute to true. WebLogic Workshop adds the Javadoc tag @common:message−buffer enabled−true to the Javadoc comment preceding the method or callback in the source code of the Web Service.

Using Message Buffers on Methods

If you add a message buffer to a method, incoming messages (that is, invocations of the method) are buffered on the local machine. The method invocation returns to the client immediately. This prevents the client from waiting until your web service or Java control processes the request. Because the buffering occurs on your end of the wire, the client still must wait for the network roundtrip even though it will return a void result. But the client does not have to wait for your web service or Java control to process the message.

Determining the Execution Order of Methods

When a web service or Java control defines buffered (asynchronous) methods, neither the designer nor the client can make any assumptions about the order in which methods should be executed.

This is especially true when defining both buffered and non−buffered methods in the same web service or Java control. Invocations of synchronous methods are guaranteed to be in order of arrival. But if buffered methods are also defined, you cannot determine when their invocations will occur relative to the synchronous methods.

Using Message Buffers on Callbacks

When you invoke a callback, your web service or Java control is acting as the caller, that is your service or control is sending an outgoing message (a callback invocation) to the client, which responds with an incoming message containing the callback's return value.

If you add a message buffer to a callback of a web service, outgoing messages (that is, invocations of the callback) are buffered on the local machine and the callback invocation returns immediately. This prevents your service from having to wait while the message is sent to the remote server and the (void) response is received. In other words, your web service doesn't have to wait for the network roundtrip to occur. If you add a message buffer to a callback of a Java control, the buffer is implemented by the web service. The Java control must still wait for the network roundtrip even though it will return a void result, but the Java control does not have to wait for the web service to process the message.

All Parameters Must Be Serializable

All objects that are passed as method parameters to buffered methods or callbacks must be serializable. Method and callback invocations are placed in the queue through Java serialization.

Many built in Java classes are already serializable, and most other classes can be made serializable very easily by adding implements java.io.Serializable to the class declaration.

For example, if the Person class in the following example is passed as a method parameter, just add the bold text to the declaration to allow the class to be passed as a parameter to buffered methods or callbacks.

```
public static class Person implements java.io.Serializable
{
    public String firstName;
    public String lastName;
}
```

Controlling Retry Behavior

You can control the retry behavior of WebLogic Server during design or at runtime.

To Specify Retry Behavior During Design

1. Select the asynchronous method or callback as described above.
2. In the ***Properties*** pane, navigate to the ***message−buffer*** property.
3. Set the ***retry−count*** attribute to determine the number of times that redelivery is attempted.
4. Set the ***retry−delay*** attribute to determine the delay between delivery attempts.

To Specify Retry Behavior at Runtime

You can control the retry behavior of WebLogic Server at runtime from within a buffered method. If the method is invoked but you do not wish to service the request immediately, you can request that WebLogic Server reissue the request at a later time. You do this by throwing a weblogic.jws.RetryException from within the buffered method.

The constructors for the RetryException class, described in the following list, take a retry delay as an argument:

RetryException("Try sooner", 5)

The second argument is a long, specifying the number of seconds to delay before retrying.

RetryException("Try later", "5 days")

The second argument is a String specifying the delay before retrying.

RetryException("Try default", RetryException.DEFAULT_DELAY)

Passing the DEFAULT_DELAY constant specifies that the value of the retryDelay attribute of the @jws:message−buffer tag should be used.

Related Topics

Getting Started with Web Services

Getting Started: Using Asynchrony to Enable Long−Running Operations

Designing Conversational Web Services

Designing Conversational Web Services

A single web service may communicate with multiple clients at the same time, and it may communicate with each client multiple times. In order for the web service to track data for the client during asynchronous communication, it must have a way to remember which data belongs to which client and to keep track of where each client is in the process of operations. In WebLogic Workshop, you use conversations to uniquely identify a given communication between a client and your web service and to maintain state between operations.

Conversations are essential for any web service involved in asynchronous communication. This includes web services that communicate with clients using callbacks or polling interfaces, and web services that use controls with callbacks.

Topics Included in This Section

Overview: Conversations

Introduces the concept of conversations.

Life of a Conversation

Describes an example of a web service in a conversation.

Implementing Conversations

Offers guidelines for building web services that supports conversations.

Managing Conversation Lifetime

Describes how you can write code that controls and responds to lifetime events of a conversation.

Supporting Serialization

Introduces Java serialization and explains why it is important for WebLogic Workshop conversations.

Conversing with Non−Workshop Clients

Explains how clients built with tools other than WebLogic Workshop can participate in conversations.

Samples

The following sample web services use conversations:

- Conversation.jws
- HelloWorldAsync.jws
- LuckyNumberDBClient.jws
- QuoteClient.jws

For more information about samples, see Samples.

Related Topics

Getting Started: Using Asynchrony to Enable Long−Running Operations

Overview: Conversations

A web service and a client may communicate multiple times to complete a single task. Also, multiple clients may communicate with the same web service at the same time. *Conversations* provide a straightforward way to keep track of data between calls and to ensure that the web service always responds to the right client.

Conversations meet two challenges inherent in persisting data across multiple communications:

- Conversations uniquely identify a communication between a client and a web service, so that messages are always returned to the correct client. For example, in a shopping cart application, a conversational web service keeps track of which shopping cart belongs to which customer.
- Conversations maintain state between calls to the web service; that is, they keep track of the data associated with a particular client between calls. Conversations ensure that the data associated with a particular client is saved until it is no longer needed or the operation is complete. For example, in a shopping cart application, a conversational web service remembers which items are in the shopping cart while the customer continues shopping.

Correlating Messages with a Unique Identifier

When a client begins a conversation with a service, WebLogic Server creates a context in which to keep track of state−related data during the exchange. This new context is identified by a conversation ID, a string that uniquely identifies the conversation. The web service uses this conversation ID to correlate messages to the client through each operation it performs. The conversation ID ensures that a message sent or received by the web service is always associated with the appropriate client.

You can see the conversation ID in action when you test an asynchronous web service in Test View. For example, if you test the sample service HelloWorldAsynch, you can start multiple conversations and observe how the results are correlated with the conversation ID in test view.

Maintaining State with Conversations

At any given time during a communication between a client and your web service, the web service may need to keep track of data for the client. The web service's *state* refers to all of the data that it is tracking at a particular moment. For example, consider the Investigate sample web service (described in Tutorial: Your First Web Service). In that example, the Investigate service receives a taxpayer ID from its client in the first call to the service and uses this ID to gather credit information. The service stores the taxpayer ID in a member variable so it can be retrieved as needed throughout the lifetime of the exchange between client and service. On the next call, the client does not have to pass the taxpayer ID to the service, which reduces complexity for both the client and the service. In this example, the taxpayer ID is state−related data—that is, data that is saved between calls.

Now imagine that the computer on which the service was running suddenly shut down. If the taxpayer ID were held only by the member variable in the service instance, it would be lost when the instance vanished from memory as the computer shut down. Fortunately, conversations make your web service more robust by writing state−related data to disk, so that the data is preserved through a system failure. This additional security is provided for you automatically when you specify that calls to your web service are part of a conversation.

Related Topics

How Do I: Add Support for Conversations?

Tutorial: Your First Web Service

Implementing Conversations

Life of a Conversation

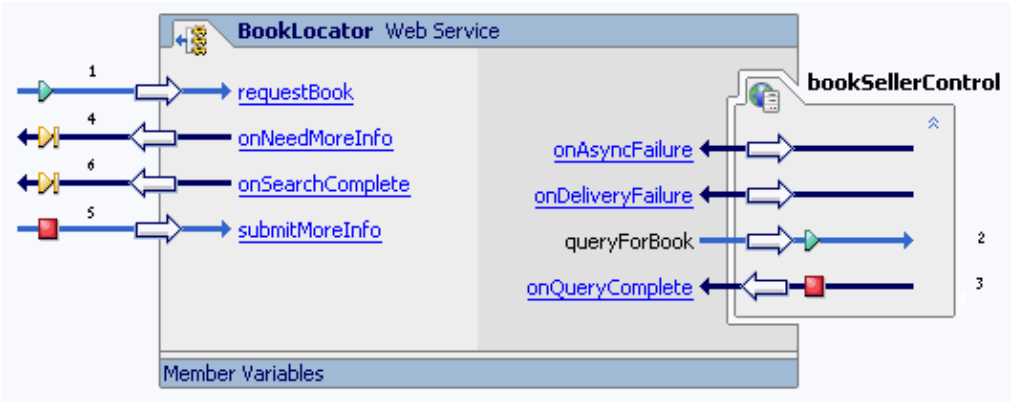
Each method and callback in a conversational web service plays a specific role in the communication between the web service, its clients, and any Java controls the web service uses. A method may begin a conversation, and a method or callback may continue or end a conversation.

To indicate that a method or callback should participate in a conversation, you can select that method or callback in Design View and set the phase attribute of the conversation property for that method or callback. When you set this property for a method or callback, WebLogic Workshop adds one of the following icons to indicate whether the item starts, continues, or finishes a conversation:

To learn more about these attributes, see Implementing Conversations.

To understand how a conversation progresses, let's examine the life cycle of a sample scenario that employs two conversations, one between a web service and its client and another between the web service and a Web Service control. In this example, the BookLocator web service searches a network of book sellers on behalf of a customer. The code within the BookLocator service acts as the middleman and exchanges information between the client and the book seller. The first conversation begins when the client requests a book from the BookLocator service. The BookLocator service then calls the bookSeller service, implemented via a Web Service control, to request the title. When the bookSeller service responds to the BookLocator service, that conversation is complete. The BookLocator then resumes its conversation with the client.

Each of the numbers in the illustration below corresponds to a stage in the life cycle description that follows. Stages 1, 4, 5, and 6 describe the conversation between the BookLocator service and the client. Stages 2 and 3 describe the conversation between the BookLocator service and the bookSeller control.



Stage 1

The client submits a search request by calling the requestBook method, passing information about the book and its author. Code in the requestBook() method saves the book title and author name in member variables as state−related data.

When the request is received, WebLogic Server creates a new context for the service and associates it with the conversation ID. The conversation ID correlates further interactions between the client and the service, ensuring that calls back to the client and calls from other controls are correctly associated with this particular

instance of the service.

The requestBook() method has a void return value. The requestBook() method returns immediately, so that the client is not blocked waiting for a response. The result is sent to the client later via the callback.

As the method returns to the client, WebLogic Server starts the conversation's idle and age timers, which may cause the conversation to time out at a later time for lack of activity.

Stage 2

Next, the code executing as part of the requestBook() method's implementation calls the queryForBook() method of the bookSeller service.

Because this method begins a conversation implemented by the bookSeller service, WebLogic Server creates a new context for communication between the BookLocator service and the bookSeller service. This exchange is managed within its own context with its own conversation ID, and is separate from the exchange between the client and the BookLocator service.

Stage 3

After the book seller searches its inventory for the requested book, the bookSeller service returns its results using the onQueryComplete() callback that is part of its interface. This callback finishes the bookSeller service's conversation with the BookLocator service, and WebLogic Server releases its context and state−related data. The conversation context for the BookLocator service remains active.

The response is handled by the BookLocator service through the onQueryComplete() callback handler. The BookLocator service then examines the data returned by the bookSeller service to determine if it is likely to be useful to the client. In this case, the data includes a long list of books of varying condition and price, so the BookLocator service requests more information from the client in order to narrow the list.

Stage 4

The BookLocator service requests more information from the client through the onNeedMoreInfo() callback. Because the callback is part of its interface, the client software is prepared to handle it. The BookLocator service passes to the client information it can use to decide how to narrow the request.

The onNeedMoreInfo() callback continues the conversation with the client; that is, its conversation phase attribute is set to continue. The onNeedMoreInfo() callback is an intermediary step in the conversation.

Stage 5

The client then responds with information to narrow the list, passing that information back to the BookLocator service as parameters of the submitMoreInfo() method. The call to submitMoreInfo(), like the callback made to the client, also continues the conversation.

Stage 6

The BookLocator service uses the additional information to shorten the list, then employs another callback—onSearchComplete()—to send the short list to the client. This callback completes the conversation

since the callback’s conversation phase attribute is set to finish. Once the callback has executed, WebLogic Server removes the conversation context and state−related data from memory.

Related Topics

How Do I: Add Support for Conversations?

Implementing Conversations

Implementing Conversations

Conversations maintain a web service's state−related data and correlate communications between the web service, its clients, and other resources. You should implement conversations in any web service design that is asynchronous or involves multiple communications with a client or Java control in connection with a single request.

This topic describes the components of a conversation and explains how to build web services that support conversations. It introduces the phase attribute and provides guidelines for implementing conversational callbacks and methods.

Understanding Conversation Context

When a client calls a service operation that is annotated to start a conversation, WebLogic Server creates a conversation *context* through which to correlate calls to and from the service and to persist its state−related data.

When a conversation starts, WebLogic Server does the following:

- Creates a context through which to maintain the scope of the conversation and associates it with a conversation ID.
- Starts an internal timer to measure idle time.
- Starts an internal timer to measure the conversation's age.

When WebLogic Server performs all of these tasks, it creates a context for the conversation. Each piece of information—including the conversation ID, persistent data, idle time and age—is part of the conversation's context.

The conversation ID is a particularly useful item in the conversation’s context. It attaches to each communication, which helps each of the resources, web services, and clients involved in the conversation identify which communications belong to which conversation. To learn more about conversation IDs, see Overview: Conversations.

Designing a Web Service to Use Conversations

As you build services that support conversations, you should keep in mind a few characteristics of conversations. First, WebLogic Server automatically handles correlation between two web services that support conversations. In other words, if your web service supports conversations and calls the conversational methods of another web service that supports conversations, WebLogic Server manages the conversation, including the conversation ID, automatically.

However, the scope of conversation context is limited to the service itself. You cannot assume that the state of another web service is being persisted simply because your service calls one of its methods during the course of a conversation. The other web service is responsible for its own state maintenance.

Also keep in mind that a web service's state is only updated on the successful completion of methods or callback handlers that are marked with the conversation phase attributes start, continue, or finish. This excludes internal methods of your service, which are not operations and so can not be conversational.

Applying the Conversation Phase Attribute

When you apply a conversation phase attribute to a method or callback, you identify it as having a role in conversations. In Design View, you can apply the attribute using the Properties Editor when a method is selected. When you set the conversation phase property, a corresponding annotation like the one shown here is added to your web service's source code immediately preceding the method or callback declaration:

```
@jws:conversation phase="start"
```

The possible values for the conversation phase attribute are start, continue, finish, and none. The following sections describe these values.

The start Value

You can apply the start value to methods only. It specifies that a call to the method starts a new conversation. Each call creates a new conversation context and an accompanying unique conversation ID, saves the service's state, and starts its idle and age timer.

The continue Value

You can apply the continue value to methods and callbacks. It specifies that a call to this method or callback is part of a conversation in progress. WebLogic Server attempts to use the incoming conversation ID to correlate each call to an existing conversation. Each call to a continue method saves the service's state and resets its idle timer.

Set the phase attribute to continue for any method or callback that is likely to be used for communication with a client in connection with an ongoing operation. This includes methods that are clearly intended to be called subsequent to the conversation's start and before its finish as well as requests for responses with additional information, requests for progress or status, and so on.

The finish Value

You can apply the finish value to methods and callbacks. It specifies that a call to this method or callback finishes an existing conversation. A call to a finish method or callback marks the conversation for destruction by WebLogic Server. At some point after a finish method returns successfully, all data associated with the conversation's context is removed.

It is also possible to finish a conversation by calling the JwsContext interface finishConversation method. For more information, see Managing Conversation Lifetime.

The none Value

You can apply the none value to methods and callbacks. It specifies that a call to this method or callback has no meaning in the context of the conversation. none is the default value for the conversation phase attribute.

For more information on adding support for conversations with the conversation phase attribute, see How Do I: Add Support for Conversations?

Implementing Conversational Methods and Callbacks

In practice, adding support for conversations is simply a matter of annotating methods and callbacks with the conversation phase attribute, as described above. Even so, as you write code for methods and callback handlers that will be annotated for use in conversations, you might want to keep the following issues in mind.

Handle Exceptions

Be sure to handle exceptions thrown from conversational methods and callback handlers. An unhandled exception thrown from a conversational method or callback handler cancels any update of state from that method or handler. In other words, state data will not be serialized. So as you plan exception handling for conversational methods and callback handlers, you should consider what an appropriate response to the exception is. The JwsContext interface provides an onException callback handler that is invoked whenever an exception is thrown from an operation (a method marked with the @common:operation annotation) or callback handler.

Depending on your web service's design, your code in the onException handler might:

- Log the error in an application log.
- Notify the client of the failure.
- Finish the conversation.

Keep Methods and Callbacks Short−Lived

Keep conversational methods and callbacks short−lived. Only one method or callback may be executing within a conversation at a given time. This is a benefit in that it means you needn't worry about a new call interfering with one that is already being processed. But because new calls are blocked until the current one finishes executing, you should try to implement each method or callback so that it executes in a short amount of time. Use buffering where appropriate to prevent subsequent clients from being blocked while the web service is busy.

Ensure Support for Serialization

Ensure serialization support for variables that will hold state−related data. In Java, serialization is a technology that makes it possible to write information about an object instance to disk. This requirement can in most cases be met with no extra effort. For more information, see Supporting Serialization.

Stay Within the Code of a Conversation

Do not save state from within code that is not part of a conversation. If your service must save state, you should always try to do so by writing to member variables from within a method that starts or continues a conversation.

Related Topics

JwsContext Interface

@jws:conversation Annotation

@jws:message−buffer Annotation

Managing Conversation Lifetime

WebLogic Server maintains the aspects of a conversation that regulate its life span, including how old it is and how long it has been idle. You can write code to control and respond to these aspects at run time using methods and callbacks exposed by the JwsContext interface.

The life span you want for your conversation depends on the specific actions your service performs. For example, you can extend the maximum age or idle time (or both) when you anticipate that certain processes will take a long time to complete. You can finish a conversation immediately if you know the conversation should terminate. Or you can write code that executes when a conversation is about to end, so you can clean up after your service, release acquired resources, or notify the client that the conversation is about to end.

Using the JwsContext interface, you can manage a conversation's life span by:

- Controlling how long a conversation may remain idle
- Limiting a conversation's duration
- Finishing a conversation
- Doing something when a conversation finishes

Each of these aspects of managing the life span of a conversation is described in greater detail in the sections that follow.

Controlling How Long a Conversation Can Remain Idle

Maximum idle time is the amount of time that can go by between incoming messages before your web service's conversation automatically ends. The length of time you specify depends on what you need your conversation to accomplish. For example, you can temporarily increase the maximum idle time if your code makes an asynchronous follow−up request to a client and you suspect that it will take longer than the maximum idle time to respond. Once the client does respond, your code can update the maximum idle time back to a shorter duration.

You can control the maximum idle time in two ways: you first set the initial idle time at design time, and you can then change the maximum idle time at run time.

Note: The timer keeping track of idle time is only reset by interactions with a client. The idle timer is *not* reset when your web service receives a callback from another web service or Java control (in other words, from entities on the right side of your service in Design View). To reset the idle timer in such circumstances, you can call the resetIdleTime method of the JwsContext interface in a callback handler.

Setting the Initial Maximum Idle Time Value at Design Time

To set the initial maximum idle time in Design View, first click the service area. Next, use the Property Editor window to set the max−idle−time attribute, which is part of the conversation−lifetime property.

By default, the value of this attribute is 0, which means that the conversation may remain idle indefinitely. In general, you want to estimate a reasonable amount of time for your service to remain idle, then give this attribute the value you choose. Allowing a conversation to remain idle for a long period of time can absorb system resources, particularly as new conversation contexts are created for new calls to the service (each with indefinite idle timeout values!).

WebLogic Workshop annotates your service class with a Javadoc annotation when you change the default value of the max−idle−time attribute. So if you do not change the default value the annotation does not appear in your source code. Once the annotation is added, it resembles the following, which sets the value to 2 minutes:

```
/**
 * @jws:conversation−lifetime max−idle−time="2 minutes"
 */
```

The value specified in this attribute represents the starting value for maximum idle time when a new conversation with this service starts. Thereafter, you can update the value through your service's code.

Note: You can change the initial value by editing the Javadoc annotation either in your source code or in the Property Editor; changes in one are automatically reflected in the other.

For reference information about the @jws:conversation annotation, see @jws:conversation−lifetime Annotation.

Changing the Maximum Idle Time Value at Run Time

To manage the maximum idle time at run time, you can use the following JwsContext methods:

- void setMaxIdleTime(long seconds): Sets a new value to the specified number of seconds.
- void setMaxIdleTime(String duration): Sets a new value using a string expressing the idle time duration (such as "2 minutes," "5 days," and so on).
- long getMaxIdleTime(): Returns the current maximum idle time as a number of seconds.
- void resetIdleTime(): Resets the idle timer. This may be useful when the conversation involves some activity besides receiving messages from the client, but this activity is still considered "non−idle." This could include interacting with other web services or Java controls.
- long getCurrentIdleTime(): Returns the number of seconds the conversation has been idle.

In the following example involving a database query, maximum idle time is set to allow at least five minutes for the database query to return:

```
public void getEmployeeContactInfo(String employeeID)
{
    context.setMaxIdleTime("5 minutes");
    Employee m_contactInfo =
        employeeDataControl.getContactInformation(employeeID);
}
```

Limiting the Duration of a Conversation

A conversation's maximum age is the maximum time a conversation instance will exist before it is finished by WebLogic Server.

You can set the maximum age at design time, and you can change the maximum age during run time. This can be useful, for example, if it appears that the combined actions taken by your service to return a response to the client will take longer than originally expected.

Setting the Initial Maximum Age Value at Design Time

To set the initial maximum age in Design View, click the service area, then use the Property Editor window to set the conversation−lifetime max−age attribute.

By default, the value of this attribute is 1 day, meaning that the conversation automatically finishes 24 hours after it starts. In general, it is a good practice to estimate a reasonable amount of time for your service to remain active, then set this attribute to that value. Allowing a conversation to remain active for a long period of time can absorb system resources, particularly as new conversation contexts are created for new calls to the service.

When you change the max−age attribute, WebLogic Workshop annotates your service class with a Javadoc annotation resembling the following example, which sets the value to 7 days.

```
/**
 * @jws:conversation−lifetime max−age="7 days"
 */
```

The value specified in this attribute is the starting value for the maximum age when a new conversation with this service starts. Thereafter, you can update the value through your service's code.

Note: You can change the initial value by editing the Javadoc annotation either in your source code or through the Properties window; changes in one are automatically reflected in the other. Also, note that WebLogic Workshop does not add the Javadoc annotation to your source code until you change the default value of max−age in the Properties window.

For reference information about the @jws:conversation−lifetime annotation, see @jws:conversation−lifetime Annotation.

Changing the Maximum Age Value at Run Time

To manage a conversation's maximum age at run time, you can use the following JwsContext methods:

- void setMaxAge(String duration): Sets a new value using a string expressing the maximum age duration (such as 3 hours, 2 days, and so on).
- void setMaxAge(java.util.Date date): Sets a new value to the specified number of seconds.
- long getMaxAge(): Returns the maximum age of the conversation number in seconds.
- long getCurrentAge(): Returns the number of seconds since the conversation began.

The following simple example illustrates how you can set the maximum age to a duration intended to allow time for a client to gather more information related to their request and return. Extending the conversation's life allows the service's context (along with any persistent state accumulated so far) to remain active until the client returns.

```
public void needMoreInfo(String neededInfoDescription)
{
    context.setMaxAge("2 days");
    callback.onRequestMoreInfo(neededInfoDescription);
}
```

Finishing a Conversation

The end of a conversation marks the conversation context to allow WebLogic Server to remove it from memory. All data that is part of its persistent state is removed. The following list describes the four ways in which a conversation can finish:

- On completion of a web service method annotated with finish or one that invokes a client callback marked finish.
- When the conversation's maximum idle time is exceeded.
- When the conversation's maximum age is exceeded.
- When your code calls the JwsContext interface's finishConversation method.

Note that a conversation never ends during a method invocation. The conversation terminates immediately after the current method or callback invocation completes.

Defining What Happens When a Conversation Finishes

You can write code that executes when a conversation finishes, regardless of how the conversation meets its end. To do this, you implement a handler for JwsContext's onFinish callback handler. You can add this handler to any web service that supports conversations.

When a conversation ends, your callback handler receives a boolean value that indicates whether or not the conversation ended by expiring. A conversation expires if it lasts longer than the value set in the service's conversation−lifetime property max−age attribute or if it has remained idle longer than the value set in the conversation−lifetime property max−idle−time attribute.

The onFinish callback handler will receive a value of false if the conversation ended because your code called the JwsContext.finishConversation method or via execution of an method or callback whose conversation phase attribute is set to finish.

To Add Code to Handle the End of a Conversation

1. Declare a variable for com.bea.control.JwsContext:

   ```
   /** @common:context */
   com.bea.control.JwsContext context;
   ```
2. Using the variable name, add a callback handler for the onFinish callback exposed by the JwsContext interface.
   Your callback handler declaration should resemble the following example, where the variable is context:

   ```
   public void context_onFinish(boolean expired)
   {
   }
   ```
3. Add the code you would like to have execute when the conversation finishes. For example, if for debugging purposes you wanted to write a message to the console to indicate how the conversation had ended, you might implement the handler in the following way:
   ```
   public void context_onFinish(boolean expired)
   {
       String finishStatus = "";
       if(expired){
           finishStatus = "Conversation expired.";
   ```

```
            }
            else{
                finishStatus = "Conversation finished normally.";
            }
            context.getLogger("myWebService").debug(finishStatus);

        }
```

Related Topics

Life of a Conversation

Implementing Conversations

JwsContext Interface

@jws:conversation−lifetime Annotation

Supporting Serialization

When you build web services that use conversations, the conversation state is persisted between method invocations by *serializing* the web service instance to a database. Serialization is the technique through which Java objects may be written to a persistent store as a stream of bytes. When an object is serialized, methods are ignored but all member variables are serialized unless the member variable is marked transient. This means that all member variables of conversational web services must be serializable unless the variable is marked transient.

Note: Since Java controls used by a web service are represented as member variables in the web service, each Java control will also be serialized, meaning the Java control's member variables must also be serializable. The same is true for Java controls that are used, in turn, by Java controls. The WebLogic Workshop compiler will issue compiler errors if member variables are not serializable.

Note: In the case of Web Service controls, the state of the Web Service control will only be serialized of the target web service is conversational.

Note: Remember that a Java control is only a proxy for the resource it represents. Serialization of a control persists the state of the proxy, not the state of the resource itself.

During a conversation, a web service's state is serialized after each successful execution of:

- A method marked with a @common:operation annotation and marked with a @jws:conversation phase attribute of start or continue.
- A control callback handler.

Note that serialization will not occur for these if the method, callback, or callback handler throws an unhandled exception.

The requirement that member variables be serializable is typically easy to meet. For Java primitive types, serialization is supported by default. These types include byte, boolean, char, short, int, long, double, and float. Also, while classes in Java must implement the Serializable interface to be serializable, many of the Java classes representing common data types already do so. These types include those that are wrappers around the primitive types—such as Boolean, Long, Double, Integer, and so on—as well as String and StringBuffer. (When you are in doubt about whether a particular common class implements Serializable, check reference information for the class before using it.)

When you create your own classes for use in typing member variables, or when you handle classes created by others, you must take care to ensure that these classes implement the Serializable interface. Even so, this is typically easy to do because the interface contains no methods to implement—implementing the interface merely marks the class as supporting serialization. Your class code must simply import the package containing the interface and its declaration must be marked with the implements keyword, as follows:

```
import java.io.Serializable;


public class MyClass implements Serializable
{...}
```

For more information on Java object serialization, please consult your favorite Java book.

Conversations and Clusters

Clusters are not directly aware of conversations. Conversational state is stored in a database that is available from all servers in the cluster. The cluster handles a conversation continuing or finishing request in the following way: (1) the request gets routed to a cluster member, (2) the current conversational state is loaded from the database, (3) the operation is run, and (4) the conversational state is updated in the database. Subsequent conversational requests are handled this way each time, regardless of which server in the cluster handles the request.

The conversational state is stored in the database and is updated by conversational operations from whatever cluster member handles the request. Row−level locks on the database prevent the execution of multiple, simultaneous queries to the same conversation. The transactional capabilities of the database are used to insure that conversational state is consistent and durable.

To optimize performance, if the request continues a conversation and the conversational state hasn't changed, only the last−access−time field in the database is updated. This writes a Java.lang.long data type to the database, but avoids writing a BLOB.

Related Topics

Overview: Conversations

How Do I: Add Support for Conversations?

Conversing with Non−Workshop Clients

For services you build with WebLogic Workshop, adding support for conversations is easy. At design time, you simply set the conversation property's phase attribute for your service's methods and callbacks.

The SOAP messages that invoke conversational web service methods contain extra SOAP headers that include a conversation ID. If both the client and web service are build in WebLogic Workshop, the conversation SOAP headers are managed automatically. Web service clients built using other tools, however, must manually provide the necessary SOAP headers when invoking conversational WebLogic Workshop web service methods.

Note: If you are developing web services with Microsoft .NET (or need to interoperate with .NET services), you may be interested in the ConversationClient.asmx.cs sample. This file contains code for a conversational web service client written in C#. For more information, see .NET Client Sample.

The three SOAP headers related to conversations are:

- <StartHeader>: The client sends this header with the call to a method marked to "start" a conversation. The <StartHeader> contains <conversationID> and <callbackLocation> elements.
- <ContinueHeader>: The client sends this with any subsequent calls to the web service's methods. It contains the conversation ID the client used to start the conversation.
- <CallbackHeader>: The client receives this header when it receives a callback from the web service. The <CallbackHeader> includes the conversation ID the client used to start the conversation.

As you can see, one piece of information all of the headers carry is the conversation ID. The client proposes a conversation ID with any request to a start method. The same ID is then passed back and forth between the client and the web service for each subsequent exchange. This way, both know to which conversation each message is related.

Conversation SOAP Headers

A client's first request, which begins a conversation, is always sent to a method marked to "start" the conversation. The SOAP message for this request must include the <StartHeader> header. In the message, this header looks like the following:

```
<SOAP−ENV:Header>
    <StartHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
        <conversationID>someUniqueIdentifier</conversationID>
        <callbackLocation>http://www.mydomain.com/myClient</callbackLocation>
    </StartHeader>
</SOAP−ENV:Header>
```

The <StartHeader> element simply contains the header information. The XML namespace must be specified as it is shown here.

The <conversationID> element is technically optional. If it is omitted, WebLogic Server will invent one for use locally. However, the client will have no way of knowing what this conversation ID is if they don't send it. This means that they will be unable to correlate any responses received with the original request. For this reason, the client should only omit the conversation ID if they do not expect any further conversation with the web service. The value of the conversation ID is a string with a maximum length of 768 bytes. The conversation ID should be unique across all possible conversational clients of the target web service. To

construct such an identifier, you may wish to combine data such as the process ID of the client, the current date and time and the host name or host IP address of the client.

The <callbackLocation> element, also optional, contains the URL to which callbacks should be sent. The client needs to send this only if it expects to handle callbacks from the web service. All callbacks intended for this client will be sent to the specified URL.

After the first request, the client either continues its side of the conversation by calling additional methods of the web service or receives callbacks from the web service. Each subsequent call to the web service must include the <ContinueHeader>. Note that this means even calls to web service methods marked to "finish" the conversation must include the <ContinueHeader>. The <ContinueHeader> looks like the following:

```
<SOAP−ENV:Header>
    <ContinueHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
        <conversationID>theUniqueIdentifierSentWithTheStartHeader</conversationID>
    </ContinueHeader>
</SOAP−ENV:Header>
```

In order to correctly correlate this invocation, the <conversationID> used here must be the same as the ID sent with the <StartHeader>.

Finally, the client may be designed to handle callbacks from the web service. This means that the client software includes an operation capable of receiving the message sent by the WebLogic Workshop web service's callback. How this is implemented differs depending on the technology used to build the client. In general, though, the message sent with a callback contains the callback's parameter values.
The callback message also contains the <CallbackHeader>. The <CallbackHeader> also includes the conversation ID. It looks like the following:

```
<SOAP−ENV:Header>
    <CallbackHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
        <conversationID>theUniqueIdentifierSentWithTheStartHeader</conversationID>
    </CallbackHeader>
</SOAP−ENV:Header>
```

Notice that there is no "finish" header. The conversation is over when the web service finishes it. This may be through execution of an operation (method or callback) marked with the conversation phase set to "finish". The web service can also call the finish method of the JwsContext interface (something it might do in the event of an exception). Using the WSDL file for the web service, the client knows which operations are designed to finish the conversation.

Once a conversation is finished, any further continue messages sent with that conversation ID will result in a SOAP fault.

What to Look for in a Conversational Web Service's WSDL File

By looking at the WSDL file generated from a WebLogic Workshop service, a developer of a client of the web service can discover which of the service's operations start, continue or finish a conversation. They can also see which SOAP header each operation requires.

The following excerpt is from a WSDL generated from the Conversation.jws web service in the WebLogic Workshop samples project. In this example:

- The name of each operation (method and callback) is highlighted in ***bold***.
- The phase attribute describing the operation's role in a conversation is highlighted in ***blue***.
- The part attribute describing which header the service will receive (as an "input" header) and which it will send (as an "output" header) is highlighted in ***red***.

By looking at this excerpt, you can see that:

- The startRequest operation starts the conversation and requires a <StartHeader>.
- The getRequestStatus operation continues the conversation and requires a <ContinueHeader>.
- The terminateRequest operation finishes the conversation and requires a <ContinueHeader>.
- The onResultReady operation finishes the conversation and sends a <CallbackHeader>.

```
<binding name="ConversationSoap" type="s0:ConversationSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
  <operation name="startRequest">
    <soap:operation soapAction="http://www.openuri.org/startRequest" style="document" />
      <cw:transition phase="start" />
    <input>
      <soap:body use="literal" />
      <soap:header wsdl:required="true" message="s0:StartHeader_literal" part="StartHeader" u
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
  <operation name="getRequestStatus">
    <soap:operation soapAction="http://www.openuri.org/getRequestStatus" style="document" />
    <cw:transition phase="continue" />
    <input>
      <soap:body use="literal" />
      <soap:header wsdl:required="true" message="s0:ContinueHeader_literal" part="ContinueHea
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
  <operation name="terminateRequest">
    <soap:operation soapAction="http://www.openuri.org/terminateRequest" style="document" />
    <cw:transition phase="finish" />
    <input>
      <soap:body use="literal" />
      <soap:header wsdl:required="true" message="s0:ContinueHeader_literal" part="ContinueHea
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
  <operation name="onResultReady">
    <soap:operation soapAction="http://www.openuri.org/onResultReady" style="document" />
    <cw:transition phase="finish" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
      <soap:header wsdl:required="true" message="s0:CallbackHeader_literal" part="CallbackHea
    </output>
  </operation>
</binding>
```

Related Topics

Life of a Conversation

Overview: Conversations

How Do I: Add Support for Conversations?

How Do I: Tell Developers of Non−WebLogic Workshop Clients How to Participate in Conversations?

Best Practices for Designing Asynchronous Interfaces

This section discusses the best practices for creating and using web services and Java controls with asynchronous interfaces. The first topic describes how to use polling as an alternative to callbacks. Then, various design principles are recommended for designing web services and Java controls that can be called by both other web services and JSP (web) pages.

Topics Included in This Section

Using Polling as an Alternative to Callbacks

Describes how to provide polling interfaces for clients of your web service or Java control that cannot handle callbacks.

Designing Robust Asynchronous Interfaces

Discusses how to design robust asynchronous web services and Java controls.

Related Topics

Building Web Services

Developing Web Applications

Using Polling as an Alternative to Callbacks

Because callbacks are, by definition, separated from the original request to which the callback is a response, they appear as unsolicited messages to the client's host. Many hosts refuse unsolicited network traffic, either because they directly reject such traffic or because they are protected by firewalls or other network security apparatus. Clients that run in such environments are therefore not capable of receiving callbacks.

Another requirement for handling callbacks is that the client is persistent by being conversational. If the client is a web application, that is, a JSP page, or a non−conversational web service, it cannot handle callbacks.

In order to allow clients that can't accept callbacks to use your web services, you can supply a polling interface as an alternative. In a polling interface, you provide one or more methods that a client can call periodically to determine whether the result of a previous request is ready. Although the web service or Java control will still be asynchronous in design, the interactions with a client are handled with synchronous (unbuffered) methods.

A typical design of a polling interface will have these three methods:

- A *start_request* method that the client will call to initiate a request. If the client calls a web service, the *start_request* method will start a conversation.
- A *check_status* method that the client will periodically call to check the status of the request. The method returns a boolean value indicating whether or not the request has been handled. If the client calls a web service, the *check_status* method will continue the conversation.
- A *get_results* method that the client will call to get the results of the request. The results may for instance be returned as a String or an object of some kind, or null if the request could not be processed successfully. If the client calls a web service, the *get_results* method will finish the conversation.

Notice that a client using a polling interface needs to periodically check the status of the request, because the web service or Java control cannot notify the client when its request has been processed. Also notice that the three methods will not be buffered. The *check_status* and *get_results* methods do not return void and cannot be buffered, while the *start_request* method cannot be buffered because you need to ensure that this method has been handled before the *check_status* is handled. (Remember that the relative handling order of buffered and unbuffered methods is uncertain. For more information, see Using Buffering to Create Asynchronous Methods and Callbacks.)

There are several other ways to implement a polling interface. The following example taken from the source code of the Conversation.jws Sample web service shows one such variation:

```
public class Conversation {
    /**
     * @common:operation
     * @jws:conversation phase="start"
     */
    public void startRequest()
    {
         ...
    }
    /**
     * @common:operation
     * @jws:conversation phase="continue"
     */
    public String getRequestStatus()
    {
```

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

A client uses the startRequest method to initiate a request from a conversational web service. The client then calls getRequestStatus periodically to check on the result. As before, the client is free to perform other processing between calls to getRequestStatus. The getRequestStatus method returns an indication that the request is pending until the request is complete. The next time the client calls getRequestStatus after the request is complete, the result is returned to the client. The client then calls terminateRequest to finish the conversation.

Related Topics

Web Services Development Cycle

Getting Started: Using Asynchrony to Enable Long−Running Operations

Designing Conversational Web Services

Designing Robust Asynchronous Interfaces

The preceding sections have presented various design options that can be included when building web services and Java controls. Given these options, what exactly constitutes a good design and creates a successful and robust web service or Java control? This topic explores several typical design solutions.

Do I Need an Asynchronous Interface?

The first question you might need to answer for a web service or Java control is whether the service or control needs to be asynchronous. There are certainly cases where a non−conversational, synchronous service or control will suffice, especially when the functionality it implements is relatively small, the request handling is relatively short, and the underlying infrastructure supporting this web service is solid, for instance if you are working on a fail−proof intranet or if your control is called by a (synchronous) web service on the same server. However, if any of these factors are not true or uncertain at the time of design, you will want to make your service or control asynchronous.

Do I Need to Use Callbacks?

Callbacks are a powerful approach to designing asynchronous web services or Java controls, relieving the client from periodically checking a request's status, as is required with polling. Especially when it is unclear how long a request will take to process, or if processing times vary wildly, using a callback is likely the most elegant implementation of asynchrony and loose coupling. Using callbacks in combination with buffering of both methods and callbacks is particularly effective in dealing with high−volume traffic. However, callbacks require that the client is designed to accept incoming messages and is hosted in an environment that supports message delivery.

Do I Need to Use Polling?

All asynchronous web services and Java controls should provide a polling interface. If the web service or Java control is asynchronous, a polling interface is required by any client that cannot accept callbacks; a polling interface is the only way such a client can obtain the results of operations initiated by asynchronous method invocations. You should think of a polling interface as the foundation interface of a web service or Java control, and callbacks as "extra" functionality that is convenient for clients who can handle callbacks.

The exception to this guideline is a Java control for which the *only* clients will be conversational web services or Java controls invoked by conversational web services. Conversational WebLogic Workshop web services can always accept callbacks. However, Java controls should be designed to be reusable. Assuming that the only clients a Java control will ever have are WebLogic Workshop web services limits the reusability of the Java control.

A Robust Web Service or Java Control

To create an asynchronous web service or Java control that is robust and handles all situations, it is recommended that you implement both a callback and a polling interface. Your design might (among others) include the following methods:

- A *start_request*_asynch buffered method that the client will call to initiate a request. The method starts a conversation and notes that the callback mechanism will be used when the results are ready.

- A *callback_results* buffered callback that sends the results to the client when the request is completed and finishes the conversation.

- A *start_request*_synch buffered method that the client will call to initiate a request. The method starts a conversation and notes that the polling mechanism will be used when the results are ready.
- A *check_status* unbuffered method that the client will periodically call to check the status of the request. The method continues a conversation and returns a Boolean value indicating whether or not the request has been completely handled.
- A *get_results* unbuffered method that the client will call to get the results of the request. The method finishes the conversation.

Other implementations of this design are possible. For a variation, see Using Polling as an Alternative to Callbacks.

Related Topics

Building Web Services with WebLogic Workshop