BLAZEDS
Developer Guide

© 2008 Adobe Systems Incorporated. All rights reserved.

BlazeDS Developer Guide

If this guide is distributed with software that includes an end-user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end-user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names in sample templates are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, ActionScript, Adobe AIR, ColdFusion, Dreamweaver, Flash, Flash Player, Flex, Flex Builder, and LiveCycle are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

The name “BlazeDS” and the BlazeDS logo must not be used to endorse or promote products derived from this software without prior written permission from Adobe. Similarly, products derived from this open source software may not be called "BlazeDS", nor may "BlazeDS" appear in their name or the BlazeDS logo appear with such products, without prior written permission of Adobe.

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Apple, Macintosh, and Mac OS are trademarks of Apple Inc., registered in the United States and other countries. Java and JavaScript are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All other trademarks are the property of their respective owners. All other trademarks are the property of their respective owners.

This product includes software developed by the OpenSymphony Group (http://www.opensymphony.com/).

This product includes software developed by the Apache Software Foundation (http://www.apache.org/).

This product contains either BISAFE and/or TIPEM software by RSA Data Security, Inc.

The Flex Builder 3 software contains code provided by the Eclipse Foundation (“Eclipse Code”). The source code for the Eclipse Code as contained in Flex Builder 3 software (“Eclipse Source Code”) is made available under the terms of the Eclipse Public License v1.0 which is provided herein, and is also available at http://www.eclipse.org/legal/epl-v10.html.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA.

Notice to U.S. government end users. The software and documentation are “Commercial Items,” as that term is defined at 48 C.F.R. §2.101, consisting of “Commercial Computer Software” and “Commercial Computer Software Documentation,” as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. For U.S. Government

End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250 ,and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

Contents

Part 1: Getting started with BlazeDS

Chapter 1: Introduction to BlazeDS

BlazeDS overview 2
BlazeDS features 3
Example BlazeDS applications 5

Chapter 2: Building and deploying BlazeDS applications

Setting up your development environment 10
Running the BlazeDS sample applications 14
Building your client-side application 15
Building your server-side application 20
Debugging your application 22
Deploying your application 25

Part 2: BlazeDS architecture

Chapter 3: BlazeDS architecture

BlazeDS client architecture 27
BlazeDS server architecture 29
About configuration files 31

Chapter 4: Channels and endpoints

About channels and endpoints 38
Configuring channels with servlet-based endpoints 43
Channel and endpoint recommendations 49
Using BlazeDS clients and servers behind a firewall 50

Chapter 5: Managing session data

FlexClient, MessageClient, and FlexSession objects 51
Using the FlexContext class with FlexSession and FlexClient attributes 53
Session life cycle 54

Chapter 6: Data serialization

Serializing between ActionScript and Java 56
Serializing between ActionScript and web services 64

Part 3: RPC services

Chapter 7: Using HTTP and web services

RPC components 73
RPC components versus other technologies 77
Using destinations 78

Defining and invoking a service component 83
Handling service events 86
Passing parameters to a service 90
Handling service results 94
Using capabilities specific to WebService components 103
Handling asynchronous calls to services 107

Chapter 8: Using the Remoting Service
RemoteObject component 110
Configuring a destination 114
Calling a service 115
Handling events 117
Passing parameters 117
Handling results 119
Accessing EJBs and other objects in JNDI 120

Part 4: Messaging Service

Chapter 9: Using the Messaging Service
Using the Messaging Service 122
Working with Producer components 125
Working with Consumer components 128
Using a pair of Producer and Consumer components in an application 131
Message filtering 132
Configuring the Messaging Service 136

Chapter 10: Connecting to the Java Message Service (JMS)
About JMS 141
Configuring the Messaging Service to connect to a JMSAdapter 143

Part 6: Administering BlazeDS applications

Chapter 11: Logging
Client-side logging 149
Server-side logging 150
Monitoring and managing services 153

Chapter 12: Security
Securing BlazeDS 156
Configuring security 158
Basic authentication 162
Custom authentication 162
Passing credentials to a proxy service 166

Chapter 13: Clustering
Server clustering 167
Handling channel failover 167

Cluster-wide message and data routing 169
Configuring clustering 170

Part 7: Additional programming topics

Chapter 14: Run-time configuration
About run-time configuration 174
Configuring components with a bootstrap service 175
Configuring components with a remote object 175
Accessing dynamic components with a Flex client application 177

Chapter 15: The Ajax client library
About the Ajax client library 179
Using the Ajax client library 179
Ajax client library API reference 183

Chapter 16: Extending applications with factories
The factory mechanism 188

Chapter 17: Message delivery with adaptive polling
Adaptive polling 193
Using a custom queue processor 194

Chapter 18: Measuring message processing performance
About measuring message processing performance 199
Measuring message processing performance 204

Part 1: Getting started with BlazeDS

Introduction to BlazeDS 2
Building and deploying BlazeDS applications 10

Chapter 1: Introduction to BlazeDS

BlazeDS provides highly scalable remote access and messaging for use with client-side applications built in Adobe® Flex® or Adobe® AIR™.

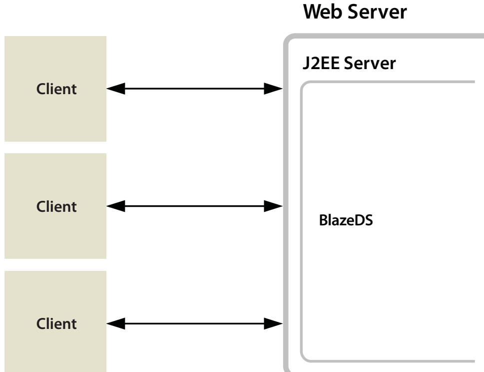
Topics

BlazeDS overview 2
BlazeDS features 3
Example BlazeDS applications 5

BlazeDS overview

BlazeDS provides a set of services that lets you connect a client-side application to server-side data, and pass data among multiple clients connected to the server.BlazeDS implements real-time messaging between clients.

A BlazeDS application consists of two parts: a client-side application and a server-side J2EE web application. The following figure shows this architecture:



The client-side application

A BlazeDS client application is typically an Adobe Flex or AIR application. Flex and AIR applications use Flex components to communicate with the BlazeDS server, including the RemoteObject, HTTPService, WebService, Producer, and Consumer components. The HTTPService, WebService, Producer, and Consumer components are part of the Flex Software Development Kit (SDK).

Although you typically use Flex or AIR to develop the client-side application, you can develop the client as a combination of Flex, HTML, and JavaScript. Or, you can develop it in HTML and JavaScript by using the Ajax client library to communicate with BlazeDS. For more information on using the Ajax client library, see "The Ajax client library" on page 179.

The BlazeDS server

The BlazeDS server runs in a web application on a J2EE application server. BlazeDS includes three preconfigured web applications that you can use as the basis of your application development. For more information on using these web applications, see "Building and deploying BlazeDS applications" on page 10.

Configure an existing J2EE web application to support BlazeDS by performing the following steps:

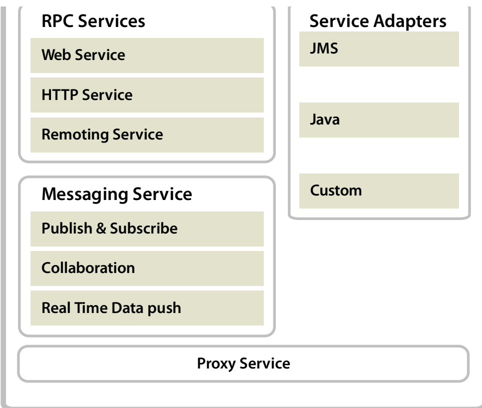
1 Add the BlazeDS JAR files and dependent JAR files to the WEB-INF/lib directory.

2 Edit the BlazeDS configuration files in the WEB-INF/flex directory.

3 Define MessageBrokerServlet and a session listener in WEB-INF/web.xml.

BlazeDS features

The following figure shows the main features of BlazeDS:



BlazeDS core features

The BlazeDS core features include the RPC services and the Messaging Service.

RPC services

The Remote Procedure Call (RPC) services are designed for applications in which a call and response model is a good choice for accessing external data. RPC services let a client application make asynchronous requests to remote services that process the requests and then return data directly to the client. You can access data through client-side RPC components that include HTTP GET or POST (HTTP services), SOAP (web services), or Java objects (remote object services).

Use RPC components when you want to provide enterprise functionality, such as proxying of service traffic from different domains, client authentication, whitelists of permitted RPC service URLs, server-side logging, localization support, and centralized management of RPC services. BlazeDS lets you use RemoteObject components to access remote Java objects without configuring them as SOAP-compliant web services.

A client-side RPC component calls a remote service. The component then stores the response data from the service in an ActionScript object from which you can easily obtain the data. The client-side RPC components are the HTTPService, WebService, and RemoteObject components.

***Note:** You can use Flex SDK without the BlazeDS proxy service to call HTTP services or web services directly. You cannot use RemoteObject components without BlazeDS or ColdFusion.*

For more information, see "Using HTTP and web services" on page 73.

Messaging Service

The Messaging Service lets client applications communicate asynchronously by passing messages back and forth through the server. A message defines properties such as a unique identifier, BlazeDS headers, any custom headers, and a message body.

Client applications that send messages are called message *producers*. You define a producer in a Flex application by using the Producer component. Client applications that receive messages are called message *consumers*. You define a consumer in a Flex application by using the Consumer component. A Consumer component subscribes to a server-side destination and receives messages that a Producer component sends to that destination. For more information on messaging, see "Using the Messaging Service" on page 122.

The Messaging Service also supports bridging to JMS topics and queues on an embedded or external JMS server by using the JMSAdapter. Bridging lets Flex client applications exchange messages with Java client applications. For more information, see "Connecting to the Java Message Service (JMS)" on page 141.

Service adapters

BlazeDS lets you access many different persistent data stores and databases including JMS, and other data persistence mechanisms. A service adapter is responsible for updating the persistent data store on the server in a manner appropriate to the specific data store type. The adapter architecture is customizable to let you integrate with any type of messaging or back-end persistence system.

The message-based framework

BlazeDS uses a message-based framework to send data back and forth between the client and server. BlazeDS uses two primary exchange patterns between server and client. In the first pattern, the request-response pattern, the client sends a request to the server to be processed. The server returns a response to the client containing the processing outcome. The RPC services use this pattern.

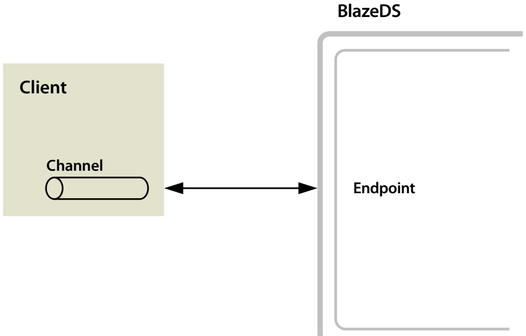
The second pattern is the publish-subscribe pattern where the server routes published messages to the set of clients that have subscribed to receive them. The Messaging Service uses this pattern to push data to interested clients. The Messaging Service also uses the request-response pattern to issue commands, publish messages, and interact with data on the server.

Channels and endpoints

To send messages across the network, the client uses channels. A channel encapsulates message formats, network protocols, and network behaviors to decouple them from services, destinations, and application code. A channel formats and translates messages into a network-specific form and delivers them to an endpoint on the server.

Channels also impose an order to the flow of messages sent to the server and the order of corresponding responses. Order is important to ensure that interactions between the client and server occur in a consistent, predictable fashion.

Channels communicate with Java-based endpoints on the server. An endpoint unmarshals messages in a protocol-specific manner and then passes the messages in generic Java form to the message broker. The message broker determines where to send messages, and routes them to the appropriate service destination.



For more information on channels and endpoints, see "BlazeDS architecture" on page 27.

Channel types

BlazeDS includes several types of channels, including standard and secure Action Message Format (AMF) channels and HTTP (AMFX) channels. AMF and HTTP channels support non-polling request-response patterns and client polling patterns to simulate real-time messaging. The streaming AMF and HTTP channels provide true data streaming for real-time messaging.

BlazeDS summary of features

The following table summarizes some of the main features of BlazeDS:

Feature	Description
Proxy service	Enables communication between clients and domains that they cannot access directly, due to security restrictions, allowing you to integrate multiple services with a single application. By using the Proxy Service, you do not have to configure a separate web application to work with web services or HTTP services.
Publish and subscribe messaging	Provides a messaging infrastructure that integrates with existing messaging systems such as JMS. This service enables messages to be exchanged in real time between browser clients and the server. It allows Flex clients to publish and subscribe to message topics with the same reliability, scalability, and overall quality of service as traditional thick client applications.
Software clustering	Handles failover when using stateful services to ensure that Flex applications continue running in the event of server failure. The more common form of clustering using load balancers, usually in the form of hardware, is supported without any feature implementation.

Example BlazeDS applications

The following example applications show client-side and server-side code that you can compile and deploy to get started with BlazeDS. You typically use the following steps to build an application:

Configure a destination in the BlazeDS server used by the client application to communicate with the server. A destination is the server-side code that you connect to from the client. Configure a destination in one of the configuration files in the WEB-INF/flex directory of your web application.

1 Configure a channel used by the destination to send messages across the network. The channel encapsulates message formats, network protocols, and network behaviors and decouples them from services, destinations, and application code. Configure a channel in one of the configuration files in the WEB-INF/flex directory of your web application.

2 Write the Flex client application in MXML or ActionScript.

3 Compile the client application into a SWF file by using Flex Builder or the mxmlc compiler. When you compile your Flex application, specify the services-config.xml configuration file to the compiler. This file defines the destinations and channels that the client application uses to communicate with the server.

Deploy the SWF file to your BlazeDS web application.

Running the examples

The BlazeDS installation creates a directory structure on your computer that contains all of the resources necessary to build applications. The installation includes three web applications that you can use as the basis of your development environment. The samples web application contains many BlazeDS examples.

You can run the following examples if you compile them for the samples web application and deploy them to the samples directory structure. For more information on building and running the examples, see "Building and deploying BlazeDS applications" on page 10.

RPC service example

The Remoting Service is one of the RPC services included with BlazeDS. The Remoting Service lets clients access methods of Plain Old Java Objects (POJOs) on the server.

In this example, you deploy a Java class, EchoService.java, on the server that echoes back a String passed to it from the client. The following code shows the definition of EchoService.java:

```
package remoting;
public class EchoService
{
    public String echo(String text) {
        return "Server says: I received '" + text + "' from you";
    }
}
```

The `echo()` method takes a String argument and returns it with additional text. After compiling EchoService.java, place EchoService.class in the WEB-INF/classes/remoting directory. Notice that the Java class does not have to import or reference any BlazeDS resources.

Define a destination, and reference one or more channels that transport the data. Configure EchoService.class as a remoting destination by editing the WEB-INF/flex/remoting-config.xml file and adding the following code:

```
<destination id="echoServiceDestination" channels="my-amf">
    <properties>
        <source>remoting.EchoService</source>
    </properties>
</destination>
```

The `source` element references the Java class, and the `channels` attribute references a channel called my-amf.

Define the my-amf channel in WEB-INF/flex/services-config.xml, as the following example shows:

```
<channel-definition id="my-amf" class="mx.messaging.channels.AMFChannel">
    <endpoint url="http://{server.name}:{server.port}/{context.root}/messagebroker/amf"
        class="flex.messaging.endpoints.AMFEndpoint"/>
    <properties>
        <polling-enabled>false</polling-enabled>
```

```
    </properties>
</channel-definition>
```

The channel definition specifies that the Flex client uses a non-polling AMFChannel to communicate with the AMFEndpoint on the server.

***Note:** If you deploy this application on the samples web application installed with BlazeDS, services-config.xml already contains a definition for the my-amf channel.*

The Flex client application uses the RemoteObject component to access EchoService. The RemoteObject component uses the `destination` property to specify the destination. The user clicks the Button control to invoke the remote `echo()` method:

```
<?xml version="1.0"?>
<!-- intro\intro_remoting.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="100%" height="100%">

    <mx:Script>
        <![CDATA[
            import mx.rpc.events.FaultEvent;
            import mx.rpc.events.ResultEvent;

            // Send the message in response to a Button click.
            private function echo():void {
                var text:String = ti.text;
                remoteObject.echo(text);
            }

            // Handle the recevied message.
            private function resultHandler(event:ResultEvent):void {
                ta.text += "Server responded: "+ event.result + "\n";
            }

            // Handle a message fault.
            private function faultHandler(event:FaultEvent):void {
                ta.text += "Received fault: " + event.fault + "\n";
            }
        ]]>
    </mx:Script>

    <mx:RemoteObject id="remoteObject"
        destination="echoServiceDestination"
        result="resultHandler(event);"
        fault="faultHandler(event);"/>

    <mx:Label text="Enter a text for the server to echo"/>
    <mx:TextInput id="ti" text="Hello World!"/>
    <mx:Button label="Send" click="echo();"/>
    <mx:TextArea id="ta" width="100%" height="100%"/>
</mx:Application>
```

Compile the client application into a SWF file by using Flex Builder or the mxmlc compiler, and then deploy it to your web application.

Messaging Service example

The Messaging Service lets client applications send and receive messages from other clients. In this example, create a Flex application that sends and receives messages from the same BlazeDS destination.

Define the messaging destination in WEB-INF/flex/messaging-config.xml, as the following example shows:

```
<destination id="MessagingDestination" channels="my-amf-poll"/>
```

Define the my-amf-poll channel in WEB-INF/flex/services-config.xml, as the following example shows:

```
<channel-definition id="my-amf-poll" class="mx.messaging.channels.AMFChannel">
    <endpoint
        url="http://{server.name}:{server.port}/{context.root}/messagebroker/amfpoll"
        class="flex.messaging.endpoints.AMFEndpoint"/>
    <properties>
        <polling-enabled>true</polling-enabled>
        <polling-interval-seconds>1</polling-interval-seconds>
    </properties>
</channel-definition>
```

This channel definition creates a polling channel with a polling interval of 1 second. Therefore, the client sends a poll message to the server every second to request new messages. Use a polling channel because it is the easiest way for the client to receive updates. Other options include polling with piggybacking, long-polling, and streaming.

The following Flex client application uses the Producer component to send a message to the destination, and the Consumer component to receive messages sent to the destination. To send the message, the Producer first creates an instance of the AsyncMessage class and then sets its `body` property to the message. Then, it calls the `Producer.send()` method to send it. To receive messages, the Consumer first calls the `Consumer.subscribe()` method to subscribe to messages sent to a specific destination.

```
<?xml version="1.0"?>
<!-- intro\intro_messaging.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="100%" height="100%"
    creationComplete="consumer.subscribe();">

    <mx:Script>
        <![CDATA[
            import mx.messaging.events.MessageFaultEvent;
            import mx.messaging.events.MessageEvent;
            import mx.messaging.messages.AsyncMessage;
            import mx.messaging.Producer;
            import mx.messaging.Consumer;

            // Send the message in response to a Button click.
            private function sendMessage():void {
                var msg:AsyncMessage = new AsyncMessage();
                msg.body = "Foo";
                producer.send(msg);
            }

            // Handle the received message.
            private function messageHandler(event:MessageEvent):void {
                ta.text += "Consumer received message: "+ event.message.body + "\n";
            }

            // Handle a message fault.
            private function faultHandler(event:MessageFaultEvent):void {
                ta.text += "Received fault: " + event.faultString + "\n";
            }
        ]]>
    </mx:Script>

    <mx:Producer id="producer"
        destination="MessagingDestination"
        fault="faultHandler(event);"/>

    <mx:Consumer id="consumer"
```

```
        destination="MessagingDestination"
        fault="faultHandler(event);"
        message="messageHandler(event);"/>

    <mx:Button label="Send" click="sendMessage();"/>
    <mx:TextArea id="ta" width="100%" height="100%"/>
</mx:Application>
```

Compile the client application into a SWF file by using Flex Builder or the mxmlc compiler, and then deploy it to your web application.

Chapter 2: Building and deploying BlazeDS applications

BlazeDS applications consist of client-side code and server-side code. Client-side code is typically a Flex application written in MXML and ActionScript and deployed as a SWF file. Server-side code is written in Java and deployed as Java class files or Java Archive (JAR) files. Every BlazeDS application has client-side code; however, you can implement an entire application without writing any server-side code.

For more information on the general application and deployment process for Flex applications, see *Building and Deploying Adobe Flex 3 Applications*.

Topics

Setting up your development environment 10
Running the BlazeDS sample applications 14
Building your client-side application 15
Building your server-side application 20
Debugging your application 22
Deploying your application 25

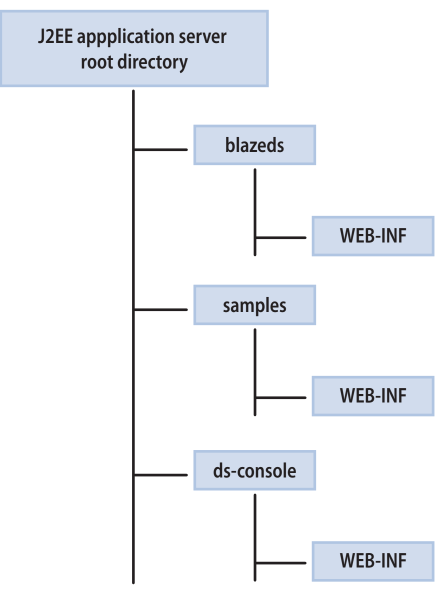
Setting up your development environment

BlazeDS applications consist of two parts: client-side code and server-side code. Before you start developing your application, configure your development environment, including the directory structure for your client-side source code and for your server-side source code.

Installation directory structure

If you have downloaded the BlazeDS Turnkey ZIP file, unzip it to create a directory structure on your computer that contains all of the resources necessary to build your application. The directory structure includes three web applications that you can use as the basis of your development environment. Alternately, you can download and unzip the BlazeDS binary distribution ZIP file, which contains just the blazeds.war web application file and a readme file; you deploy the blazeds.war file into your application server. See the BlazeDS installation instructions for more information.

The following example shows the directory structure of the web applications installed with BlazeDS Turnkey installation:



The BlazeDS Turnkey ZIP file includes an integrated Tomcat application server to host these web applications. Alternatively, you can install BlazeDS without installing Tomcat. Instead, you deploy the blazeds web application on your J2EE application server or servlet container; the blazeds web application is available as a separate download in the BlazeDS binary distribution ZIP file.

The following table describes the directory structure of each web application in the turnkey installation:

Directory	Description
/blazeds /samples /ds-console	The root directory of a web application. Contains the WEB-INF directory. This directory also includes all files that must be accessible by the user's web browser, such as SWF files, JSP pages, HTML pages, Cascading Style Sheets, images, and JavaScript files. You can place these files directly in the web application root directory or in arbitrary subdirectories that do not use the reserved name WEB-INF.
/META-INF	Contains package and extension configuration data.
/WEB-INF	Contains the standard web application deployment descriptor (web.xml) that configures the BlazeDS web application. This directory can also contain a vendor-specific web application deployment descriptor.
/WEB-INF/classes	Contains Java class files and configuration files.
/WEB-INF/flex	Contains BlazeDS configuration files.
/WEB-INF/flex/libs	Contains SWC library files used when compiling with Adobe Flex Builder.
/WEB-INF/flex/locale	Contains localization resource files used when compiling with Flex Builder.
/WEB-INF/lib	Contains BlazeDS JAR files.
/WEB-INF/src	(Optional) Contains Java source code used by the web application.

Accessing a web application

To access a web application and the services provided by BlazeDS, you need the URL and port number associated with the web application. The following table describes how to access each web application assuming that you install BlazeDS with the integrated Tomcat application server.

Note: *If you install BlazeDS into the directory structure of your J2EE application server or servlet container, modify the context root URL based on your development environment.*

Application	Context root URL for Tomcat	Description
Sample application	http://localhost:8400/samples/	A sample web application that includes many BlazeDS examples. To start building your own applications, start by editing these samples. The root directory of the installed web application is *install_root*\samples. For example, if you install BlazeDS Turnky on Microsoft Windows, the root directory is C:\blazeds\tomcat\webapps\samples.
Template application	http://localhost:8400/blazeds/	A fully configured BlazeDS web application that contains no application code. You can use this application as a template to create your own web application. The root directory of the installed web application is *install_root*\blazeds. For example, if you install BlazeDS Turnkey on Microsoft Windows, the root directory is C:\blazeds\tomcat\webapps\blazeds.
Console application	http://localhost:8400/ds-console/	A console application that lets you view information about BlazeDS web applications. The root directory of the installed web application is *install_root*\ds-console. For example, if you install BlazeDS Turnky on Microsoft Windows, the root directory is C:\blazeds\tomcat\webapps\ds-console.

If you install BlazeDS with the integrated Tomcat application server, you can also access the ROOT web application by using the following URL: `http://localhost:8400/`.

Creating a web application

To get started writing BlazeDS applications, you can edit the samples in the samples application, add your application code to the samples application, or add your application code to the empty blazeds application. However, Adobe recommends leaving the blazeds application alone, and instead copying its contents to a new web application. That leaves the blazeds web application empty so that you can use it as the template for creating web applications.

Defining the directory structure for client-side code

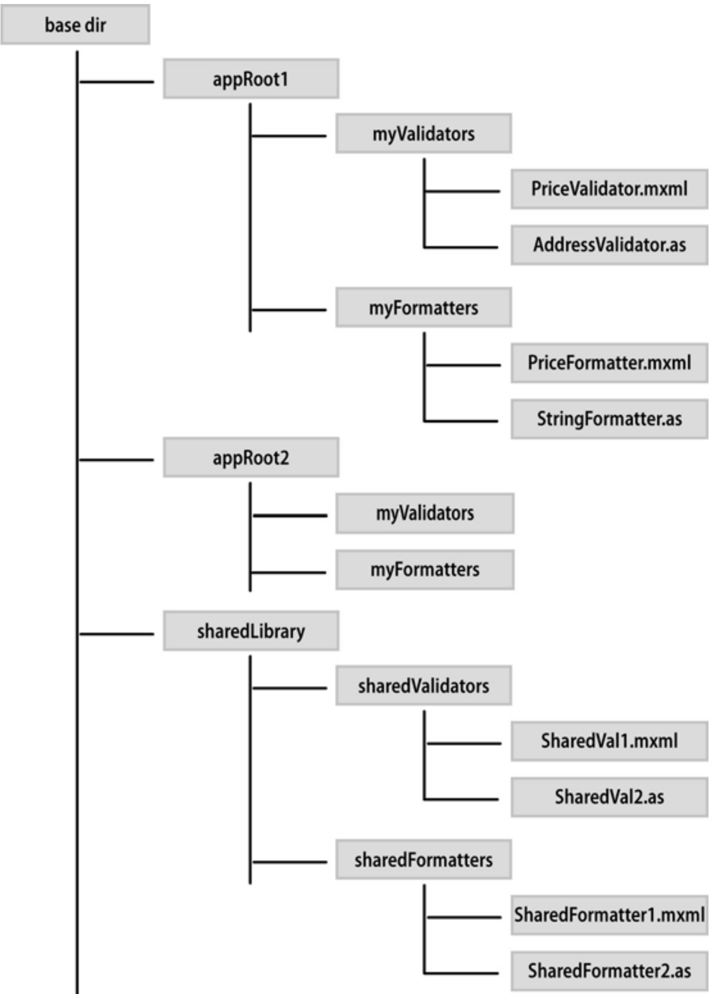
You develop BlazeDS client-side applications in Flex, and compile them in the same way that you compile applications that use the Flex Software Development Kit (SDK). That means you can use the compiler built in to Flex Builder, or the command line compiler, mxmlc, supplied with the Flex SDK.

When you develop applications, you have two choices for how you arrange the directory structure of your application:

- Define a directory structure on your computer outside any BlazeDS web application. Compile the application into a SWF file, and then deploy it, along with any run-time assets, to a BlazeDS web application.
- Define a directory structure in a BlazeDS web application. In this scenario, all of your source code and assets are stored in the web application. When you deploy the application, make sure to deploy only the application SWF file and run-time assets. Otherwise, you run the risk of deploying your source code on a production server.

You define each application in its own directory structure, with the local assets for the application under the root directory. For assets shared across applications, such as image files, you can define a directory that is accessible by all applications.

The following example shows two applications, appRoot1 and appRoot2. Each application has a subdirectory for local MXML and ActionScript components, and can also reference a library of shared components:



Defining the directory structure for server-side code

You develop the server-side part of a BlazeDS application in Java. For example, the client-side RemoteObject component lets you access the methods of server-side Java objects to return data to the client.

You also write Java classes to extend the functionality of the BlazeDS server. For example, a Messaging Service destination references one or more message channels that transport messages, and contains network- and server-related properties. The destination can also reference a *data adapter*, server-side code that lets the destination work with data through a particular type of interface such as a Java object.

When you develop server-side code, you have several choices for how you arrange the directory structure of your application:

- Define a directory structure that corresponds to the package hierarchy of your Java source code outside any BlazeDS web application. Compile the Java code, and then deploy the corresponding class files and JAR files, along with any run-time assets, to a BlazeDS web application.

- Define a directory structure in a BlazeDS web application. In this scenario, all of your source code and assets are stored in the web application. When you deploy the application, make sure to deploy only the class and JAR files. Otherwise, you risk deploying source code on a production server.

The WEB-INF/classes and WEB-INF/lib directories are automatically included in the classpath of the web application. When you deploy your server-side code, place the compiled Java class files in the WEB-INF/classes directory. Place JAR files in the WEB-INF/lib directory.

Running the BlazeDS sample applications

The BlazeDS Turnkey installation includes the samples web application that contains sample applications, including the BlazeDS Test Drive application. The sample applications demonstrate basic capabilities and best practices for developing BlazeDS applications.

The samples use an HSQLDB database that is installed in the *install_root*/sampledb directory. You must start the BlazeDS server and the samples database before you can run the BlazeDS samples. After starting the server and database, access the main sample application page by opening the following URL in a browser:

http://localhost:8400/samples/

The objective of the BlazeDS Test Drive is to give you, in a very short time, an understanding of how BlazeDS works. Access the BlazeDS Test Drive application by opening the following URL in a browser:

http://localhost:8400/samples/testdrive.htm

The client-side source code for the samples is shipped in the samples\WEB-INF\flex-src\flex-src.zip file. To modify the client-side code, extract the flex-src.zip file into the samples directory, and then edit, compile, and deploy the modified examples. Editing the samples makes it easier to get started developing applications because you only have to modify existing code, rather than creating it from scratch.

Extract the client-side source code

1 Open samples\WEB-INF\flex-src\flex-src.zip file.
2 Extract the ZIP file into the samples directory.

   Expanding the ZIP file adds a src directory to each sample in the samples directory. For example, the source code for the chat example, Chat.mxml, is written to the directory samples\testdrive-chat\src.

The server-side source code for these examples is shipped in the samples\WEB-INF\src\flex\samples directory. These source files are not zipped, but shipped in an expanded directory structure. To modify the server-side code you can edit and compile it in that directory structure, and then copy it to the samples directory to deploy it.

For more information on the sample applications, see “Running the BlazeDS sample applications” on page 14.

Run the sample applications

1 Change directory to *install_root*/sampledb.
2 Start the samples database by using the following command:

```
startdb
```

   You can stop the database by using the command:

```
stopdb
```

3 Start BlazeDS.

   How you start BlazeDS depends on your system.
4 Open the following URL in a browser:

http://localhost:8400/samples/

Building your client-side application

You write the client-side part of a BlazeDS application in Flex, and then use Flex Builder or the mxmlc command line compiler to compile it.

Note: *This content assumes you have a BlazeDS Turnkey installation. That installation includes all the files necessary to complete these instructions.*

Before you begin

Before you begin to develop your client-side code, determine the files required to perform the compilation. Ensure that you configured your Flex installation to compile SWF files for BlazeDS applications.

Note: *When you compile an application using mxmlc, by default the compiler references the flex-config.xml configuration file, which specifies to include the libs/player directory in the library path for Flash Player. When you compile an application for AIR, use the* `load-config` *option to the mxmlc compiler to specify the air-config.xml file, which specifies to include the libs/air directory in the library path.*

Unzip the Flex 3 SDK

Unzip *install_root*/resources/flex_sdk/flex_sdk_3.zip to *install_root*/resources/flex_sdk, where *install_root* is the BlazeDS installation directory. For example, the default value of *install_root* is c:\blazeds on Microsoft Windows.

Specifying the services-config.xml file in a compilation

When you compile your Flex application, you typically specify the services-config.xml configuration file to the compiler. This file defines the channel URLs that the client-side Flex application uses to communicate with the BlazeDS server. Then the channel URLs are compiled into the resultant SWF file.

Both client-side and server-side code use the services-config.xml configuration file. If you change anything in services-config.xml, you usually have to recompile your client-side applications and restart your server-side application for the changes to take effect.

In Flex Builder, the appropriate services-config.xml file is included automatically based on the BlazeDS web application that you specified in the configuration of your Flex Builder project. When you use the mxmlc compiler, use the `services` option to specify the location of the file.

Note: *You can also create channel definitions at run time in ActionScript. In that case, you might be able to omit the reference to the services-config.xml configuration file from the compiler. For more information, see* *"Run-time configuration" on page 174*.

Specifying the context root in a compilation

The services-config.xml configuration file typically uses the `context.root` token to specify the context root of a web application. At compile time, you use the compiler `context-root` option to specify that information.

During a compilation, Flex Builder automatically sets the value of the `context.root` token based on the BlazeDS web application that you specified in the configuration of your project. When you use the mxmlc compiler, use the `context-root` option to set it.

Using Flex Builder to compile client-side code

Adobe Flex Builder is an integrated development environment (IDE) for developing applications that use the Adobe Flex framework, MXML, Adobe Flash Player 9, AIR 1.0, ActionScript 3.0, BlazeDS, and the Flex Charting components.

Flex Builder is built on top of Eclipse, an open-source IDE. It runs on Microsoft Windows, Apple Mac OS X, and Linux, and is available in several versions. Installation configuration options let you install Flex Builder as a plug-in to an existing Eclipse workbench installation, or to install it as a stand-alone application.

For more information, see *Using Adobe Flex Builder 3*.

Using stand-alone or plug-in configuration of Flex Builder

The Flex Builder installer provides the following two configuration options:

Plug-in configuration This configuration is for users who already use the Eclipse workbench, who already develop in Java, or who want to add the Flex Builder plug-ins to their toolkit of Eclipse plug-ins. Because Eclipse is an open, extensible platform, hundreds of plug-ins are available for many different development purposes.

Stand-alone configuration This configuration is a customized packaging of Eclipse and the Flex Builder plug-in created specifically for developing Flex and ActionScript applications. The stand-alone configuration is ideal for new users and users who intend to develop only Flex and ActionScript applications.

Both configurations provide the same functionality. You select the configuration when you install Flex Builder.

Most BlazeDS developers choose to use the Eclipse plug-in configuration. Then they develop the Java code that runs on the server in the same IDE that they use to develop the MXML and ActionScript code for the client Flex application.

***Note:** The stand-alone configuration of Flex Builder does not contain tools to edit Java code, however, you can install them. Select Help > Software Updates > Find and Install menu command to open the Install/Update dialog box. Then select Search For New Features To Install. In the results, select Europa Discovery Site, and then select the Java Development package to install.*

If you aren't sure which configuration to use, follow these guidelines:

- If you already use and have Eclipse 3.11 (or later) installed, select the plug-in configuration. On Macintosh, Eclipse 3.2 is the earliest version.
- If you don't have Eclipse installed and your primary focus is on developing Flex and ActionScript applications, select the stand-alone configuration. This configuration also lets you install other Eclipse plug-ins, so you can expand the scope of your development work in the future.

Add the Flex SDK to Flex Builder

Before you can build your first BlazeDS application in Flex Builder, add the Flex SDK to Flex Builder. You perform this procedure only once.

1. Start Flex Builder.
2. Select Window > Preferences.
3. Select Flex > Installed Flex SDKs.
4. Click Add.
5. Specify *install_root*/resources/flex_sdk for the Flex SDK location in the Add Flex SDK dialog box, where *install_root* is the BlazeDS installation directory. For example, the default value of *install_root* is c:\blazeds on Microsoft Windows.
6. Specify BlazeDS as the Flex SDK name, and click OK.
7. (Optional) Select the BlazeDS entry to make it the default Flex SDK for new Flex projects.

8 Click OK.

Create a Flex Builder project

Use this procedure to create a Flex Builder project to edit one of the samples shipped with the Test Drive application. The procedure for creating and configuring a new project is almost the same as the following procedure.

For more information on the Test Drive application, see "Running the BlazeDS sample applications" on page 14.

1 Start Flex Builder.

2 Select File > New > Flex Project.

3 Enter a project name. You are editing an existing application, so use the exact name of the sample folder: **testdrive-chat**.

Note: *If you are creating an empty project, you can name it anything that you want.*

4 If you unzipped flex-src.zip in the samples directory, deselect the Use Default Location option, and specify the directory as C:\blazeds\tomcat\webapps\samples\testdrive-chat, or wherever you unzipped the file on your computer.

Note: *By default, Flex Builder creates the project directory based on the project name and operating system. For example, if you are using the plug-in configuration of Flex Builder on Microsoft Windows, the default project directory is C:\Documents and Settings*\USER_NAME*workspace*\PROJECT_NAME.

5 Select the application type as Web application (runs in Flash Player) to configure the application to run in the browser as a Flash Player application.

If you are creating an AIR application, select Desktop Application (Runs In Adobe AIR). However, make sure that you do not have any server tokens in URLs in the configuration files. In the web application that ships with BlazeDS, server tokens are used in the channel endpoint URLs in the WEB-INF/flex/services-config.xml file, as the following example shows:

```
<endpoint
url="https://{server.name}:{server.port}/{context.root}/messagebroker/streamingamf"
      class="flex.messaging.endpoints.StreamingAMFEndpoint"/>
```

You would change that line to the following:

```
<endpoint url="http://your_server_name:8400/samples/messagebroker/streamingamf"
      class="flex.messaging.endpoints.StreamingAMFEndpoint"/>
```

6 Select J2EE as the Application server type.

7 Select Use Remote Object Access.

8 Select LiveCycle Data Services.

9 If the option is available, deselect Create Combined Java/Flex Project With WTP.

10 Click Next.

11 Deselect Use Default Location For Local LiveCycle Data Services Server.

12 Set the root folder, root URL, and context root of your web application.

The root folder specifies the top-level directory of the web application (the directory that contains the WEB-INF directory). The root URL specifies the URL of the web application, and the context root specifies the root of the web application.

If you are using a BlazeDS Turnkey installation, set the properties as follows:

Root folder: C:\blazeds\tomcat\webapps\samples\

Root URL: http://localhost:8400/samples/

Context root: /samples/

Modify these settings as appropriate if you are not using the Tomcat application server.

13 Make sure that your BlazeDS server is running, and click Validate Configuration to ensure that your project is valid.

14 Verify that the Compile The Application Locally In Flex Builder option is selected.

15 Clear the Output Folder field to set the directory of the compiled SWF file to the main project directory.

By default, Flex Builder writes the compiled SWF file to the bin-debug directory under the main project directory. To use a different output directory, specify it in the Output Folder field.

16 Click Next.

17 Set the name of the main application file to **Chat.mxml**, and click Finish.

Configure your project to use the Flex SDK that ships with BlazeDS

1 Select the project, and select Project > Properties.

2 Select Flex Compiler.

3 Select the Use A Specific SDK option under Flex SDK version.

4 Select BlazeDS, or whatever you named the Flex SDK, when you performed the procedure in "Add the Flex SDK to Flex Builder" on page 16.

5 Click OK.

You can now edit, compile, and deploy an application that uses BlazeDS.

Edit, compile, and deploy a BlazeDS application in Flex Builder

1 Open src\Chat.mxml in your Flex Builder project.

2 Edit Chat.mxml to change the definition of the TextArea control so that it displays an initial text string when the application starts:

```
<mx:TextArea id="log" width="100%" height="100%" text="My edited file!"/>
```

3 Save the file.

When you save the file, Flex Builder automatically compiles it. By default, the resultant SWF file is written to the C:\blazeds\tomcat\webapps\samples\testdrive-chat\bin-debug directory, or the location you set for the Output directory for the project. You should have set the Output directory to the main project directory in the previous procedure.

***Note:** If you write the Chat.SWF file to any directory other than samples\testdrive-chat, deploy the SWF file by copying it to the samples\testdrive-chat directory.*

4 Make sure that you have started the samples database and BlazeDS, as described in "Running the BlazeDS sample applications" on page 14.

5 Select Run > Run to run the application.

You can also request the application in a browser by using the URL http://localhost:8400/samples/testdrive-chat/index.html.

***Note:** By default, Flex Builder creates a SWF file that contains debug information. When you are ready to deploy your final application, meaning one that does not contain debug information, select File > Export > Release Build. For more information, see Using Adobe Flex Builder 3.*

6 Verify that your new text appears in the TextArea control.

Create a linked resource to the BlazeDS configuration files

While working on the client-side of your applications, you often look at or change the BlazeDS configuration files. You can create a linked resource inside a Flex Builder project to make the BlazeDS configuration files easily accessible.

1 Right-click the project name in the project navigation view.

2 Select New > Folder in the pop-up menu.

3 Specify the name of the folder as it will appear in the navigation view. This name can be different from the name of the folder in the file system. For example, type **server-config**.

4 Click the Advanced button.

5 Select the Link To Folder In The File System option.

6 Click the Browse button and select the flex folder under the WEB-INF directory of your web application. For example, on a typical Windows installation that uses the Tomcat integrated server, select: *install_root*/tomcat/webapps/samples/WEB-INF/flex.

7 Click Finish. The BlazeDS configuration files are now available in your Flex Builder project under the server-config folder.

***Note:** If you change anything in services-config.xml, you usually have to recompile your client-side applications and restart your server-side application for the changes to take effect.*

Using mxmlc to compile client-side code

You use the mxmlc command line compiler to create SWF files from MXML, ActionScript, and other source files. Typically, you pass the name of the MXML file that contains an `<mx:Application>` tag to the compiler. The output is a SWF file. The mxmlc compiler ships in the bin directory of the Flex SDK. You run the mxmlc compiler as a shell script and executable file on Windows and UNIX systems. For more information, see *Building and Deploying Adobe Flex 3 Applications*.

The basic syntax of the mxmlc utility is as follows:

```
mxmlc [options] target_file
```

The target file of the compile is required. If you use a space-separated list as part of the options, you can terminate the list with a double hyphen before adding the target file.

```
mxmlc -option arg1 arg2 arg3 -- target_file.mxml
```

To see a list of options for mxmlc, use the `help list` option, as the following example shows:

```
mxmlc -help list
```

To see a list of all options available for mxmlc, including advanced options, use the following command:

```
mxmlc -help list advanced
```

The default output of mxmlc is *filename*.swf, where *filename* is the name of the target file. The default output location is in the same directory as the target, unless you specify an output file and location with the `output` option.

The mxmlc command line compiler does not generate an HTML wrapper. Create your own wrapper to deploy a SWF file that the mxmlc compiler produced. The wrapper embeds the SWF object in the HTML tag. The wrapper includes the `<object>` and `<embed>` tags, and scripts that support Flash Player version detection and history management. For information about creating an HTML wrapper, see *Building and Deploying Adobe Flex 3 Applications*.

Note: *Flex Builder automatically generates an HTML wrapper when you compile your application.*

Compiling BlazeDS applications

Along with the standard options that you use with the mxmlc compiler, use the following options to specify information about your BlazeDS application.

- `services` *filename*

  Specifies the location of the services-config.xml file.

- `context-root` *context-path*

  Sets the value of the context root of the application. This value corresponds to the `{context.root}` token in the services-config.xml file, which is often used in channel definitions. The default value is null.

Edit, compile, and deploy the Chat.mxml file

1 Unzip flex-src.zip in the blazeds\tomcat\webapps\samples directory, as described in "Running the BlazeDS sample applications" on page 14.

2 Open the file C:\blazeds\tomcat\webapps\samples\testdrive-chat\src\Chat.mxml in an editor. Modify this path as necessary based on where you unzipped flex-src.zip.

3 Change the definition of the TextArea control so that it displays an initial text string when the application starts:

```
<mx:TextArea id="log" width="100%" height="100%" text="My edited file!"/>
```

4 Change the directory to C:\blazeds\tomcat\webapps\samples.

5 Use the following command to compile Chat.mxml:

Note: *This command assumes that you added the mxmlc directory to your system path. The default location is install_root\resources\flex_sdk\bin.*

```
mxmlc -strict=true
    -show-actionscript-warnings=true
    -use-network=true
    -services=WEB-INF/flex/services-config.xml
    -context-root=samples
    -output=testdrive-chat/Chat.swf
    testdrive-chat/src/Chat.mxml
```

The compiler writes the Chat.swf file to the samples\testdrive-chat directory.

6 Start the samples database and BlazeDS as described in "Running the BlazeDS sample applications" on page 14.

7 Request the application by using the URL http://localhost:8400/samples/testdrive-chat/index.html.

8 Verify that your new text appears in the TextArea control.

Rather than keeping your source code in your deployment directory, you can set up a separate directory, and then copy Chat.swf to samples\testdrive-chat to deploy it.

Building your server-side application

You write the server-side part of a BlazeDS application in Java, and then use the javac compiler to compile it.

Creating a simple Java class to return data to the client

A common reason to create a server-side Java class is to represent data returned to the client. For example, the client-side RemoteObject component lets you access the methods of server-side Java objects to return data to the client.

The Test Drive sample application contains the Accessing Data Using Remoting sample where the client-side code uses the RemoteObject component to access product data on the server. The Product.java class represents that data. After starting the BlazeDS server and the samples database, view this example by opening the following URL in a browser: http://localhost:8400/samples/testdrive-remoteobject/index.html.

The source code for Product.java is in the *install_root*\samples\WEB-INF\src\flex\samples\product directory. For example, if you are using a BlazeDS Turnkey installation on Microsoft Windows, the directory is C:\blazeds\tomcat\webapps\samples\WEB-INF\src\flex\samples\product.

Modify, compile, and deploy Product.java in the samples web application

1. Start the samples database.
2. Start BlazeDS.
3. View the running example by opening the following URL in a browser:

   http://localhost:8400/samples/testdrive-remoteobject/index.html
4. Click the Get Data button to download data from the server. Notice that the description column contains product descriptions.
5. In an editor, open the file C:\blazeds\tomcat\webapps\samples\WEB-INF\src\flex\samples\product\Product.java. Modify this path as necessary for your installation.
6. Modify the following `getDescription()` method definition so that it always returns the String "My description" rather than the value from the database:

```
public String getDescription() {
    return description;
}
```

   The modified method definition appears as the following:

```
public String getDescription() {
    return "My description.";
}
```

7. Change the directory to samples.
8. Compile Product.java by using the following javac command line:

```
javac -d WEB-INF/classes/ WEB-INF/src/flex/samples/product/Product.java
```

   This command creates the file Product.class, and deploys it to the WEB-INF\classes\flex\samples\product directory.
9. View the running example by opening the following URL in a browser:

   http://localhost:8400/samples/testdrive-remoteobject/index.html
10. Click the Get Data button.

   Notice that the description column now contains the String "My description" for each product.

Creating a Java class that extends a BlazeDS class

As part of developing your server-side code, you can create a custom assembler class, factory class, or other type of Java class that extends the BlazeDS Java class library. For example, a *data adapter* is responsible for updating the persistent data store on the server in a manner appropriate to the specific data store type.

You perform many of the same steps to compile a Java class that extends the BlazeDS Java class library as you do for compiling a simple class. The major difference is to ensure that you include the appropriate BlazeDS JAR files in the classpath of your compilation so that the compiler can locate the appropriate files.

Debugging your application

If you encounter errors in your applications, you can use the debugging tools to perform the following:

- Set and manage breakpoints in your code
- Control application execution by suspending, resuming, and terminating the application
- Step into and over the code statements
- Select critical variables to watch
- Evaluate watch expressions while the application is running

Debugging Flex applications can be as simple as enabling `trace()` statements or as complex as stepping into a source files and running the code, one line at a time. The Flex Builder debugger and the command line debugger, fdb, let you step through and debug ActionScript files used by your Flex applications. For information on how to use the Flex Builder debugger, see *Using Adobe Flex Builder 3*. For more information on the command line debugger, fdb, see *Building and Deploying Adobe Flex 3 Applications*.

Using Flash Debug Player

To use the fdb command line debugger or the Flex Builder debugger, install and configure Flash Debug Player. To determine whether you are running the Flash Debug Player or the standard version of Flash Player, open any Flex application in Flash Player and right-click. If you see the Show Redraw Regions option, you are running Flash Debug Player. For more information about installing Flash Debug Player, see the BlazeDS installation instructions.

Flash Debug Player comes in ActiveX, Plug-in, and stand-alone versions for Microsoft Internet Explorer, Netscape-based browsers, and desktop applications. You can find Flash Debug Player installers in the following locations:

- Flex Builder: *install_dir*/Player/*os_version*
- Flex SDK: *install_dir*/runtimes/player/*os_version*/

Like the standard version of Adobe Flash Player 9, Flash Debug Player runs SWF files in a browser or on the desktop in a stand-alone player. Unlike Flash Player, the Flash Debug Player enables you to do the following:

- Output statements and application errors to the local log file of Flash Debug Player by using the `trace()` method.
- Write data services log messages to the local log file of Flash Debug Player.
- View run-time errors (RTEs).
- Use the fdb command line debugger.
- Use the Flex Builder debugging tool.
- Use the Flex Builder profiling tool.

***Note:** ADL logs `trace()` output from AIR applications.*

Using logging to debug your application

One tool that can help in debugging is the logging mechanism. You can perform server-side and client-side logging of requests and responses.

Client-side logging

For client-side logging, you directly write messages to the log file, or configure the application to write messages generated by Flex to the log file. Flash Debug Player has two primary methods of writing messages to a log file:

- The global `trace()` method. The global `trace()` method prints a String to the log file. Messages can contain checkpoint information to signal that your application reached a specific line of code, or the value of a variable.
- Logging API. The logging API, implemented by the TraceTarget class, provides a layer of functionality on top of the `trace()` method. For example, you can use the logging API to log debug, error, and warning messages generated by Flex while applications execute.

Flash Debug Player sends logging information to the flashlog.txt file. The operating system determines the location of this file, as the following table shows:

Operating system	Location of log file
Windows 95/98/ME/2000/XP	`C:\Documents and Settings\`*username*`\Application Data\Macromedia\Flash Player\Logs`
Windows Vista	`C:\Users\`*username*`\AppData\Roaming\Macromedia\Flash Player\Logs`
Mac OS X	`/Users/`*username*`/Library/Preferences/Macromedia/Flash Player/Logs/`
Linux	`/home/`*username*`/.macromedia/Flash_Player/Logs/`

Use settings in the mm.cfg text file to configure Flash Debug Player for logging. If this file does not exist, you can create it when you first configure Flash Debug Player. The location of this file depends on your operating system. The following table shows where to create the mm.cfg file for several operating systems:

Operating system	Location of mm.cfg file
Mac OS X	`/Library/Application Support/Macromedia`
Windows 95/98/ME	`%HOMEDRIVE%\%HOMEPATH%`
Windows 2000/XP	`C:\Documents and Settings\`*username*
Windows Vista	`C:\Users\`*username*
Linux	`/home/`*username*

The mm.cfg file contains many settings that you can use to control logging. The following sample mm.cfg file enables error reporting and trace logging:

```
ErrorReportingEnable=1
TraceOutputFileEnable=1
```

After you enable reporting and logging, call the `trace()` method to write a String to the flashlog.txt file, as the following example shows:

```
trace("Got to checkpoint 1.");
```

Insert the following MXML line to enable the logging of all Flex-generated debug messages to flashlog.txt:

```
<mx:TraceTarget loglevel="2"/>
```

For information about client-side logging, see *Building and Deploying Adobe Flex 3 Applications*.

Server-side logging

You configure server-side logging in the logging section of the services configuration file, services-config.xml. By default, output is sent to System.out.

You set the logging level to one of the following available levels:

- `All`
- `Debug`

- `Info`
- `Warn`
- `Error`
- `None`

You typically set the server-side logging level to `Debug` to log all debug messages, and also all info, warning, and error messages. The following example shows a logging configuration that uses the `Debug` logging level:

```
<logging>

<!-- You may also use flex.messaging.log.ServletLogTarget. -->
    <target class="flex.messaging.log.ConsoleTarget" level="Debug">
        <properties>
            <prefix>[Flex]</prefix>
            <includeDate>false</includeDate>
            <includeTime>false</includeTime>
            <includeLevel>false</includeLevel>
            <includeCategory>false</includeCategory>
        </properties>
        <filters>
            <pattern>Endpoint</pattern>
            <!--<pattern>Service.*</pattern>-->
            <!--<pattern>Message.*</pattern>-->
        </filters>
    </target>
</logging>
```

For more information, see “Logging” on page 149.

Measuring application performance

As part of preparing your application for final deployment, you can test its performance to look for ways to optimize it. One place to examine performance is in the message processing part of the application. To help you gather this performance information, enable the gathering of message timing and sizing data.

The mechanism for measuring the performance of message processing is disabled by default. When enabled, information regarding message size, server processing time, and network travel time is captured. This information is available to the client that pushed a message to the server, to a client that received a pushed message from the server, or to a client that received an acknowledge message from the server in response a pushed message. A subset of this information is also available for access on the server.

You can use this mechanism across all channel types, including polling and streaming channels, that communicate with the server. However, this mechanism does not work when you make a direct connection to an external server by setting the `useProxy` property to `false` for the HTTPService and WebService tags because it bypasses the BlazeDS Proxy Server.

You use two parameters in a channel definition to enable message processing metrics:

- `<record-message-times>`
- `<record-message-sizes>`

Set these parameters to `true` or `false`; the default value is `false`. You can set the parameters to different values to capture only one type of metric. For example, the following channel definition specifies to capture message timing information, but not message sizing information:

```
<channel-definition id="my-streaming-amf"
    class="mx.messaging.channels.StreamingAMFChannel">
    <endpoint
        url="http://{server.name}:{server.port}/{context.root}/messagebroker/streamingamf"
        class="flex.messaging.endpoints.StreamingAMFEndpoint"/>
    <properties>
```

```
        <record-message-times>true</record-message-times>
        <record-message-sizes>false</record-message-sizes>
    </properties>
</channel-definition>
```

For more information, see “Measuring message processing performance” on page 199.

Deploying your application

Your production environment determines how you deploy your application. One option is to package your application and assets in a BlazeDS web application, and then create a single WAR file that contains the entire web application. You can then deploy the single WAR file on your production server.

Alternatively, you can deploy multiple BlazeDS applications on a single web application. In this case, your production server has an expanded BlazeDS web application to which you add the directories that are required to run your new application.

When you deploy your BlazeDS application in a production environment, ensure that you deploy all the necessary parts of the application, including the following:

- The compiled SWF file that contains your client-side application
- The HTML wrapper generated by Flex Builder or created manually if you use the mxmlc compiler
- The compiled Java class and JAR files that represent your server-side application
- Any run-time assets required by your application
- A BlazeDS web application
- Updated BlazeDS configuration files that contain the necessary information to support your application

Part 2: BlazeDS architecture

BlazeDS architecture 27
Channels and endpoints 38
Managing session data 51
Data serialization 56

Chapter 3: BlazeDS architecture

A BlazeDS application consists of a client application running in a web browser or Adobe AIR and a J2EE web application on the server that the client application communicates with. The client application can be a Flex application or it can be a combination of Flex, HTML, and JavaScript. When you use JavaScript, communication with the BlazeDS server is accomplished using the JavaScript proxy objects provided in the Ajax Client Library. For information about the Ajax Client Library, see “The Ajax client library” on page 179.

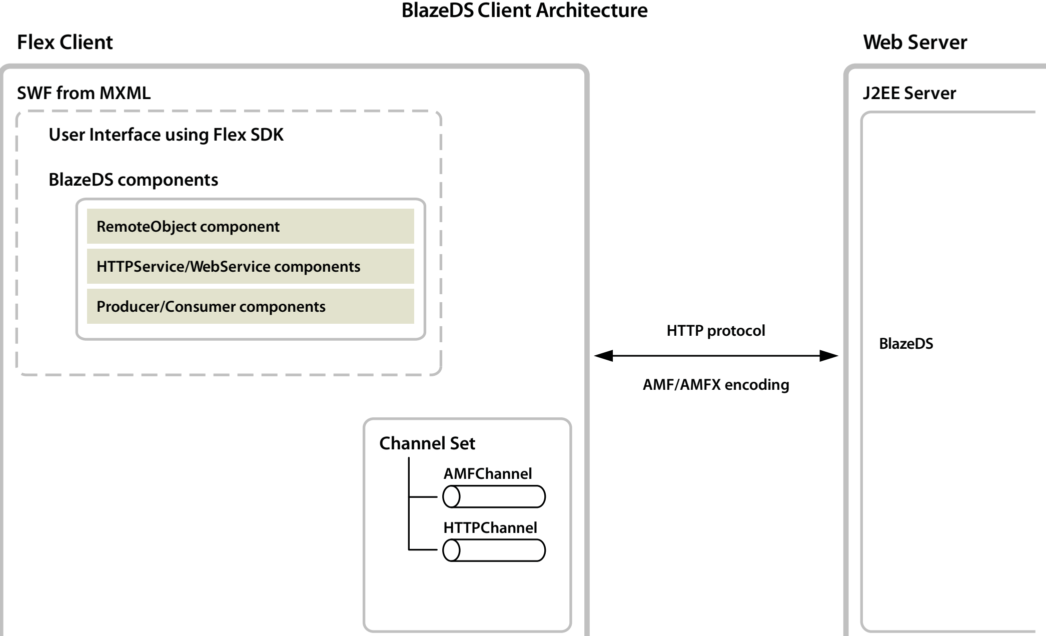
Topics

BlazeDS client architecture 27
BlazeDS server architecture. 29

BlazeDS client architecture

BlazeDS clients use a message-based framework provided by BlazeDS to interact with the server. On the client side of the message-based framework are channels that encapsulate the connection behavior between the Flex client and the BlazeDS server. Channels are grouped together into channel sets that are responsible for channel hunting ad channel failover. For information about client class APIs, see the *ActionScript 3.0 Language Reference*.

The following illustration shows the BlazeDS client architecture:



Flex components

The following Flex components interact with a BlazeDS server:

- RemoteObject

- HTTPService
- WebService
- Producer
- Consumer

All of these components are included in the Flex SDK in the rpc.swc component library.

Although the RemoteObject, Producer, and Consumer components are included with the Flex SDK, they require a server that can interpret the messages that they send. The BlazeDS and LiveCycle Data Services ES servers are two examples of such servers. A Flex application can also make direct HTTP service or web service calls to remote servers without BlazeDS in the middle tier. However, going through the BlazeDS Proxy Service is beneficial for several reasons; for more information, see "Using HTTP and web services" on page 73.

Client-side components communicate with services on the BlazeDS server by sending and receiving messages of the correct type. For more information about messages, see "Messages" on page 28.

Channels and channel sets

A Flex component uses a channel to communicate with a BlazeDS server. A channel set contains channels; its primary function is to provide connectivity between the Flex client and the BlazeDS server. A channel set contains channels ordered by preference. The Flex component tries to connect to the first channel in the channel set and in the case where a connection cannot be established falls back to the next channel in the list. The Flex component continues to go through the list of channels in the order in which they are specified until a connection can be established over one of the channels or the list of channels is exhausted.

Channels encapsulate the connection behavior between the Flex components and the BlazeDS server. Conceptually, channels are a level below the Flex components and they handle the communication between the Flex client and the BlazeDS server. They communicate with their corresponding endpoints on the BlazeDS server; for more information about endpoints, see "Endpoints" on page 29.

Flex clients can use different channel types such as the AMFChannel and HTTPChannel. Channel selection depends on a number of factors, including the type of application you are building. If non-binary data transfer is required, you would use the HTTPChannel, which uses a non-binary format called AMFX (AMF in XML). For more information about channels, see "Channels and endpoints" on page 38.

Messages

All communication between Flex client components and BlazeDS is performed with messages. Flex components use several message types to communicate with their corresponding services in BlazeDS. All messages have client-side (ActionScript) implementations and server-side (Java) implementations because the messages are serialized and deserialized on both the client and the server. You can also create messages directly in Java and have those messages delivered to clients using the server push API.

Some message types, such as AcknowledgeMessage and CommandMessage, are used across different Flex components and BlazeDS services. Other message types are used by specific Flex components and BlazeDSservices. For example, to have a Producer component send a message to subscribed Consumer components, you create a message of type AsyncMessage and pass it to the `send()` method of the Producer component.

In other situations, you do not write code for constructing and sending messages. For example, you simply use a RemoteObject component to call the remote method from the Flex application. The RemoteObject component creates a RemotingMessage to encapsulate the RemoteObject call. In response it receives an AcknowledgeMessage from the server. The AcknowledgeMessage is encapsulated in a ResultEvent in the Flex application.

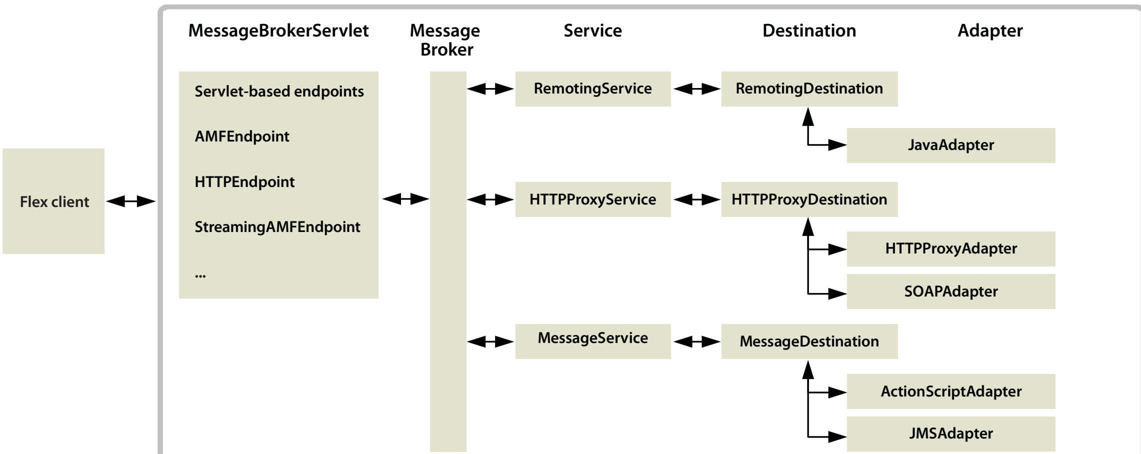
Sometimes you must create a message to send to the server. For example, you could send a message by creating an AsyncMessage and passing it to a Producer.

BlazeDS uses two patterns for sending and receiving messages: the request/reply pattern and the publish/subscribe pattern. RemoteObject, HTTPService, and WebService components use the request/reply message pattern, in which the Flex component makes a request and receives a reply to that request. Producer and Consumer components use the publish/subscribe message pattern. In this pattern, the Producer publishes a message to a destination defined on the BlazeDS server. All Consumers subscribed to that destination receive the message.

BlazeDS server architecture

The BlazeDS server is contained in a J2EE web application. A Flex client makes a request over a channel and the request is routed to an endpoint on the BlazeDS server. From the endpoint, the request is routed through a chain of Java objects that includes the MessageBroker object, a service object, a destination object, and finally an adapter object. The adapter fulfills the request either locally, or by contacting a backend system or a remote server such as Java Message Service (JMS) server.

The following illustration shows the BlazeDS server architecture:



Endpoints

BlazeDS servlet-based endpoints are inside the J2EE servlet container, which means that the servlet handles the I/O and HTTP sessions for the endpoints. Servlet-based endpoints are bootstrapped by the MessageBrokerServlet, which is configured in the web.xml file of the web application. In addition to the MessageBrokerServlet, an HTTP session listener is registered with the J2EE server in the web application's web.xml file so that BlazeDS has HTTP session attribute and binding listener support.

BlazeDSFlex client applications use channels to communicate with BlazeDS endpoints. There is a mapping between the channels on the client and the endpoints on the server. It is important that the channel and the endpoint use the same message format. A channel that uses the AMF message format, such as the AMFChannel, must be paired with an endpoint that also uses the AMF message format, such as the AMFEndpoint. A channel that uses the AMFX message format such as the HTTPChannel cannot be paired with an endpoint that uses the AMF message format. Also, a channel that uses streaming must be paired with an endpoint that uses streaming.

You configure endpoints in the services-config.xml file in the WEB-INF/flex directory of your BlazeDS web application. For more information about endpoints, see “Channels and endpoints” on page 38.

MessageBroker

The MessageBroker is responsible for routing messages to services and is at the core of BlazeDS on the server. After an endpoint initially processes the request, it extracts the message from the request and passes it to the MessageBroker. The MessageBroker inspects the message's destination and passes the message to its intended service. If the destination is protected by a security constraint, the MessageBroker runs the authentication and authorization checks before passing the message along (see "Configuring security" on page 158). You configure the MessageBroker in the services-config.xml file in he WEB-INF/flex directory of your BlazeDS web application.

Services and destinations

Services and destinations are the next links in the message processing chain in the BlazeDS server. The system includes four services and their corresponding destinationsBlazeDS:

- RemotingService and RemotingDestination
- HTPProxyService and HTTPProxyDestination
- MessageService and MessageDestination

Services are the targets of messages from client-side Flex components. Think of destinations as instances of a service configured in a certain way. For example, a RemoteObject component is used on the Flex client to communicate with the RemotingService. In the RemoteObject component, you must specify a destination `id` property that refers to a remoting destination with certain properties, such as the class you want to invoke methods on. The mapping between client-side Flex components and BlazeDS services is as follows:

- HTTPService and WebService communicate with HTTPProxyService/HTTPProxyDestination
- RemoteObject communicates with RemotingService/RemotingDestination
- Producer and Consumer communicate with MessageService/MessageDestination

You can configure services and their destinations in the services-config.xml file, but it is best practice to put them in separate files as follows:

- RemotingService configured in the remoting-config.xml file
- HTTPProxyService configured in the proxy-config.xml file
- MessageService configured in the messaging-config.xml file

For more information on RPC services (HTTPProxy Service and RemotingService) and MessageService, see the following topics:

- "Using HTTP and web services" on page 73
- "Using the Remoting Service" on page 110
- "Using the Messaging Service" on page 122

Adapters and assemblers

Adapters, and optionally assemblers, are the last link in the message processing chain. When a message arrives at the correct destination, it is passed to an adapter that fulfills the request either locally or by contacting a backend system or a remote server such as a JMS server. BlazeDS uses the following mappings between destinations and adapters/assemblers:

- RemotingDestination uses JavaAdapter
- HTTPProxyDestination uses HTTPProxyAdapter or SOAPAdapter
- MessageDestination uses ActionScriptAdapter or JMSAdapter

Adapters and assemblers are configured along with their corresponding destinations in the same configuration files.

Although the BlazeDS server comes with a rich set of adapters and assemblers to communicate with different systems, custom adapters and assemblers can be plugged into the BlazeDS server. Similarly, you do not have to create all destinations in configuration files, but instead you can create them dynamically at server startup or when the server is running; for more information, see “Run-time configuration” on page 174.

For information about the BlazeDS server-side classes, see the Javadoc API documentation.

About configuration files

You configure BlazeDS in the services-config.xml file. The default location of this file is the WEB-INF/flex directory of your BlazeDS web application. You can set this location in the configuration for the MessageBrokerServlet in the WEB-INF/web.xml file.

You can include files that contain service definitions by reference in the services-config.xml file. Your BlazeDS installation includes the Remoting Service, Proxy Service, and Message Service by reference.

The following table describes the typical setup of the configuration files. Commented versions of these files are available in the resources/config directory of the BlazeDS installation.

Filename	Description
services-config.xml	The top-level BlazeDS configuration file. This file usually contains security constraint definitions, channel definitions, and logging settings that each of the services can use. It can contain service definitions inline or include them by reference. Generally, the services are defined in the remoting-config.xml, proxy-config.xml, and messaging-config.xml.
remoting-config.xml	The Remoting Service configuration file, which defines Remoting Service destinations for working with remote objects. For information about configuring the Remoting Service, see “Using the Remoting Service” on page 110.
proxy-config.xml	The Proxy Service configuration file, which defines Proxy Service destinations for working with web services and HTTP services (REST services). For information about configuring the Proxy Service, see “Using HTTP and web services” on page 73.
messaging-config.xml	The Messaging Service configuration file, which defines Messaging Service destinations for performing publish subscribe messaging. For information about configuring the Messaging Service, see “Using the Messaging Service” on page 122.

When you include a file by reference, the content of the referenced file must conform to the appropriate XML structure for the service. The file-path value is relative to the location of the services-config.xml file. The following example shows service definitions included by reference:

```
<services>
    <!-- REMOTING SERVICE -->
    <service-include file-path="remoting-config.xml"/>

    <!-- PROXY SERVICE -->
    <service-include file-path="proxy-config.xml"/>

    <!-- MESSAGE SERVICE -->
    <service-include file-path="messaging-config.xml"/>
</services>
```

Configuration tokens

The configuration files sometimes contain special `{server.name}` and `{server.port}` tokens. These tokens are replaced with server name and port values based on the URL from which the SWF file is served when it is accessed through a web browser from a web server. Similarly, a special `{context.root}` token is replaced with the actual context root of a web application.

Note: *If you use server tokens in a configuration file for an Adobe AIR application and you compile using that file, the application will not be able to connect to the server. You can avoid this issue by configuring channels in ActionScript rather than in a configuration file (see* *“Channels and endpoints” on page 38**).*

You can also use custom run-time tokens in service configuration files; for example, {messaging-channel} and {my.token}. You specify values for these tokens in Java Virtual Machine (JVM) options. The server reads these JVM options to determine what values are defined for them, and replaces the tokens with the specified values. If you have a custom token for which a value cannot be found, an error is thrown. Because {server.name}, {server.port}, and {context.root} are special tokens, no errors occur when these tokens are not specified in JVM options.

How you define JVM options depends on the application server you use. For example, in Apache Tomcat, you can define an environment variable JAVA_OPTS that contains tokens and their values, as this code snippet shows:

```
JAVA_OPTS=-Dmessaging.channel=my-amf -Dmy.token=myValue
```

Configuration elements

The following table describes the XML elements of the services-config.xml file. The root element is the `services-config` element.

XML element			Description
services			Contains definitions of individual data services or references to other XML files that contain service definitions. It is a best practice to use a separate configuration file for each type of standard service. These services include the Proxy Service, Remoting Service, and and Messaging Service. The services element is declared at the top level of the configuration as a child of the root element, services-config. For information about configuring specific types of services, see the following topics: • "Using HTTP and web services" on page 73 • "Using the Remoting Service" on page 110 • "Using the Messaging Service" on page 122
	default-channels		Sets the application-level default channels to use for all services. The default channels are used when a channel is not explicitly referenced in a destination definition. The channels are tried in the order in which they are included in the file. When one is unavailable, the next is tried. Default channels can also be defined individually for each service, in which case the application-level default channels are overwritten by the service level default channels. Application-level default channels are necessary when a dynamic component is created using the run-time configuration feature and no channel set has been defined for the component. In that case, application-level default channels are used to contact the destination. For more information about channels and endpoints, see "Channels and endpoints" on page 38.
	service-include		Specifies the full path to an XML file that contains the configuration elements for a service definition. **Attributes:** • file-path Path to the XML file that contains a service definition.
	service		Contains a service definition. (Optional) You can use the service-include element to include a file that contains a service definition by reference instead of inline in the services-config.xml file. In addition to standard data services, you can define custom bootstrap services here for use with the run-time configuration feature; bootstrap services dynamically create services, destinations, and adapters at server startup. For more information, see "Run-time configuration" on page 174.
		properties	Contains service properties.
		adapters	Contains service adapter definitions that are referenced in a destination to provide specific types of functionality.

XML element			Description
			`adapter-definition` Contains a service adapter definition. Each type of service has its own set of adapters that are relevant to that type of service. An `adapter-definition` has the following attributes: • `id` Identifier of an adapter, which you use to reference the adapter inside a destination definition. • `class` Fully qualified name of the Java class that provides the adapter functionality. • `default` Boolean value that indicates whether this adapter is the default adapter for service destinations. The default adapter is used when you do not explicitly reference an adapter in a destination definition.
		`default-channels`	Contains references to default channels. The default channels are used when a channel is not explicitly referenced in a destination definition. The channels are tried in the order in which they are included in the file. When one is unavailable, the next is tried.
			`channel` Contains a reference to the `id` of a channel definition. A `channel` element contains the following attribute: • `ref` The `id` value of a channel definition. For more information about channels and endpoints, see “Channels and endpoints” on page 38.
		`destination`	Contains a destination definition.
			`adapter` Contains a reference to a service adapter. If this element is omitted, the destination uses the default adapter.
			`properties` Contains destination properties. The properties available depend on the type of service, which the specified service class determines.
			`channels` Contains references to the channels that the service can use for data transport. The channels are tried in the order in which they are included in the file. When one is unavailable, the next is tried. The `channel` child element contains references to the `id` value of a channel. Channels are defined in the `channels` element at the top level of the configuration as a child of the root element, `services-config`.

XML element			Description
			security Contains a reference to a security constraint definition and login command definitions that are used for authentication and authorization. This element can also contain complete security constraint definitions instead of references to security constraints that are defined globally in the top-level security element. For more information, see "Security" on page 156. The security-constraint child element contains references to the id value of a security constraint definition or contains a security constraint definition. **Attributes:** • ref The id value of a security-constraint element defined in the security element at the top level of the services configuration. • id Identifier of a security constraint when you define the actual security constraint in this element. The login-command child element contains a reference to the id value of a login command definition that is used for performing authentication. **Attributes:** • ref The id value of a login command definition.
security			Contains security constraint definitions and login command definitions for authentication and authorization. For more information, see "Security" on page 156.
	security-constraint		Defines a security constraint.
	login-command		Defines a login command that is used for custom authentication. **Attributes:** • class Fully qualified class name of a login command class. • server Application server on which custom authentication is performed. • per-client-authentication You can only set this attribute to true for a custom login command and not an application-server-based login command. Setting it to true allows multiple clients sharing the same session to have distinct authentication states. For example, two windows of the same web browser could authenticate users independently. This attribute is set to false by default.
channels			Contains the definitions of message channels that are used to transport data between the server and clients. For more information about channels and endpoints, see "Channels and endpoints" on page 38.
	channel-definition		Defines a message channel that can be used to transport data. **Attributes:** • id Identifier of the channel definition. • class Fully qualified class name of a channel class.

XML element			Description
		endpoint	Specifies the endpoint URI and the endpoint class of the channel definition. **Attributes:** • uri Endpoint URI. • class Fully qualified name of the channel class used on the client.
		properties	Contains the properties of a channel definition. The properties available depend on the type of channel specified.
clusters			Contains cluster definitions, which configure software clustering across multiple hosts. For more information, see “Clustering” on page 167.
flex-client			
	timeout-minutes		Each Flex application that connects to the server triggers the creation of a FlexClient instance that represents the remote client application. I the value of the timeout-minutes element is left undefined or set to 0 (zero), FlexClient instances on the server are shut down when all associated FlexSessions (corresponding to connections between the client and server) are shut down. If this value is defined, FlexClient instances are kept alive for this amount of idle time. For HTTP connections/sessions, if the remote client application is polling, the FlexClient is kept alive. If the remote client is not polling and a FlexClient instance is idle for this amount of time. it is shut down even if an associated HttpSession is still valid. This is because multiple Flex client applications can share a single HttpSession. A valid HttpSession does not indicate that a specific client application instance is still running.
logging			Contains server-side logging configuration. For more information, see “Logging” on page 149.
	target		Specifies the logging target class and the logging level. **Attributes:** • class Fully qualified logging target class name. • level The logging level.
system			System-wide settings that do not fall into a previous category. In addition to locale information, it also contains redeployment and watch file settings.
	locale		(Optional) Locale string; for example, "en", "de", "fr", and "es" are valid locale strings.
	default-locale		The default locale string. If no default-locale element is provided, a base set of English error messages is used.
	redeploy		Support for web application redeployment when configuration files are updated. This feature works with J2EE application server web application redeployment. The touch-file value is the file used by your application server to force web redeployment. Check the application server to confirm what the touch-file value should be.
		enabled	Boolean value that indicates whether redeployment is enabled.

XML element			Description
		`watch-interval`	Number of seconds to wait before checking for changes to configuration files.
		`watch-file`	A data services configuration file watched for changes. The `watch-file` value must start with `{context.root}` or be an absolute path. The following example uses `{context.root}`: `{context.root}/WEB-INF/flex/data-management-config.xml`
		`touch-file`	The file that the application server uses to force redeployment of a web application. The value of the `touch-file` element must start with `{context.root}` or be an absolute path. Check the application server documentation to determine the `touch-file` value. For Tomcat, the `touch-file` value is the web.xml file, as the following example shows: `{context.root}/WEB-INF/web.xml`

Chapter 4: Channels and endpoints

Channels are the client-side representation of the connection to a service, while endpoints are the server-side representation.

Topics

About channels and endpoints 38
Configuring channels with servlet-based endpoints 43
Channel and endpoint recommendations 49
Using BlazeDS clients and servers behind a firewall 50

About channels and endpoints

Channels are client-side objects that encapsulate the connection behavior between Flex components and the BlazeDS server. Channels communicate with corresponding endpoints on the BlazeDS server. You configure the properties of a channel and its corresponding endpoint in the services-config.xml file.

Configuring channels and endpoints

You configure channels in channel definitions in the services-config.xml file. The channel definition in the following example creates an AMFChannel that communicates with an AMFEndpoint on the server:

```
<channels>
...
<channel-definition id="samples-amf"
        type="mx.messaging.channels.AMFChannel">
        <endpoint url="http://servername:8400/myapp/messagebroker/amf"
            type="flex.messaging.endpoints.AMFEndpoint"/>
</channel-definition>
</channels>
```

The `channel-definition` element specifies the following information:

- id and channel class type of the client-side channel that the Flex client uses to contact the server
- `endpoint` element that contains the URL and endpoint class type of the server-side endpoint
- `properties` element that contains channel and endpoint properties

The endpoint URL is the specific network location that the endpoint is exposed at. The channel uses this value to connect to the endpoint and interact with it. The URL must be unique across all endpoints exposed by the server. The `url` attribute points to the MessageBrokerServlet.

How channels are assigned to a Flex component

Flex components use channel sets, which contain one or more channels, to contact the server. You can automatically or manually create and assign a channel set to a Flex component. The channel allows the component to contact the endpoint, which forwards the request to the destination.

If you compile an MXML file using the MXML compiler option `-services` pointing to the services-config.xml file, the component (RemoteObject, HTTPService, and so on) is automatically assigned a channel set that contains one or more appropriately configured channel instances. The configuration is based on the channel definition assigned to a destination in a configuration file. Alternatively, if you do not compile your application with the `-services` option or want to override the compiled-in behavior, you can manually create a channel set in MXML or Action-Script, populate it with one or more channels, and then assign the channel set to the Flex component.

Application-level default channels are especially important when you want to use dynamically created destinations and you do not want to create and assign channel sets to your Flex components that use the dynamic destination. In that case, application-level default channels are used. For more information, see "Assigning channels and endpoints to a destination" on page 40.

When you compile a Flex client application with the MXML compiler`-services` option, it contains all of the information from the configuration files that is needed for the client to connect to the server.

To manually create channels, you create your own channel set in MXML or ActionScript, add channels to it, and then assign the channel set to a component. This process is common in the following situations:

- You do not compile your MXML file using the `-services` MXML compiler option. This is useful when you do not want to hard code endpoint URLs into your compiled SWF files on the client.
- You want to use a dynamically created destination (the destination is not in the services-config.xml file) with the run-time configuration feature. For more information, see "Run-time configuration" on page 174.
- You want to control in your client code the order of channels that a Flex component uses to connect to the server.

When you create and assign a channel set on the client, the client requires the correct channel type and endpoint URL to contact the server. The client does not specify the endpoint class that handles that request, but there must be a channel definition in the services-config.xml file that specifies the endpoint class to use with the specified endpoint URL.

The following example shows a RemoteObject component that defines a channel set and channel inline in MXML:

```
...
<RemoteObject id="ro" destination="Dest">
    <mx:channelSet>
        <mx:ChannelSet>
            <mx:channels>
                <mx:AMFChannel id="myAmf"
                    uri="http://myserver:2000/myapp/messagebroker/amf"/>
            </mx:channels>
        </mx:ChannelSet>
    </mx:channelSet>
</RemoteObject>
...
```

The following example shows ActionScript code that is equivalent to the MXML code in the previous example:

```
...
    private function run():void {
            ro = new RemoteObject();
            var cs:ChannelSet = new ChannelSet();
            cs.addChannel(new AMFChannel("myAmf",
                "http://servname:2000/eqa/messagebroker/amf"));
```

```
            ro.destination = "Dest";
            ro.channelSet = cs;
    }
...
```

Important: *When you create a channel on the client, you must still include a channel definition that specifies an endpoint class in the services-config.xml file. Otherwise, the message broker cannot pass a Flex client request to an endpoint.*

Assigning channels and endpoints to a destination

Settings in the BlazeDS configuration files determine the channels and endpoints from which a destination can accept messages, invocations, or data, except when you use the run-time configuration feature. The channels and endpoints are determined in one of the following ways:

- If most of the destinations across all services use the same channels, you can define application-level default channels in the services-config.xml file, as the following example shows.

Note: *Using application-level default channels is a best practice whenever possible.*

```
<services-config ...>
...
    <default-channels>
        <channel ref="my-http"/>
        <channel ref="my-amf"/>
    </default-channels>
...
```

In this case, all destinations that do not define channels use a default channel. Destinations can override the default channel setting by specifying their own channels, and services can also override it by specifying their own default channels.

- If most of the destinations in a service use the same channels, you can define service-level default channels, as the following example shows:

```
<service ...>
...
    <default-channels>
        <channel ref="my-http"/>
        <channel ref="my-amf"/>
    </default-channels>
...
```

In this case, all destinations in the service that do not explicitly specify their own channels use the default channel.

- The destination definition can reference a channel inline, as the following example shows:

```
<destination id="sampleVerbose">
    <channels>
        <channel ref="my-secure-amf"/>
    </channels>
...
</destination>
```

Fallback and failover behavior

The primary reason that channels are contained in a channel set is to provide a fallback mechanism from one channel to the next listed in the channel set, and so on, in case the first choice is unable to establish a connection to the server. For example, you could define a channel set that falls back from a StreamingAMFChannel to an AMFChannel with polling enabled to work around network components such as web server connectors, HTTP proxies, or reverse proxies that could buffer chunked responses incorrectly.

In addition to the fallback behavior that the channel set provides, the channel defines a `failoverURIs` property. This property lets you configure a channel in ActionScript that causes failover across this array of endpoint URLs when it tries to connect to its destination. The connection process involves searching for the first channel and trying to connect to it. If the attempt fails, and the channel defines failover URIs, each is attempted before the channel gives up and the channel set searches for the next available channel. If no channel in the set can connect, any pending unsent messages generate faults on the client.

Choosing an endpoint

BlazeDS provides the following servlet-based channel and endpoint combinations. A secure version of each of these channels/endpoints transports data over a secure HTTPS connection. The names of the secure channels and endpoints all start with the text "Secure"; for example, SecureAMFChannel and SecureAMFEndpoint.

Servlet-based channel/endpoint classes	Description
AMFChannel/AMFEndpoint	A simple channel/endpoint that transports data over HTTP in the binary AMF format in an asynchronous call and response model. Use for RPC requests/responses with RPC-style Flex components such as RemoteObject, HTTPService, and WebService. You can also configure a channel that uses this endpoint to repeatedly poll the endpoint for new messages. You can combine polling with a long wait interval for long polling, which handles near real-time communication. For more information, see “Simple channels and endpoints” on page 43.
HTTPChannel/HTTPEndpoint	Provides the same behavior as the AMF channel/endpoint, but transports data in the AMFX format, which is the text-based XML representation of AMF. Transport with this endpoint is not as fast as with the AMFEndpoint because of the overhead of text-based XML. Use when binary AMF is not an option in your environment. For more information, see “Simple channels and endpoints” on page 43.
StreamingAMFChannel/StreamingAMFEndpoint	Streams data in real time over the HTTP protocol in the binary AMF format. Use for real-time data services, such as the Messaging Service where streaming data is critical to performance. For more information, see “Streaming AMF and HTTP channels” on page 47.
StreamingHTTPChannel/StreamingHTTPEndpoint	Provides the same behavior model as the streaming AMF channel/endpoint, but transports data in the AMFX format, which is the text-based XML representation of AMF. Transport with this endpoint is not as fast as with the StreamingAMFEndpoint because of the overhead of text-based XML. Use when binary AMF is not an option in your environment. For more information, see “Streaming AMF and HTTP channels” on page 47.

Choosing a channel

Depending on your application requirements, you can use simple AMF or HTTP channels without polling or with piggybacking, polling, or long polling. You can also use streaming AMF or HTTP channels. The difference between AMF and HTTP channels is that AMF channels transport data in the binary AMF format and HTTP channels transport data in AMFX, the text-based XML representation of AMF. Because AMF channels provide better performance, use an HTTP channel instead of an AMF channel only when you have auditing or compliance requirements that preclude the use of binary data over your network or when you want the contents of messages to be easily readable over the network (on the wire).

Non-polling AMF and HTTP channels

You can use AMF and HTTP channels without polling for remote procedure call (RPC) services, such as remoting service calls, proxied HTTP service calls and web service requests. These scenarios do not require the client to poll for messages or the server to push messages to the client.

Piggybacking on AMF and HTTP channels

The piggybacking feature enables the transport of queued messages along with responses to any messages the client sends to the server over the channel. Piggybacking provides lightweight pseudo polling, where rather than the client channel polling the server on a fixed or adaptive interval, when the client sends a non-command message to the server (using a Producer or RemoteObject object), the server sends any pending data for client messaging or data management subscriptions along with the response to the client message.

Piggybacking can also be used on a channel that has polling enabled but on a wide interval like 5 seconds or 10 seconds or more, in which case the application appears more responsive if the client is sending messages to the server. In this mode, the client sends a poll request along with any messages it sends to the server between its regularly scheduled poll requests. The channel piggybacks a poll request along with the message being sent, and the server piggybacks any pending messages for the client along with the acknowledge response to the client message.

Polling AMF and HTTP channels

AMF and HTTP channels support simple polling mechanisms that clients can use to request messages from the server at set intervals. A polling AMF or HTTP channel is useful when other options such as long polling or streaming channels are not acceptable and also as a fallback channel when a first choice, such as a streaming channel, is unavailable at run time.

Long polling AMF and HTTP channels

You can use AMF and HTTP channels in long polling mode to get pushed messages to the client when the other more efficient and real-time mechanisms are not suitable. This mechanism uses the normal application server HTTP request processing logic and works with typical J2EE deployment architectures.

You can establish long polling for any channel that uses a non-streaming AMF or HTTP endpoint by setting the `polling-enabled`, `polling-interval-millis`, `wait-interval-millis`, and `client-wait-interval-millis` properties in a channel definition; for more information, see “Simple channels and endpoints” on page 43.

Streaming channels

For streaming, you can use streaming AMF or HTTP channels. Streaming AMF and HTTP channels work with streaming AMF or HTTP endpoints.

For more information about endpoints, see “Choosing an endpoint” on page 41.

Configuring channels with servlet-based endpoints

The servlet-based endpoints are part of both BlazeDS and LiveCycle Data Services ES.

Simple channels and endpoints

The AMFEndpoint and HTTPEndpoint are simple servlet-based endpoints. You generally use channels with these endpoints without client polling for RPC service components, which require simple call and response communication with a destination. When working with the Messaging Service, you can use these channels with client polling to constantly poll the destination on the server for new messages, or with long polling to provide near real-time messaging when using a streaming channel is not an option in your network environment.

Property	Description
`polling-enabled`	Optional channel property. Default value is false.
`polling-interval-millis`	Optional channel property. Default value is 3000. This parameter specifies the number of milliseconds the client waits before polling the server again. When `polling-interval-millis` is 0, the client polls as soon as it receives a response from the server with no delay.
`wait-interval-millis`	Optional endpoint property. Default value is `0`. This parameter specifies the number of milliseconds the server poll response thread waits for new messages to arrive when the server has no messages for the client at the time of poll request handling. For this setting to take effect, you must use a nonzero value for the `max-waiting-poll-requests` property. A value of `0` means that server does not wait for new messages for the client and returns an empty acknowledgment as usual. A value of `-1` means that server waits indefinitely until new messages arrive for the client before responding to the client poll request. The recommended value is `60000` milliseconds (one minute).
`client-wait-interval-millis`	Optional channel property. Default value is `0`. Specifies the number of milliseconds the client will wait after it receives a poll response from the server that involved a server wait. A value of `0` means the client uses its configured `polling-interval-millis` value to determine the wait until its next poll. Otherwise, this value overrides the default polling interval of the client. Setting this value to 1 allows clients that poll the server with wait to poll immediately upon receiving a poll response, providing a real-time message stream from the server to the client. Any clients that poll the server and are not serviced with a server wait use the `polling-interval-millis` value.
`max-waiting-poll-requests`	Optional endpoint property. Default value is `0`. Specifies the maximum number of server poll response threads that can be in wait state. When this limit is reached, the subsequent poll requests are treated as having zero `wait-interval-millis`.
`piggybacking-enabled`	Optional endpoint property. Enable to support piggybacking of queued messaging and data management subscription data along with responses to any messages the client sends to the server over this channel.
`login-after-disconnect`	Optional channel property. Default value is `false`. Setting to `true` causes clients to automatically attempt to reauthenticate themselves with the server when they send a message that fails because credentials have been reset due to server session timeout. The failed messages are resent after reauthentication, making the session timeout transparent to the client with respect to authentication.

Property	Description
`flex-client-outbound-queue-processor`	Optional channel property. Use to manage messaging quality of service for subscribers. Every client that subscribes to the server over this channel is assigned a unique instance of the specified outbound queue processor implementation that manages the flow of messages to the client. This can include message conflation, filtering, scheduled delivery and load shedding. You can define configuration properties, and if so, they are used to configure each new queue processor instance that is created. The following example shows how to provide a configuration property: `<flex-client-outbound-queue-processor class="my.company.QoSQueueProcessor"> <properties> <custom-property>5000</custom-property> </properties> </flex-client-outbound-queue-processor>`
`serialization`	Optional serialization properties on endpoint. For more information, see "Configuring AMF serialization on a channel" on page 60.
`connect-timeout-seconds`	Optional channel property. Use to limit the client channel's connect attempt to the specified time interval.
`invalidate-session-on-disconnect`	Optional endpoint property. Disabled by default. If enabled, when a disconnect message is received from a client channel, the corresponding server session is invalidated. If the client is closed without first disconnecting its channel, no disconnect message is sent, and the server session is invalidated when its idle timeout elapses.
`add-no-cache-headers`	Optional endpoint property. Default value is true. HTTPS requests on some browsers do not work when pragma no-cache headers are set. By default, the server adds headers, including pragma no-cache headers to HTTP responses to stop caching by the browsers.

Non-polling AMF and HTTP channels

The simplest types of channels are AMF and HTTP channels in non-polling mode, which operate in a single request-reply pattern. The following example shows AMF and HTTP channel definitions configured for no polling:

```
<!-- Simple AMF -->
<channel-definition id="samples-amf"
    type="mx.messaging.channels.AMFChannel">
    <endpoint url="http://{server.name}:8400/myapp/messagebroker/amf"
        type="flex.messaging.endpoints.AmfEndpoint"/>
</channel-definition>
<!-- Simple secure AMF -->
<channel-definition id="my-secure-amf"
    class="mx.messaging.channels.SecureAMFChannel">
    <endpoint url="https://{server.name}:9100/dev/messagebroker/
        amfsecure" class="flex.messaging.endpoints.SecureAMFEndpoint"/>
</channel-definition>
<!-- Simple HTTP -->
<channel-definition id="my-http"
    class="mx.messaging.channels.HTTPChannel">
    <endpoint url="http://{server.name}:8400/dev/messagebroker/http"
        class="flex.messaging.endpoints.HTTPEndpoint"/>
</channel-definition>
<!-- Simple secure HTTP -->
<channel-definition id="my-secure-http" class="mx.messaging.channels.SecureHTTPChannel">
    <endpoint url=
        "https://{server.name}:9100/dev/messagebroker/
            httpsecure"
        class="flex.messaging.endpoints.SecureHTTPEndpoint"/>
</channel-definition>
```

Polling AMF and HTTP channels

You can use an AMF or HTTP channel in polling mode to repeatedly poll the endpoint to create client-pull message consumers. The interval at which the polling occurs is configurable on the channel. You can also manually poll by calling the `poll()` method of a channel for which polling is enabled; for example, you want set the polling interval to a high number so that the channel does not automatically poll, and call the `poll()` method to poll manually based on an event, such as a button click.

When you use a polling AMF or HTTP channel, you set the `polling` property to `true` in the channel definition. You can also configure the polling interval in the channel definition.

***Note:** You can also use AMF and HTTP channels in long polling mode to get pushed messages to the client when the other more efficient and real-time mechanisms are not suitable. For information about long polling, see* *"Long polling AMF and HTTP channels" on page 45*.

The following example shows AMF and HTTP channel definitions configured for polling:

```
<!-- AMF with polling -->
<channel-definition id="samples-polling-amf"
    type="mx.messaging.channels.AMFChannel">
    <endpoint url="http://{server.name}:8700/dev/messagebroker/amfpolling"
        type="flex.messaging.endpoints.AMFEndpoint"/>
    <properties>
            <polling-enabled>true</polling-enabled>
            <polling-interval-seconds>8</polling-interval-seconds>
    </properties>
</channel-definition>
<!-- HTTP with polling -->
<channel-definition id="samples-polling-http"
    type="mx.messaging.channels.HTTPChannel">
    <endpoint url="http://{server.name}:8700/dev/messagebroker/httppolling"
        type="flex.messaging.endpoints.HTTPEndpoint"/>
    <properties>
            <polling-enabled>true</polling-enabled>
            <polling-interval-seconds>8</polling-interval-seconds>
    </properties>
</channel-definition>
```

***Note:** You can also use secure AMF or HTTP channels in polling mode.*

Long polling AMF and HTTP channels

In the default configuration for a polling AMF or HTTP channel, the endpoint does not wait for messages on the server. When the poll request is received, it checks whether any messages are queued for the polling client and if so, those messages are delivered in the response to the HTTP request. You configure long polling in the same way as polling, but you also must set the `wait-interval-millis`, `max-waiting-poll-requests`, and `client-wait-interval-millis` properties.

To achieve long polling, you set the following properties in the `properties` section of a channel definition in the services-config.xml file:

- `polling-enabled`
- `polling-interval-millis`
- `wait-interval-millis`
- `max-waiting-poll-requests`.
- `client-wait-interval-millis`

Using different settings for these properties results in different behavior. For example, setting the `wait-interval-millis` property to `0` (zero) and setting the `polling-interval-millis` property to a nonzero positive value results in normal polling. Setting the `wait-interval-millis` property to a high value reduces the number of poll messages that the server must process, but the number of request handling threads on the server limits the total number of parked poll requests.

The following example shows AMF and HTTP channel definitions configured for long polling:

```
<!-- Long polling AMF -->
<channel-definition id="my-amf-longpoll" class="mx.messaging.channels.AMFChannel">
    <endpoint
        url="http://servername:8700/contextroot/messagebroker/myamflongpoll"
        class="flex.messaging.endpoints.AMFEndpoint"/>
    <properties>
        <polling-enabled>true</polling-enabled>
        <polling-interval-seconds>0</polling-interval-seconds>
        <wait-interval-millis>60000</wait-interval-millis>
        <client-wait-interval-millis>3000</client-wait-interval-millis>
        <max-waiting-poll-requests>100</max-waiting-poll-requests>
    </properties>
</channel-definition>

<!-- Long polling HTTP -->
<channel-definition id="my-http-longpoll" class="mx.messaging.channels.HTTPChannel">
    <endpoint
        url="http://servername:8700/contextroot/messagebroker/myhttplongpoll"
        class="flex.messaging.endpoints.HTTPEndpoint"/>
    <properties>
        <polling-enabled>true</polling-enabled>
        <polling-interval-seconds>0</polling-interval-seconds>
        <wait-interval-millis>60000</wait-interval-millis>
        <client-wait-interval-millis>3000</client-wait-interval-millis>
        <max-waiting-poll-requests>100</max-waiting-poll-requests>
    </properties>
</channel-definition>
```

***Note:** You can also use secure AMF or HTTP channels in polling mode.*

The caveat for using the `wait-interval-millis` BlazeDS is the utilization of available application server threads. Because this channel ties up one application server request handling thread for each parked poll request, this mechanism can have an impact on server resources and performance. Modern JVMs can typically support about 200 threads comfortably if given enough heap space. Check the maximum thread stack size (often 1 or 2 megabytes per thread) and make sure that you have enough memory and heap space for the number of application server threads you configure.

To ensure that Flex clients using channels with `wait-interval-millis` do not lock up your application server, BlazeDS requires that you set the `max-waiting-poll-requests` property, which specifies the maximum number of waiting connections that BlazeDS should manage. This number must be set to a number smaller than the number of HTTP request threads your application server is configured to use. For example, you configure the application server to have at most 200 threads and allow at most 170 waiting poll requests. This setting would ensure that you have at least 30 application server threads to use for handling other HTTP requests. Your free application server threads should be large enough to maximize parallel opportunities for computation. Applications that are I/O heavy can require a large number of threads to ensure all I/O channels are utilized completely. Multiple threads are useful for the following operations:

- Simultaneously writing responses to clients behind slow network connections
- Executing database queries or updates
- Performing computation on behalf of user requests

Another consideration for using `wait-interval-millis` is that BlazeDS must avoid monopolizing the available connections that the browser allocates for communicating with a server. The HTTP 1.1 specification recommends that browsers allocate at most two connections to the same server when the server supports HTTP 1.1. To avoid using more that one connection from a single browser, BlazeDS allows only one waiting thread for a given application server session at a time. If more than one Flash Player instance within the same browser process attempts to interact with the server using long polling, the server forces them to poll on the default interval with no server wait to avoid busy polling.

Streaming AMF and HTTP channels

The streaming AMF and HTTP channels are HTTP-based streaming channels that the BlazeDS server can use to push updates to clients using a technique called HTTP streaming. These channels give you the option of using standard HTTP for real-time messaging. This capability is supported for HTTP 1.1, but is not available for HTTP 1.0. There are also a number of proxy servers still in use that are not compliant with HTTP 1.1. When using a streaming channel, make sure that the channel has `connect-timeout-seconds` defined and the channel set has a channel to fall back to, such as an AMF polling channel.

Using streaming AMF or HTTP channels/endpoints is like setting a long polling interval on a standard AMF or HTTP channel/endpoint, but the connection is never closed even after the server pushes the data to the client. By keeping a dedicated connection for server updates open, network latency is greatly reduced because the client and the server do not continuously open and close the connection. Unlike polling channels, because streaming channels keep a constant connection open, they can be adversely affected by HTTP connectors, proxies, reverse proxies or other network components that can buffer the response stream.

The following table describes the channel and endpoint configuration properties in the services-config.xml file that are specific to streaming AMF and HTTP channels/endpoints. The table includes the default property values as well as considerations for specific environments and applications.

Property	Description
`connect-timeout-seconds`	Using a streaming connection that passes through an HTTP 1.1 proxy server that incorrectly buffers the response sent back to the client hangs the connection. For this reason, you must set the `connect-timeout-seconds` property to a relatively short timeout period and specify a fallback channel such as an AMF polling channel.
`idle-timeout-minutes`	Optional channel property. Default value is `0`. Specifies the number of minutes that a streaming channel is allowed to remain idle before it is closed. Setting the `idle-timeout-minutes` property to `0` disables the timeout completely, but it is a potential security concern.
`max-streaming-clients`	Optional endpoint property. Default value is `10`. Limits the number of Flex clients that can open a streaming connection to the endpoint. To determine an appropriate value, consider the number of threads available on your application server because each streaming connection open between a FlexClient and the streaming endpoints uses a thread on the server. Use a value that is lower than the maximum number of threads available on the application server. This value is for the number of Flex client application instances, which can each contain one or more MessageAgents (Producer or Consumer components).

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td><code>server-to-client-heartbeat-millis</code></td><td>Optional endpoint property. Default value is <code>5000</code>. Number of milliseconds that the server waits before writing a single byte to the streaming connection to make sure that the client is still available. This is important to determine when a client is no longer available so that its resources associated with the streaming connection can be cleaned up. Note that this functionality keeps the session alive. A non-positive value disables this functionality.</td></tr>
<tr><td><code>user-agent-settings</code></td><td>Optional endpoint property. Default values are as shown in the following example:
<pre><code><user-agent-settings>
    <user-agent
        match-on="MSIE"
        kickstart-bytes= "2048"
        max-streaming-connections-per-session="1"/>
    <user-agent
        match-on="Firefox"
        kickstart-bytes="0"
        max-streaming-connections-per-session="1"/>
</user-agent-settings></code></pre>
User agents are used to customize the streaming endpoint for specific web browsers for the following reasons:

A certain number of bytes must be written before the endpoint can reliably use a streaming connection as specified by the <code>kickstart-bytes</code> attribute.
There is a browser-specific limit to the number of connections allowed per session. In Firefox, the limit is eight connections per session. In Internet Explorer, the limit is two connections allowed per session. Therefore, BlazeDS must limit the number of streaming connections per session on the server to stay below this limit.

By default, BlazeDS uses <code>1</code> as the value for <code>max-streaming-connections-per-session</code>. You can change the values for Internet Explorer and Firefox, and you can add other browser-specific limits by specifying a new <code>user-agent</code> element with a <code>match-on</code> value for a specific browser user agent.</td></tr>
</table>

The following example shows streaming AMF and HTTP channel definitions:

```
<!-- AMF with streaming -->
<channel-definition id="my-amf-stream"
    class="mx.messaging.channels.StreamingAMFChannel">
    <endpoint url="http://servername:2080/myapp/messagebroker/streamingamf"
        class="flex.messaging.endpoints.StreamingAMFEndpoint"/>
</channel-definition>

<!-- Secure AMF with streaming -->
<channel-definition id="my-secure-amf-stream"
    class="mx.messaging.channels.SecureStreamingAMFChannel">
    <endpoint url="http://servername:2080/myapp/messagebroker/securestreamingamf"
        class="flex.messaging.endpoints.SecureStreamingAMFEndpoint"/>
</channel-definition>

<!-- HTTP with streaming -->
<channel-definition id="my-http-stream"
    class="mx.messaging.channels.StreamingHTTPChannel">
    <endpoint url="http://servername:2080/myapp/messagebroker/streaminghttp"
        class="flex.messaging.endpoints.StreamingHTTPEndpoint"/>
</channel-definition>
```

```
<!-- Secure HTTP with streaming -->
<channel-definition id="my-secure-http-stream"
    class="mx.messaging.channels.SecureStreamingHTTPChannel">
    <endpoint url="http://servername:2080/myapp/messagebroker/securestreaminghttp"
        class="flex.messaging.endpoints.SecureStreamingHTTPEndpoint"/>
</channel-definition>
```

Channel and endpoint recommendations

***Note:** The HTTPChannel is the same as the AMFChannel behaviorally, but serializes data in an XML format called AMFX. This channel only exists for customers who require all data sent over the wire to be non-binary for auditing purposes. There is no other reason to use this channel instead of the AMFChannel for RPC-based applications.*

If you are only using remote procedure calls, you can use the AMFChannel. The process for using real-time data push to web clients is not as simple as the RPC scenario. There are a variety of trade-offs, and benefits and disadvantages to consider. Although the answer is not simple, it is prescriptive based on the requirements of your application.

If your application uses both real-time data push as well as RPC, you do not need to use separate channels. All of the channels listed can send RPC invocations to the server. Use a single channel set, possibly containing just a single channel, for all of your RPC, messaging and data management components.

Servlet-based endpoints

Some servlet-based channel/endpoint combinations are preferred over others, depending on your application environment. Each combination is listed here in order of preference.

1. AMFChannel/Endpoint configured for long polling (no fallback needed)

The channel issues polls to the server in the same way as simple polling, but if no data is available to return immediately the server parks the poll request until data arrives for the client or the configured server wait interval elapses.

The client can be configured to issue its next poll immediately following a poll response making this channel configuration feel like real-time communication.

A reasonable server wait time would be one minute. This eliminates the majority of busy polling from clients without being so long that you're keeping server sessions alive indefinitely or running the risk of a network component between the client and server timing out the connection.

Benefits	Disadvantages
Valid HTTP request/response pattern over standard ports that nothing in the network path will have trouble with.	When many messages are being pushed to the client, this configuration has the overhead of a poll round trip for every pushed message or small batch of messages queued between polls. Most applications are not pushing data so frequently for this to be a problem.
	The Servlet API uses blocking IO, so you must define an upper bound for the number of long poll requests parked on the server at any single instant. If your number of clients exceeds this limit, the excess clients devolve to simple polling on the default 3- second interval with no server wait. For example, if you server request handler thread pool has a size of 500, you could set the upper bound for waited polls to 250, 300, or 400 depending on the relative amount of non-poll requests you expect to service concurrently.

2. StreamingAMFChannel/Endpoint (in a channel set followed by the polling AMFChannel for fallback)

Because HTTP connections are not duplex, this channel sends a request to open an HTTP connection between the server and client, over which the server writes an infinite response of pushed messages. This channel uses a separate transient connection from the browser connection pool for each send it issues to the server. The streaming connection is used purely for messages pushed from the server down to the client. Each message is pushed as an HTTP response chunk (HTTP 1.1 Transfer-Encoding: chunked).

Benefits	Disadvantages
No polling overhead associated with pushing messages to the client.	Holding onto the open request on the server and writing an infinite response is not typical HTTP behavior. HTTP proxies that buffer responses before forwarding them can effectively consume the stream. Assign the channel's 'connect-timeout-seconds' property a value of 2 or 3 to detect this and trigger fallback to the next channel in your channel set.
Uses standard HTTP ports so firewalls do not interfere and all requests/responses are HTTP so packet inspecting proxies won't drop the packets.	No support for HTTP 1.0 client. If the client is 1.0, the open request is faulted and the client falls back to the next channel in its channel set.
	The Servlet API uses blocking IO so as with long polling above, you must set a configured upper bound on the number of streaming connections you allow. Clients that exceed this limit are not able to open a streaming connection and will fall back to the next channel in their channel set.

3. AMFChannel/Endpoint with simple polling and piggybacking enabled (no fallback needed)

This configuration is the same as simple polling support but with piggybacking enabled. When the client sends a message to the server between its regularly scheduled poll requests, the channel piggybacks a poll request along with the message being sent, and the server piggybacks any pending messages for the client along with the response.

Benefits	Disadvantages
Valid HTTP request/response pattern over standard ports that nothing in the network path will have trouble with.	Less real-time behavior than long polling or streaming. Requires client interaction with the server to receive pushed data faster than the channel's configured polling interval.
User experience feels more real-time than with simple polling on an interval.	
Does not have thread resource constraints like long polling and streaming due to the blocking IO of the Servlet API.	

Using BlazeDS clients and servers behind a firewall

Because servlet-based endpoints use standard HTTP requests, communicating with clients inside firewalls usually works, as long as the client-side firewall has the necessary ports open. Using the standard HTTP port 80 and HTTPS port 443 is recommended because many firewalls block outbound traffic over non-standard ports.

Chapter 5: Managing session data

Instances of the FlexClient, MessageClient, and FlexSession classes on the BlazeDS server represent a Flex application and its connections to the server. You can use these objects to manage synchronization between a Flex application and the server.

Topics

FlexClient, MessageClient, and FlexSession objects 51
Using the FlexContext class with FlexSession and FlexClient attributes 53
Session life cycle 54

FlexClient, MessageClient, and FlexSession objects

The FlexClient object

Every Flex application, written in MXML or ActionScript, is eventually compiled into a SWF file. When the SWF file connects to the BlazeDS server, a flex.messaging.client.FlexClient object is created to represent that SWF file on the server. SWF files and FlexClient instances have a one-to-one mapping. In this mapping, every FlexClient instance has a unique identifier named `id`, which the BlazeDS server generates. An ActionScript singleton class, mx.messaging.FlexClient, is also created for the Flex application to access its unique FlexClient `id`.

The MessageClient object

If a Flex application contains a Consumer component (flex.messaging.Consumer), the server creates a corresponding flex.messaging.MessageClient instance that represents the subscription state of the Consumer component. Every MessageClient has a unique identifier named `clientId`. The BlazeDS server can automatically generate the `clientId` value, but the Flex application can also set the value in the `Consumer.clientId` property before calling the `Consumer.subscribe()` method.

The FlexSession object

A FlexSession object represents the connection between the Flex application and the BlazeDS server. Its life cycle depends on the underlying protocol, which is determined by the channels and endpoints used on the client and server, respectively.

If an HTTP-based channel, such as AMFChannel or HTTPChannel, is used in the Flex application, the FlexSession on the BlazeDS server is scoped to the browser and wraps an HTTP session. If the HTTP-based channel connects to a servlet-based endpoint, the underlying HTTP session is a J2EE HttpSession object.

The relationship between FlexClient, MessageClient, and FlexSession classes

A FlexClient object can have one or more FlexSession instances associated with it depending on the channels that the Flex application uses. For example, if the Flex application uses one HTTPChannel, one FlexSession represents the HTTP session created for that HTTPChannel on the BlazeDS server.

A FlexSession can also have one or more FlexClients associated with it. For example, when a SWF file that uses an HTTPChannel is opened in two tabs, two FlexClient instances are created in the BlazeDS server (one for each SWF file), but there is only one FlexSession because two tabs share the same underlying HTTP session.

In terms of hierarchy, FlexClient and FlexSession are peers whereas there is a parent-child relationship between FlexClient/FlexSession and MessageClient. A MessageClient is created for every Consumer component in the Flex application. A Consumer must be contained in a single SWF file and it must subscribe over a single channel. Therefore, each MessageClient is associated with exactly one FlexClient and one FlexSession.

If either the FlexClient or the FlexSession is invalidated on the server, it invalidates the MessageClient. This behavior matches the behavior on the client. If you close the SWF file, the client subscription state is invalidated. If you disconnect the channel or it loses connectivity, the Consumer component is unsubscribed.

Event listeners for FlexClient, MessageClient, and FlexSession

The BlazeDS server provides the following set of event listener interfaces that allow you to execute custom business logic as FlexClient, FlexSession, and MessageClient instances are created and destroyed and as their state changes:

Event listener	Description
FlexClientListener	FlexClientListener supports listening for life cycle events for FlexClient instances.
FlexClientAttributeListener	FlexClientAttributeListener supports notification when attributes are added, replaced, or removed from FlexClient instances.
FlexClientBindingListener	FlexClientBindingListener supports notification when the implementing class is bound or unbound as an attribute to a FlexClient instance.
FlexSessionListener	FlexSessionListener supports listening for life cycle events for FlexSession instances.
FlexSessionAttributeListener	FlexSessionAttributeListener supports notification when attributes are added, replaced, or removed from FlexSession instances.
FlexSessionBindingListener	FlexSessionBindingListener supports notification when the implementing class is bound or unbound as an attribute to a FlexSession instance.
MessageClientListener	MessageClientListener supports listening for life cycle events for MessageClient instances representing Consumer subscriptions.

For more information about these classes, see the Javadoc API documentation.

Log categories for FlexClient, MessageClient, and FlexSession classes

The following server-side log categories can be used to track creation, destruction, and other relevant information for FlexClient, MessageClient, and FlexSession:

- Client.FlexClient
- Client.MessageClient
- Endpoint.FlexSession

Using the FlexContext class with FlexSession and FlexClient attributes

The flex.messaging.FlexContext class is a utility class that exposes the current execution context on the BlazeDS server. It provides access to FlexSession and FlexClient instances associated with the current message being processed. It also provides global context by accessing MessageBroker, ServletContext, and ServletConfig instances.

The following example shows a Java class that calls `FlexContext.getHttpRequest()` to get an HTTPServletRequest object and calls `FlexContext.getFlexSession()` to get a FlexSession object. Exposing this class as a remote object makes it accessible to a Flex client application. Place the compiled class in the WEB_INF/classes directory or your BlazeDS web application.

```
package myROPackage;

import flex.messaging.*;
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionRO {

    public HttpServletRequest request;
    public FlexSession session;

    public SessionRO() {
        request = FlexContext.getHttpRequest();
        session = FlexContext.getFlexSession();
    }

    public String getSessionId() throws Exception {
        String s = new String();
        s = (String) session.getId();
        return s;
    }

    public String getHeader(String h) throws Exception {
        String s = new String();
        s = (String) request.getHeader(h);
        return h + "=" + s;
    }
}
```

The following example shows a Remoting Service destination definition that exposes the SessionRO class as a remote object. You add this destination definition to your Remoting Service configuration file.

```
...
<destination id="myRODestination">
    <properties>
        <source>myROPackage.SessionRO</source>
    </properties>
</destination>
...
```

The following example shows an ActionScript snippet for calling the remote object from a Flex client application. You place this code inside a method declaration.

```
...
    ro = new RemoteObject();
    ro.destination = "myRODestination";
    ro.getSessionId.addEventListener("result", getSessionIdResultHandler);
    ro.getSessionId();
...
```

For more information about FlexContext, see the Javadoc API documentation.

Session life cycle

Disconnecting from an HTTP-based channel

Because HTTP is a stateless protocol, when a SWF file is disconnected, notification on the server depends on when the HTTP session times out on the server. When you want a Flex application to notify the BlazeDS server that it is closing, you call the `disconnectAll()` method on the ChannelSet from JavaScript.

In your Flex application, expose a function for JavaScript to call, as the following example shows:

```
<mx:Application creationComplete="init();">
...
<mx:Script>
    <![CDATA[
        import flash.external.ExternalInterface;
        private function init():void
        {
            if (ExternalInterface.available)
            {
                ExternalInterface.addCallback("disconnectAll", disconnectAll);
            }
        }

    // This function will be called by JavaScript
    private function disconnectAll():void
    {
        // Here you get channelSet of your component and call disconnectAll.
        // For example: producer.channelSet.disconnectAll();
    }
```

```
...
</mx:Script>
</mx:Application>
```

Next, in the `body` element of the HTML file that wraps your SWF file, you specify an `onunload()` function, as the following example shows:

```
<body ... onunload="disconnectAll()" ..>
```

Finally, you define the JavaScript `disconnectAll()` function that calls the ActionScript `disconnectAll()` method, as the following example shows:

```
<script langugage='javascript' ..>

    function disconnectAll()
    {
        var answer = confirm("Disconnect All?")
        if (answer)
        {
            // Here you'll need to provide the id of your Flex swf
            document.getElementById("yourFlexSwfId").disconnectAll();

            // And it is important that you have some alert or confirm
            // here to make sure disconnectAll is called before the
            // browser window is closed.
            alert("Disconnected!")
        }
    }
</script>
```

Invalidating an HTTP session

Another complication with using the HTTP protocol is that multiple SWF files share the same HTTP session. When one SWF file disconnects, BlazeDS cannot invalidate the HTTP session. In this scenario, the default behavior on the server is to leave the current session in place for other applications or pages that are using the HTTP session. However, you can use the optional `invalidate-session-on-disconnect` configuration property in a channel definition in the services-config.xml file to invalidate the session, as the following example shows:

```
<channel-definition id="my-amf" class="mx.messaging.channels.AMFChannel">
    <endpoint url="http://servername:port/contextroot/messagebroker/amf"
        class="flex.messaging.endpoints.AMFEndpoint"/>
    <properties>
        <invalidate-session-on-disconnect>true</invalidate-session-on-disconnect>
    </properties>
</channel>
```

Chapter 6: Data serialization

BlazeDS and Adobe® Flex™ provide functionality for serializing data to and from ActionScript objects on the client and Java objects on the server, as well as serializing to and from ActionScript objects on the client and SOAP and XML schema types.

Topics

Serializing between ActionScript and Java 56
Serializing between ActionScript and web services 64

Serializing between ActionScript and Java

BlazeDS and Flex let you serialize data between ActionScript (AMF 3) and Java in both directions.

Converting data from ActionScript to Java

When method parameters send data from a Flex application to a Java object, the data is automatically converted from an ActionScript data type to a Java data type. When BlazeDS searches for a suitable method on the Java object, it uses further, more lenient conversions to find a match.

Simple data types on the client, such as Boolean and String values, typically exactly match a remote API. However, Flex attempts some simple conversions when searching for a suitable method on a Java object.

An ActionScript Array can index entries in two ways. A *strict Array* is one in which all indices are Numbers. An *associative Array* is one in which at least one index is based on a String. It is important to know which type of Array you are sending to the server, because it changes the data type of parameters that are used to invoke a method on a Java object. A *dense Array* is one in which all numeric indices are consecutive, with no gap, starting from 0 (zero). A *sparse Array* is one in which there are gaps between the numeric indices; the Array is treated like an object and the numeric indices become properties that are deserialized into a java.util.Map object to avoid sending many null entries.

The following table lists the supported ActionScript (AMF 3) to Java conversions for simple data types:

ActionScript type (AMF 3)	Deserialization to Java	Supported Java type binding
Array (dense)	java.util.List	java.util.Collection, *Object*[ ] (native array) If the type is an interface, it is mapped to the following interface implementations • List becomes ArrayList • SortedSet becomes TreeSet • Set becomes HashSet • Collection becomes ArrayList A new instance of a custom Collection implementation is bound to that type.
Array (sparse)	java.util.Map	java.util.Map

ActionScript type (AMF 3)	Deserialization to Java	Supported Java type binding
Boolean	java.lang.Boolean	Boolean, boolean, String
String of "true" or "false"		
flash.utils.ByteArray	byte []	
flash.utils.IExternalizable	java.io.Externalizable	
Date	java.util.Date	java.util.Date, java.util.Calendar, java.sql.Timestamp, java.sql.Time, java.sql.Date
	(formatted for Coordinated Universal Time (UTC))	
int/uint	java.lang.Integer	java.lang.Double, java.lang.Long, java.lang.Float, java.lang.Integer, java.lang.Short, java.lang.Byte, java.math.BigDecimal, java.math.BigInteger, String, primitive types of double, long, float, int, short, byte
null	null	primitives
Number	java.lang.Double	java.lang.Double, java.lang.Long, java.lang.Float, java.lang.Integer, java.lang.Short, java.lang.Byte, java.math.BigDecimal, java.math.BigInteger, String, 0 (zero) if null is sent, primitive types of double, long, float, int, short, byte
Object (generic)	java.util.Map	If a Map interface is specified, creates a new java.util.HashMap for java.util.Map and a new java.util.TreeMap for java.util.SortedMap.
String	java.lang.String	java.lang.String, java.lang.Boolean, java.lang.Number, java.math.BigInteger, java.math.BigDecimal, char[], enum, any primitive number type
typed Object	typed Object	typed Object
	when you use `[RemoteClass]` metadata tag that specifies remote class name. Bean type must have a public no args constructor.	
undefined	null	null for Object, default values for primitives
XML	org.w3c.dom.Document	org.w3c.dom.Document
XMLDocument	org.w3c.dom.Document	org.w3c.dom.Document
(legacy XML type)		You can enable legacy XML support for the XMLDocument type on any channel defined in the services-config.xml file. This setting is only important for sending data from the server back to the client; it controls how org.w3c.dom.Document instances are sent to ActionScript. For more information, see "Configuring AMF serialization on a channel" on page 60.

Primitive values cannot be set to null in Java. When passing Boolean and Number values from the client to a Java object, Flex interprets null values as the default values for primitive types; for example, 0 for double, float, long, int, short, byte, \u0000 for char, and `false` for boolean. Only primitive Java types get default values.

BlazeDS handles java.lang.Throwable objects like any other typed object. They are processed with rules that look for public fields and bean properties, and typed objects are returned to the client. The rules are like normal bean rules except they look for getters for read-only properties. This lets you get more information from a Java exception. If you require legacy behavior for Throwable objects, you can set the `legacy-throwable` property to `true` on a channel; for more information, see "Configuring AMF serialization on a channel" on page 60.

You can pass strict Arrays as parameters to methods that expect an implementation of the java.util.Collection or native Java Array APIs.

A Java Collection can contain any number of Object types, whereas a Java Array requires that entries are the same type (for example, java.lang.Object[ ], and int[ ]).

BlazeDS also converts ActionScript strict Arrays to appropriate implementations for common Collection API interfaces. For example, if an ActionScript strict Array is sent to the Java object method `public void addProducts(java.util.Set products)`, BlazeDS converts it to a java.util.HashSet instance before passing it as a parameter, because HashSet is a suitable implementation of the java.util.Set interface. Similarly, BlazeDS passes an instance of java.util.TreeSet to parameters typed with the java.util.SortedSet interface.

BlazeDS passes an instance of java.util.ArrayList to parameters typed with the java.util.List interface and any other interface that extends java.util.Collection. Then these types are sent back to the client as mx.collections.ArrayCollection instances. If you require normal ActionScript Arrays sent back to the client, you must set the `legacy-collection` element to `true` in the `serialization` section of a channel-definition's properties; for more information, see “Configuring AMF serialization on a channel” on page 60.

Explicitly mapping ActionScript and Java objects

For Java objects that BlazeDS does not handle implicitly, values found in public bean properties with get/set methods and public variables are sent to the client as properties on an Object. Private properties, constants, static properties, and read-only properties, and so on, are not serialized. For ActionScript objects, public properties defined with the get/set accessors and public variables are sent to the server.

BlazeDS uses the standard Java class, java.beans.Introspector, to get property descriptors for a Java bean class. It also uses reflection to gather public fields on a class. It uses bean properties in preference to fields. The Java and ActionScript property names should match. Native Flash Player code determines how ActionScript classes are introspected on the client.

In the ActionScript class, you use the `[RemoteClass(alias=" ")]` metadata tag to create an ActionScript object that maps directly to the Java object. The ActionScript class to which data is converted must be used or referenced in the MXML file for it to be linked into the SWF file and available at run time. A good way to do this is by casting the result object, as the following example shows:

```
var result:MyClass = MyClass(event.result);
```

The class itself should use strongly typed references so that its dependencies are also linked.

The following example shows the source code for an ActionScript class that uses the `[RemoteClass(alias=" ")]` metadata tag:

```
package samples.contact {
    [Bindable]
    [RemoteClass(alias="samples.contact.Contact")]
    public class Contact {
        public var contactId:int;

        public var firstName:String;

        public var lastName:String;

        public var address:String;

        public var city:String;

        public var state:String;

        public var zip:String;
    }
}
```

You can use the [RemoteClass] metadata tag without an alias if you do not map to a Java object on the server, but you do send back your object type from the server. Your ActionScript object is serialized to a special Map object when it is sent to the server, but the object returned from the server to the clients is your original ActionScript type.

To restrict a specific property from being sent to the server from an ActionScript class, use the [Transient] metadata tag above the declaration of that property in the ActionScript class.

Converting data from Java to ActionScript

An object returned from a Java method is converted from Java to ActionScript. BlazeDS also handles objects found within objects. BlazeDS implicitly handles the Java data types in the following table.

Java type	ActionScript type (AMF 3)
enum (JDK 1.5)	String
java.lang.String	String
java.lang.Boolean, boolean	Boolean
java.lang.Integer, int	int
	If value < 0xF0000000 || value > 0x0FFFFFFF, the value is promoted to Number due to AMF encoding requirements.
java.lang.Short, short	int
	If i < 0xF0000000 || i > 0x0FFFFFFF, the value is promoted to Number.
java.lang.Byte, byte[]	int
	If i < 0xF0000000 || i > 0x0FFFFFFF, the value is promoted to Number.
java.lang.Byte[]	flash.utils.ByteArray
java.lang.Double, double	Number
java.lang.Long, long	Number
java.lang.Float, float	Number
java.lang.Character, char	String
java.lang.Character[], char[]	String
java. math.BigInteger	String
java.math.BigDecimal	String
java.util.Calendar	Date
	Dates are sent in the Coordinated Universal Time (UTC) time zone. Clients and servers must adjust time accordingly for time zones.
java.util.Date	Date
	Dates are sent in the UTC time zone. Clients and servers must adjust time accordingly for time zones.
java.util.Collection (for example, java.util.ArrayList)	mx.collections.ArrayCollection
java.lang.Object[]	Array
java.util.Map	Object (untyped). For example, a java.util.Map[] is converted to an Array (of Objects).
java.util.Dictionary	Object (untyped)
org.w3c.dom.Document	XML object

Java type	ActionScript type (AMF 3)
null	null
java.lang.Object (other than previously listed types)	Typed Object Objects are serialized using Java bean introspection rules and also include public fields. Fields that are static, transient, or nonpublic, as well as bean properties that are nonpublic or static, are excluded.

***Note:** You can enable legacy XML support for the flash.xml.XMLDocument type on any channel that is defined in the services-config.xml file.*

***Note:** In Flex 1.5, java.util.Map was sent as an associative or ECMA Array. This is no longer a recommended practice. You can enable legacy Map support to associative Arrays, but Adobe recommends against doing this.*

Configuring AMF serialization on a channel

You can support legacy AMF type serialization used in earlier versions of Flex and configure other serialization properties in channel definitions in the services-config.xml file.

The following table describes the properties that you can set in the `<serialization>` element of a channel definition:

Property	Description
`enable-small-messages`	Default value is true. If enabled, messages are sent using an alternative smaller form if one is available and the endpoint supports it.
`ignore-property-errors`	Default value is true. Determines if the endpoint should throw an error when an incoming client object has unexpected properties that cannot be set on the server object.
`log-property-errors`	Default value is false. When `true`, unexpected property errors are logged.
`legacy-collection`	Default value is false. When `true`, instances of java.util.Collection are returned as ActionScript Arrays. When `false`, instance of java.util.Collection are returned as mx.collections.ArrayCollection.
`legacy-map`	Default value is false. When `true`, java.util.Map instances are serialized as an ECMA Array or associative array instead of an anonymous Object.
`legacy-xml`	Default value is false. When `true`, org.w3c.dom.Document instances are serialized as flash.xml.XMLDocument instances instead of intrinsic XML (E4X capable) instances.
`legacy-throwable`	Default value is false. When `true`, java.lang.Throwable instances are serialized as AMF status-info objects (instead of normal bean serialization, including read-only properties).
legacy-externalizable	Default value is false. When true, java.io.Externalizable types (that extend standard Java classes like Date, Number, String) are not serialized as custom objects (for example, MyDate is serialized as Date instead of MyDate). Note that this setting overwrites any other legacy settings. For example, if `legacy-collection` is true but the collection implements java.io.Externalizable, the collection is returned as custom object without taking the `legacy-collection` value into account.
`type-marshaller`	Specifies an implementation of flex.messaging.io.TypeMarshaller that translates an object into an instance of a desired class. Used when invoking a Java method or populating a Java instance and the type of the input object from deserialization (for example, an ActionScript anonymous Object is always deserialized as a java.util.HashMap) doesn't match the destination API (for example, java.util.SortedMap). Thus, the type can be marshalled into the desired type.

Property	Description
`restore-references`	Default value is false. An advanced switch to make the deserializer keep track of object references when a type translation has to be made; for example, when an anonymous Object is sent for a property of type java.util.SortedMap, the Object is first deserialized to a java.util.Map as normal, and then translated to a suitable implementation of SortedMap (such as java.util.TreeMap). If other objects pointed to the same anonymous Object in an object graph, this setting restores those references instead of creating SortedMap implementations everywhere. Notice that setting this property to `true` can slow down performance significantly for large amounts of data.
`instantiate-types`	Default value is true. Advanced switch that when set to `false` stops the deserializer from creating instances of strongly typed objects and instead retains the type information and deserializes the raw properties in a Map implementation, specifically flex.messaging.io.ASObject. Notice that any classes under flex.* package are always instantiated.

Using custom serialization between ActionScript and Java

If the standard mechanisms for serializing and deserializing data between ActionScript on the client and Java on the server do not meet your needs, you can write your own serialization scheme. You implement the ActionScript-based flash.utils.IExternalizable interface on the client and the corresponding Java-based java.io.Externalizable interface on the server.

A typical reason to use custom serialization is to avoid passing all of the properties of either the client-side or server-side representation of an object across the network tier. When you implement custom serialization, you can code your classes so that specific properties that are client-only or server-only are not passed over the wire. When you use the standard serialization scheme, all public properties are passed back and forth between the client and the server.

On the client side, the identity of a class that implements the flash.utils.IExternalizable interface is written in the serialization stream. The class serializes and reconstructs the state of its instances. The class implements the `writeExternal()` and `readExternal()` methods of the IExternalizable interface to get control over the contents and format of the serialization stream, but not the class name or type, for an object and its supertypes. These methods supersede the native AMF serialization behavior. These methods must be symmetrical with their remote counterpart to save the class's state.

On the server side, a Java class that implements the java.io.Externalizable interface performs functionality that is analogous to an ActionScript class that implements the flash.utils.IExternalizable interface.

***Note:** You should not use types that implement the IExternalizable interface with the HTTPChannel if precise by-reference serialization is required. When you do this, references between recurring objects are lost and appear to be cloned at the endpoint.*

The following example shows the complete source code for the client (ActionScript) version of a Product class that maps to a Java-based Product class on the server. The client Product implements the IExternalizable interface and the server Product implements the Externalizable interface.

```
// Product.as
package samples.externalizable {

import flash.utils.IExternalizable;
import flash.utils.IDataInput;
import flash.utils.IDataOutput;

[RemoteClass(alias="samples.externalizable.Product")]
public class Product implements IExternalizable {
    public function Product(name:String=null) {
        this.name = name;
    }
```

```
    public var id:int;
    public var name:String;
    public var properties:Object;
    public var price:Number;

    public function readExternal(input:IDataInput):void {
        name = input.readObject() as String;
        properties = input.readObject();
        price = input.readFloat();
    }

    public function writeExternal(output:IDataOutput):void {
        output.writeObject(name);
        output.writeObject(properties);
        output.writeFloat(price);
    }
}
}
```

The client Product uses two kinds of serialization. It uses the standard serialization that is compatible with the java.io.Externalizable interface and AMF 3 serialization. The following example shows the `writeExternal()` method of the client Product, which uses both types of serialization:

```
public function writeExternal(output:IDataOutput):void {
    output.writeObject(name);
    output.writeObject(properties);
    output.writeFloat(price);
}
```

As the following example shows, the `writeExternal()` method of the server Product is almost identical to the client version of this method:

```
public void writeExternal(ObjectOutput out) throws IOException {
    out.writeObject(name);
    out.writeObject(properties);
    out.writeFloat(price);

}
```

In the client Product's `writeExternal()` method, the `flash.utils.IDataOutput.writeFloat()` method is an example of standard serialization methods that meet the specifications for the Java `java.io.DataInput.readFloat()` methods for working with primitive types. This method sends the `price` property, which is a Float, to the server Product.

The examples of AMF 3 serialization in the client Product's `writeExternal()` method is the call to the `flash.utils.IDataOutput.writeObject()` method, which maps to the `java.io.ObjectInput.readObject()` method call in the server Product's `readExternal()` method. The `flash.utils.IDataOutput.writeObject()` method sends the `properties` property, which is an Object, and the name property, which is a String, to the server Product. This is possible because the AMFChannel endpoint has an implementation of the java.io.ObjectInput interface that expects data sent from the `writeObject()` method to be formatted as AMF 3.

In turn, when the `readObject()` method is called in the server Product's `readExternal()` method, it uses AMF 3 deserialization; this is why the ActionScript version of the `properties` value is assumed to be of type Map and `name` is assumed to be of type String.

The following example shows the complete source of the server Product class:

```
// Product.java
package samples.externalizable;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.util.Map;

/**
* This Externalizable class requires that clients sending and
* receiving instances of this type adhere to the data format
* required for serialization.
*/
public class Product implements Externalizable {
    private String inventoryId;
    public String name;
    public Map properties;
    public float price;

    public Product()
    {
    }

        /**
        * Local identity used to track third party inventory. This property is
        * not sent to the client because it is server-specific.
        * The identity must start with an 'X'.
        */
        public String getInventoryId() {
            return inventoryId;
        }

        public void setInventoryId(String inventoryId) {
            if (inventoryId != null && inventoryId.startsWith("X"))
            {
                this.inventoryId = inventoryId;
            }
            else
            {
                throw new IllegalArgumentException("3rd party product
                inventory identities must start with 'X'");
            }
        }

        /**
         * Deserializes the client state of an instance of ThirdPartyProxy
         * by reading in String for the name, a Map of properties
         * for the description, and
         * a floating point integer (single precision) for the price.
         */
        public void readExternal(ObjectInput in) throws IOException,
            ClassNotFoundException {
            // Read in the server properties from the client representation.
            name = (String)in.readObject();
            properties = (Map)in.readObject();
            price = in.readFloat();
            setInventoryId(lookupInventoryId(name, price));
        }
```

```
        /**
         * Serializes the server state of an instance of ThirdPartyProxy
         * by sending a String for the name, a Map of properties
         * String for the description, and a floating point
         * integer (single precision) for the price. Notice that the inventory
         * identifier is not sent to external clients.
         */
        public void writeExternal(ObjectOutput out) throws IOException {
            // Write out the client properties from the server representation
            out.writeObject(name);
            out.writeObject(properties);
            out.writeFloat(price);
        }

        private static String lookupInventoryId(String name, float price) {
            String inventoryId = "X" + name + Math.rint(price);
            return inventoryId;
        }
}
```

The following example shows the server Product's `readExternal()` method:

```
public void readExternal(ObjectInput in) throws IOException,
    ClassNotFoundException {
    // Read in the server properties from the client representation.
    name = (String)in.readObject();
    properties = (Map)in.readObject();
    price = in.readFloat();
    setInventoryId(lookupInventoryId(name, price));
    }
```

The client Product's `writeExternal()` method does not send the `id` property to the server during serialization because it is not useful to the server version of the Product object. Similarly, the server Product's `writeExternal()` method does not send the `inventoryId` property to the client because it is a server-specific property.

Notice that the names of a Product's properties are not sent during serialization in either direction. Because the state of the class is fixed and manageable, the properties are sent in a well-defined order without their names, and the `readExternal()` method reads them in the appropriate order.

Serializing between ActionScript and web services

Default encoding of ActionScript data

The following table shows the default encoding mappings from ActionScript 3 types to XML schema complex types.

XML schema definition	Supported ActionScript 3 types	Notes
Top-level elements		
xsd:element nillable == true	Object	If input value is `null`, encoded output is set with the `xsi:nil` attribute.
xsd:element fixed != null	Object	Input value is ignored and fixed value is used instead.
xsd:element default != null	Object	If input value is `null`, this default value is used instead.

Local elements		
xsd:element maxOccurs == 0	Object	Input value is ignored and omitted from encoded output.
xsd:element maxOccurs == 1	Object	Input value is processed as a single entity. If the associated type is a SOAP-encoded array, then arrays and `mx.collection.IList` implementations pass through intact and are handled as a special case by the SOAP encoder for that type.
xsd:element maxOccurs > 1	Object	Input value should be iterable (such as an array or `mx.collections.IList` implementation), although noniterable values are wrapped before processing. Individual items are encoded as separate entities according to the definition.
xsd:element minOccurs == 0	Object	If input value is undefined or `null`, encoded output is omitted.

The following table shows the default encoding mappings from ActionScript 3 types to XML schema built-in types.

XML schema type	Supported ActionScript 3 types	Notes
`xsd:anyType` `xsd:anySimpleType`	Object	Boolean -> `xsd:boolean` ByteArray -> `xsd:base64Binary` Date -> `xsd:dateTime` int -> `xsd:int` Number -> `xsd:double` String -> `xsd:string` uint -> `xsd:unsignedInt`
`xsd:base64Binary`	`flash.utils.ByteArray`	`mx.utils.Base64Encoder` is used (without line wrapping).
`xsd:boolean`	Boolean Number Object	Always encoded as `true` or `false`. `Number == 1` then `true`, otherwise `false`. `Object.toString()` == "true" or "1" then `true`, otherwise `false`.
`xsd:byte` `xsd:unsignedByte`	Number String	String first converted to Number.
`xsd:date`	Date Number String	Date UTC accessor methods are used. Number used to set Date.time. String assumed to be preformatted and encoded as is.
`xsd:dateTime`	Date Number String	Date UTC accessor methods are used. Number used to set Date.time. String assumed to be preformatted and encoded as is.
`xsd:decimal`	Number String	`Number.toString()` is used. Note that Infinity, -Infinity, and NaN are invalid for this type. String first converted to Number.

XML schema type	Supported ActionScript 3 types	Notes
`xsd:double`	Number String	Limited to range of Number. String first converted to Number.
`xsd:duration`	Object	`Object.toString()` is called.
`xsd:float`	Number String	Limited to range of Number. String first converted to Number.
`xsd:gDay`	Date Number String	`Date.getUTCDate()` is used. Number used directly for day. String parsed as Number for day.
`xsd:gMonth`	Date Number String	`Date.getUTCMonth()` is used. Number used directly for month. String parsed as Number for month.
`xsd:gMonthDay`	Date String	`Date.getUTCMonth()` and `Date.getUTCDate()` are used. String parsed for month and day portions.
`xsd:gYear`	Date Number String	`Date.getUTCFullYear()` is used. Number used directly for year. String parsed as Number for year.
`xsd:gYearMonth`	Date String	`Date.getUTCFullYear()` and `Date.getUTCMonth()` are used. String parsed for year and month portions.
`xsd:hexBinary`	flash.utils.ByteArray	mx.utils.HexEncoder is used.

`xsd:integer`
`and derivatives:`
`xsd:negativeInteger`
`xsd:nonNegativeInteger`
`xsd:positiveInteger`
`xsd:nonPositiveInteger`
	Number
	String
		Limited to range of Number.
		String first converted to Number.

`xsd:int` `xsd:unsignedInt`	Number String	String first converted to Number.
`xsd:long` `xsd:unsignedLong`	Number String	String first converted to Number.
`xsd:short` `xsd:unsignedShort`	Number String	String first converted to Number.

XML schema type	Supported ActionScript 3 types	Notes

`xsd:string`
`and derivatives:`
`xsd:ID`
`xsd:IDREF`
`xsd:IDREFS`
`xsd:ENTITY`
`xsd:ENTITIES xsd:language`
`xsd:Name`
`xsd:NCName`
`xsd:NMTOKEN`
`xsd:NMTOKENS`
`xsd:normalizedString`
`xsd:token`
	Object	`Object.toString()` is invoked.

`xsd:time`	Date Number String	Date UTC accessor methods are used. Number used to set Date.time. String assumed to be preformatted and encoded as is.
`xsi:nil`	null	If the corresponding XML schema element definition has minOccurs > 0, a `null` value is encoded by using xsi:nil; otherwise the element is omitted entirely.

The following table shows the default mapping from ActionScript 3 types to SOAP-encoded types.

SOAPENC type	Supported ActionScript 3 types	Notes
soapenc:Array	Array mx.collections.IList	SOAP-encoded arrays are special cases and are supported only with RPC-encoded web services.
soapenc:base64	flash.utils.ByteArray	Encoded in the same manner as `xsd:base64Binary`.
soapenc:*	Object	Any other SOAP-encoded type is processed as if it were in the XSD namespace based on the `localName` of the type's QName.

Default decoding of XML schema and SOAP to ActionScript 3

The following table shows the default decoding mappings from XML schema built-in types to ActionScript 3 types.

XML schema type	Decoded ActionScript 3 types	Notes
`xsd:anyType` `xsd:anySimpleType`	String Boolean Number	If content is empty -> `xsd:string`. If content cast to Number and value is NaN; or if content starts with "0" or "-0", or if content ends with "E": then, if content is "true" or "false" -> `xsd:boolean` otherwise -> `xsd:string`. Otherwise content is a valid Number and thus -> `xsd:double`.
`xsd:base64Binary`	`flash.utils.ByteArray`	`mx.utils.Base64Decoder` is used.

XML schema type	Decoded ActionScript 3 types	Notes
`xsd:boolean`	Boolean	If content is "true" or "1" then `true`, otherwise `false`.
`xsd:date`	Date	If no time zone information is present, local time is assumed.
`xsd:dateTime`	Date	If no time zone information is present, local time is assumed.
`xsd:decimal`	Number	Content is created via `Number(content)` and is thus limited to the range of Number. Number.NaN, Number.POSITIVE_INFINITY and Number.NEGATIVE_INFINITY are not allowed.
`xsd:double`	Number	Content is created via `Number(content)` and is thus limited to the range of Number.
`xsd:duration`	String	Content is returned with whitespace collapsed.
`xsd:float`	Number	Content is converted through `Number(content)` and is thus limited to the range of Number.
`xsd:gDay`	uint	Content is converted through `uint(content)`.
`xsd:gMonth`	uint	Content is converted through `uint(content)`.
`xsd:gMonthDay`	String	Content is returned with whitespace collapsed.
`xsd:gYear`	uint	Content is converted through `uint(content)`.
`xsd:gYearMonth`	String	Content is returned with whitespace collapsed.
`xsd:hexBinary`	`flash.utils.ByteArray`	`mx.utils.HexDecoder` is used.

`xsd:integer`
 	
`and derivatives:`
`xsd:byte`
`xsd:int`
`xsd:long`
`xsd:negativeInteger`
`xsd:nonNegativeInteger`
`xsd:nonPositiveInteger`
`xsd:positiveInteger`
`xsd:short`
`xsd:unsignedByte`
`xsd:unsignedInt`
`xsd:unsignedLong`
`xsd:unsignedShort`
	Number	Content is decoded via `parseInt()`.
		
		Number.NaN, Number.POSITIVE_INFINITY and Number.NEGATIVE_INFINITY are not allowed.

XML schema type	Decoded ActionScript 3 types	Notes

`xsd:string`
and derivatives:
`xsd:ID`
`xsd:IDREF`
`xsd:IDREFS`
`xsd:ENTITY`
`xsd:ENTITIES xsd:language`
`xsd:Name`
`xsd:NCName`
`xsd:NMTOKEN`
`xsd:NMTOKENS`
`xsd:normalizedString`
`xsd:token`
	String	The raw content is simply returned as a string.

`xsd:time`	Date	If no time zone information is present, local time is assumed.
`xsi:nil`	null	

The following table shows the default decoding mappings from SOAP-encoded types to ActionScript 3 types.

SOAPENC type	Decoded ActionScript type	Notes
`soapenc:Array`	Array `mx.collections.ArrayCollection`	SOAP-encoded arrays are special cases. If `makeObjectsBindable` is `true`, the result is wrapped in an ArrayCollection; otherwise a simple array is returned.
`soapenc:base64`	`flash.utils.ByteArray`	Decoded in the same manner as `xsd:base64Binary`.
`soapenc:*`	Object	Any other SOAP-encoded type is processed as if it were in the XSD namespace based on the `localName` of the type's QName.

The following table shows the default decoding mappings from custom data types to ActionScript 3 data types.

Custom type	Decoded ActionScript 3 type	Notes
Apache Map `http://xml.apache.org/xml-soap:Map`	Object	SOAP representation of `java.util.Map`. `Keys` must be representable as strings.
Apache Rowset `http://xml.apache.org/xml-soap:Rowset`	Array of objects	
ColdFusion QueryBean `http://rpc.xml.coldfusion:QueryBean`	Array of objects `mx.collections.ArrayCollection` of objects	If `makeObjectsBindable` is true, the resulting array is wrapped in an ArrayCollection.

XML Schema element support

The following XML schema structures or structure attributes are only partially implemented in Flex 3:

```
<choice>
<all>
<union
```

The following XML Schema structures or structure attributes are ignored and are not supported in Flex 3:

```
<attribute use="required"/>

<element
    substitutionGroup="..."
    unique="..."
    key="..."
    keyref="..."
    field="..."
    selector="..."/>

<simpleType>
    <restriction>
        <minExclusive>
        <minInclusive>
        <maxExclusiv>
        <maxInclusive>
        <totalDigits>
        <fractionDigits>
        <length>
        <minLength>
        <maxLength>
        <enumeration>
        <whiteSpace>
        <pattern>
    </restriction>
</simpleType>

<complexType
    final="..."
    block="..."
    mixed="..."
    abstract="..."/>

<any
processContents="..."/>
<annotation>
```

Customizing web service type mapping

When consuming data from a web service invocation, Flex usually creates untyped anonymous ActionScript objects that mimic the XML structure in the body of the SOAP message. If you want Flex to create an instance of a specific class, you can use an mx.rpc.xml.SchemaTypeRegistry object and register a QName object with a corresponding ActionScript class.

For example, suppose you have the following class definition in a file named User.as:

```
package
{
  public class User
  {
    public function User() {}

    public var firstName:String;
    public var lastName:String;
  }
}
```

Next, you want to invoke a `getUser` operation on a web service that returns the following XML:

```
<tns:getUserResponse xmlns:tns="http://example.uri">
  <tns:firstName>Ivan</tns:firstName>
  <tns:lastName>Petrov</tns:lastName>
</tns:getUserResponse>
```

To make sure you get an instance of your User class instead of a generic Object when you invoke the getUser operation, you need the following ActionScript code inside a method in your Flex application:

```
SchemaTypeRegistry.getInstance().registerClass(new QName("http://example.uri",
"getUserResponse"), User);
```

`SchemaTypeRegistry.getInstance()` is a static method that returns the default instance of the type registry. In most cases, that is all you need. However, this registers a given QName with the same ActionScript class across all web service operations in your application. If you want to register different classes for different operations, you need the following code in a method in your application:

```
var qn:QName = new QName("http://the.same", "qname");
var typeReg1:SchemaTypeRegistry = new SchemaTypeRegistry();
var typeReg2:SchemaTypeRegistry = new SchemaTypeRegistry();
typeReg1.registerClass(qn, someClass);
myWS.someOperation.decoder.typeRegistry = typeReg1;

typeReg2.registerClass(qn, anotherClass);
myWS.anotherOperation.decoder.typeRegistry = typeReg2;
```

Using custom web service serialization

There are two approaches to take full control over how ActionScript objects are serialized into XML and how XML response messages are deserialized. The recommended approach is to work directly with E4X.

If you pass an instance of XML as the only parameter to a web service operation, it is passed on untouched as the child of the `<SOAP:Body>` node in the serialized request. Use this strategy when you need full control over the SOAP message. Similarly, when deserializing a web service response, you can set the operation's `resultFormat` property to e4x. This returns an XMLList object with the children of the `<SOAP:Body>` node in the response message. From there, you can implement the necessary custom logic to create the appropriate ActionScript objects.

The second and more tedious approach is to provide your own implementations of mx.rpc.soap.ISOAPDecoder and mx.rpc.soap.ISOAPEncoder. For example, if you have written a class called MyDecoder that implements ISOAPDecoder, you can have the following in a method in your Flex application:

```
myWS.someOperation.decoder = new MyDecoder();
```

When invoking `someOperation`, Flex calls the `decodeResponse()` method of the MyDecoder class. From that point on it is up to the custom implementation to handle the full SOAP message and produce the expected ActionScript objects.

Part 3: RPC services

Using HTTP and web services 73
Using the Remoting Service 110

Chapter 7: Using HTTP and web services

Remote Procedure Call (RPC) components let a client application make calls to operations and services across a network. The three RPC components are the RemoteObject, HTTPService, and WebService components. Your Flex client code uses these components to access remote object services, web services, and HTTP services.

Topics

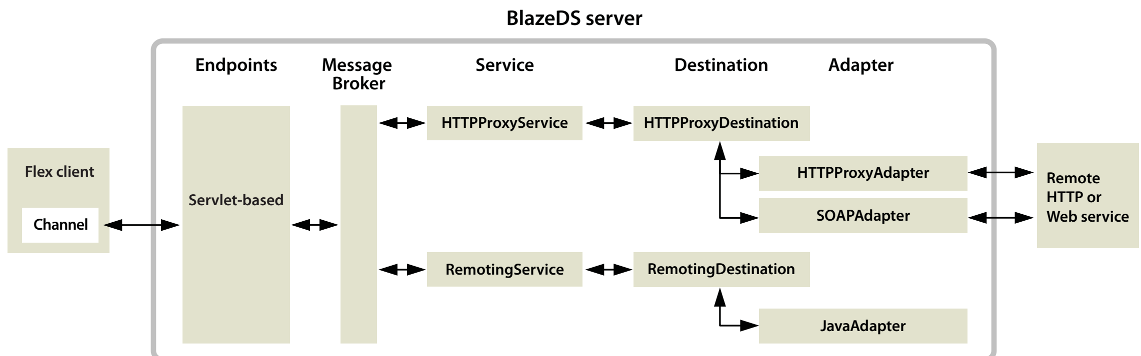
RPC components. 73
RPC components versus other technologies 77
Using destinations. 78
Defining and invoking a service component 83
Handling service events 86
Passing parameters to a service. 90
Handling service results. 94
Using capabilities specific to WebService components 103
Handling asynchronous calls to services 107

RPC components

The RPC components are designed for client applications in which a call and response model is a good choice for accessing external data. These components let the client make asynchronous requests to remote services that process the requests, and then return data to your Flex application.

The RPC components call a remote service, and then store response data from the service in an ActionScript or XML object from which you obtain the data. You can use the RPC components in the client application to work with three types of RPC services: remote object services with the RemoteObject component, web services with the WebService component, and HTTP services with the HTTPService component.

When you use BlazeDS, the client typically contacts a destination, which is an RPC service that has corresponding server-side configurations. The following diagram shows how the RPC components in a Flex client application interact with BlazeDS services:



RPC services, destinations, and adapters

The services, destinations, and adapters that you use depend on the RPC component, as the following table shows:

Component	Service	Destination	Adapter
HTTPService	HTTPProxyService	HTTPProxyDestination	HTTPProxyAdapter
WebService	HTTPProxyService	HTTPProxyDestination	SOAPAdapter
RemoteObject	RemotingService	RemotingDestination	JavaAdapter

RPC channels

With RPC services, you often use an AMFChannel. The AMFChannel uses binary AMF encoding over HTTP. If binary data is not allowed, then you can use an HTTPChannel, which is AMFX (AMF in XML) over HTTP. For more information on channels, see "BlazeDS architecture" on page 27.

Types of RPC components

Use RPC components to add enterprise functionality, such as proxying of service traffic from different domains, client authentication, whitelists of permitted RPC service URLs, security, server-side logging, localization support, and centralized management of RPC services.

Note: *You can use the HTTPService and WebService components to call HTTP services or web services directly, without going through the server-side proxy service. For more information, see Using HTTPService and WebService without a destination.*

By default, Adobe Flash Player blocks access to any host that is not exactly equal to the one used to load an application. Therefore, if you do not use BlazeDS to proxy requests, an HTTP or web service must either be on the server hosting your application, or the remote server that hosts the HTTP or web service must define a crossdomain.xml file. A *crossdomain.xml file* is an XML file that provides a way for a server to indicate that its data and documents are available to SWF files served from certain domains, or from all domains. The crossdomain.xml file must be in the web root of the server that the Flex application is contacting.

Use RemoteObject components to access remote Java objects on the BlazeDS server without configuring them as SOAP-compliant web services. You cannot use RemoteObject components without BlazeDS or ColdFusion. For detailed information on the RemoteObject component, see "Using the Remoting Service" on page 110.

HTTPService component

HTTPService components let you send HTTP GET, POST, HEAD, OPTIONS, PUT, TRACE or DELETE requests, and include data from HTTP responses in a Flex application. Flex does not support mulitpart form POST requests. However, when you do not go through the HTTPProxyService, you can use only HTTP GET or POST methods.

An HTTP service can be any HTTP URI that accepts HTTP requests and sends responses. Another common name for this type of service is a REST-style web service. REST stands for Representational State Transfer and is an architectural style for distributed hypermedia systems. For more information about REST, see www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

HTTPService components are a good option when you cannot expose the same functionality as a SOAP web service or a remote object service. For example, you can use HTTPService components to interact with JavaServer Pages (JSPs), servlets, and ASP pages that are not available as web services or Remoting Service destinations.

Use an HTTPService component for CGI-like interaction in which you use HTTP GET, POST, HEAD, OPTIONS, PUT, TRACE, or DELETE to send a request to a specified URI. When you call the HTTPService object's `send()` method, it makes an HTTP request to the specified URI, and an HTTP response is returned. Optionally, you can pass arguments to the specified URI.

WebService component

WebService components let you access *web services*, which are software modules with methods. Web service methods are commonly referred to as *operations*. Web service interfaces are defined by using XML. Web services provide a standards-compliant way for software modules that are running on a variety of platforms to interact with each other. For more information about web services, see the web services section of the World Wide Web Consortium website at www.w3.org/2002/ws/.

Flex applications can interact with web services that define their interfaces in a Web Services Description Language (WSDL) document, which is available as a URL. WSDL is a standard format for describing the messages that a web service understands, the format of its responses to those messages, the protocols that the web service supports, and where to send messages.

Flex supports WSDL 1.1, which is described at www.w3.org/TR/wsdl. Flex supports both RPC-encoded and document-literal web services.

Flex applications support web service requests and results that are formatted as SOAP messages and are transported over HTTP. SOAP provides the definition of the XML-based format that you can use for exchanging structured and typed information between a web service client, such as a Flex application, and a web service.

You can use a WebService component to connect to a SOAP-compliant web service when web services are an established standard in your environment. WebService components are also useful for objects that are within an enterprise environment, but not necessarily available on the sourcepath of the Flex web application.

RemoteObject component

RemoteObject components let you access the methods of server-side Java objects, without manually configuring the objects as web services. You can use RemoteObject components in MXML or ActionScript.

You can use RemoteObject components with a stand-alone BlazeDS web application or Macromedia® ColdFusion® MX from Adobe. When using a BlazeDS web application, you configure the objects that you want to access as Remoting Service destinations in a BlazeDS configuration file or by using BlazeDS run-time configuration. For information on using RemoteObject components with ColdFusion, see the ColdFusion documentation.

Use a RemoteObject component instead of a WebService component when objects are not already published as web services, web services are not used in your environment, or you would rather use Java objects than web services. You can use a RemoteObject component to connect to a local Java object that is in the BlazeDS or ColdFusion web application sourcepath.

When you use a RemoteObject tag, data is passed between your application and the server-side object in the binary Action Message Format (AMF) format.

For more information about using RemoteObject components, see “Using the Remoting Service” on page 110.

Using an RPC component

The following example shows MXML code for a WebService component. This example connects to a BlazeDS destination, calls the `getProducts()` operations of the web service in response to the `click` event of a Button control, and displays the result data in a DataGrid control.

```
<?xml version="1.0"?>
<!-- ds\rpc\RPCIntroExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;
```

```
            import mx.controls.Alert;

            public function handleResult(event:ResultEvent):void {
                // Handle result by populating the DataGrid control.
                // The operation returns an Array containing product ID, name, and price.
                myDG.dataProvider=event.result;
            }

            public function handleFault(event:FaultEvent):void {
                // Handle fault.
                Alert.show(event.fault.faultString, "Fault");
            }
        ]]>
    </mx:Script>

    <!-- Define a WebService component and connect to a service destination. -->
    <mx:WebService
        id="adbe_news"
        useProxy="true"
        destination="ws-catalog"
        result="handleResult(event);"
        fault="handleFault(event);"/>

    <!-- Call the getProducts() operation of the web service.
      The operation takes no parameters. -->
    <mx:Button click="adbe_news.getProducts();"/>

    <!-- Define a DataGrid control to diplay the results of the web service. -->
    <mx:DataGrid id="myDG" width="100%" height="100%">
        <mx:columns>
            <mx:DataGridColumn dataField="productId" headerText="Product Id"/>
            <mx:DataGridColumn dataField="name" headerText="Name"/>
            <mx:DataGridColumn dataField="price" headerText="Price"/>
        </mx:columns>
    </mx:DataGrid>

</mx:Application>
```

This example shows the basic process for using any RPC control, including the following:

- Defines the component in MXML. You can also define a component in ActionScript. For more information, see “Defining and invoking a service component” on page 83.
- Specifies the BlazeDS destination that the RPC component connects to. For more information, see “Using destinations” on page 78.

- Invokes the `getProducts()` operation in response to a Button `click` event. In this example, the `getProducts()` operation takes no parameters. For more information on parameter passing, see “Passing parameters to a service” on page 90.
- Defines event handlers for the `result` event and for the `fault` event. Calls to an RPC service are asynchronous. After you invoke an asynchronous call, your application does not block execution and wait for immediate response, but continues to execute. Components use events to signal that the service has completed. In this example, the handler for the `result` event populates the DataGrid control with the results of the operation. For more information, see “Handling service events” on page 86.

RPC components versus other technologies

The way that Flex works with data sources and data is different from other web application environments, such as JSP, ASP, and ColdFusion. Data access in Flex applications also differs significantly from data access in applications that are created in Flash Professional.

Client-side processing and server-side processing

Unlike a set of HTML templates created using JSPs and servlets, ASP, or CFML, the files in a Flex application are compiled into a binary SWF file that is sent to the client. When a Flex application makes a request to an external service, the SWF file is not recompiled and no page refresh is required.

The following example shows MXML code for calling a web service. When a user clicks the Button control, client-side code calls the web service, and result data is returned into the binary SWF file without a page refresh. The result data is then available to use as dynamic content within the application.

```
<?xml version="1.0"?>
<!-- ds\rpc\RPCIntroExample2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <!-- Declare a WebService component (the specified WSDL URL is not functional). -->
    <mx:WebService id="WeatherService"
        destination="wsDest"/>

    <mx:Button label="Get Weather"
        click="WeatherService.GetWeather(input.text);"/>

    <mx:TextInput id="input"/>
</mx:Application>
```

The following example shows JSP code for calling a web service using a JSP custom tag. When a user requests this JSP, the web service request is made on the server instead of on the client, and the result is used to generate content in the HTML page. The application server regenerates the entire HTML page before sending it back to the browser.

```
<%@ taglib prefix="web" uri="webservicetag" %>

<% String str1="BRL";
String str2="USD";%>

<!-- Call the web service. -->
<web:invoke
    url="http://www.itfinity.net:8008/soap/exrates/default.asp"
    namespace="http://www.itfinity.net/soap/exrates/exrates.xsd"
    operation="GetRate"
    resulttype="double"
    result="myresult">
    <web:param name="fromCurr" value="<%=str1%>"/>
```

```
    <web:param name="ToCurr" value="<%=str2%>"/>
</web:invoke>

<!-- Display the web service result. -->
<%= pageContext.getAttribute("myresult") %>
```

Data source access

Another difference between Flex and other web application technologies is that you never communicate directly with a data source in Flex. You use a Flex service component to connect to a server-side service that interacts with the data source.

The following example shows one way to access a data source directly in a ColdFusion page:

```
<CFQUERY DATASOURCE="Dsn"
    NAME="myQuery">
    SELECT * FROM table
</CFQUERY>
```

To get similar functionality in Flex, use an HTTPService, a WebService, or a RemoteObject component to call a server-side object that returns results from a data source.

Using destinations

You typically connect an RPC component to a destination defined in the services-config.xml file or a file that it includes by reference, such as the proxy-config.xml file. A destination definition is a named service configuration that provides server-proxied access to an RPC service. A destination is the actual service or object that you want to call.

Destination definitions provide centralized administration of RPC services. They also enable you to use basic or custom authentication to secure access to destinations. You can choose from several different transport channels, including secure channels, for sending data to and from destinations. Additionally, you can use the server-side logging capability to log RPC service traffic.

You have the option of omitting the destination and connecting to HTTP and web services directly by specifying the URL of the service. For more information, see “Using HTTPService and WebService without a destination” on page 82. However, you must define a destination when using the RemoteObject component.

You configure HTTP services and web services as HTTPProxyService destinations. The following example shows a HTTPProxyService destination definition for an HTTP service in the proxy-config.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<service id="proxy-service" class="flex.messaging.services.HTTPProxyService">
    ...

    <destination id="myHTTPService">
        <properties>
            <!-- The endpoint available to the http proxy service -->
            <url>http://www.mycompany.com/services/myservlet</url>
            <!-- Wildcard endpoints available to the http proxy services -->
            <dynamic-url>http://www.mycompany.com/services/*</dynamic-url>
        </properties>
    </destination>
</service>
```

Using an RPC component with a server-side destination

The `destination` property of an RPC component references a destination configured in the proxy-config.xml file. A destination specifies the RPC service class or URL, the transport channel to use, the adapter with which to access the RPC service, and security settings.

To declare a connection to a destination in MXML, set the `id` and `destination` properties in the RPC component. The `id` property is required for calling the services and handling service results. The following example shows HTTPService and WebService component declarations in MXML:

```
<?xml version="1.0"?>
<!-- ds\rpc\RPCMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HTTPService
        id="yahoo_web_search"
        useProxy="true"
        destination="catalog"/>

    <mx:WebService
        id="adbe_news"
        useProxy="true"
        destination="ws-catalog"/>
</mx:Application>
```

When you use a destination with an HTTPService or WebService component, you set its `useProxy` property to `true` to configure it to use the HTTPProxyService. The HTTPProxyService and its adapters provide functionality that lets applications access HTTP services and web services on different domains. Additionally, the HTTPProxyService lets you limit access to specific URLs and URL patterns, and provide security.

***Note:** Setting the `destination` property of the HTTPService or WebService component automatically sets the `useProxy` property to `true`.*

When you do not have BlazeDS or do not require the functionality provided by the HTTPProxyService, you can bypass it. You bypass the proxy by setting the `useProxy` property of an HTTPService or WebService component to `false`, which is the default value. Also set the `HTTPService.url` property to the URL of the HTTP service, or set the `WebService.wsdl` property to the URL of the WSDL document. The RemoteObject component does not define the `useProxy` property; you always use a BlazeDS destination with the RemoteObject component.

Configuring a destination

You configure a destination in a service definition in the proxy-config.xml file. The following example shows a basic server-side configuration for a WebService component in the proxy-config.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<service id="proxy-service" class="flex.messaging.services.HTTPProxyService">

    <properties>
        <connection-manager>
            <max-total-connections>100</max-total-connections>
            <default-max-connections-per-host>2</default-max-connections-per-host>
        </connection-manager>
        <allow-lax-ssl>true</allow-lax-ssl>
    </properties>

    <!-- Channels are typically defined in the services-config.xml file. -->
    <default-channels>
        <channel ref="my-http"/>
        <channel ref="my-amf"/>
    </default-channels>
```

```
    <!-- Define the adapters used by the different destinations. -->
    <adapters>
        <adapter-definition id="http-proxy"
            class="flex.messaging.services.http.HTTPProxyAdapter"
            default="true"/>
        <adapter-definition id="soap-proxy"
            class="flex.messaging.services.http.SOAPProxyAdapter"/>
    </adapters>

    <!-- HTTPService destination uses the default adapter. -->
    <destination id="catalog">
        <properties>
            <url>/{context.root}/testdrive-httpservice/catalog.jsp</url>
        </properties>
    </destination>

    <!-- WebService destination uses the SOAPAdapter. -->
    <destination id="ws-catalog">
        <properties>
            <wsdl>http://server.org/services/ProductWS?wsdl</wsdl>
            <soap>*</soap>
        </properties>
        <adapter ref="soap-proxy"/>
    </destination>
</service>
```

HTTPService and WebService components connect to HTTPProxyService destinations. Therefore, the `class` attribute of the `<service>` tag specifies the HTTPProxyService class.

The adapter is server-side code that interacts with the remote service. Specify the HTTPProxyAdapter with a destination defined for the HTTPService component, and the SOAPAdapter for the WebService component.

This destination uses an HTTP or an Action Message Format (AMF) message channel for transporting data. Optionally, it could use one of the other supported message channels. Message channels are defined in the services-config.xml file, in the `channels` section under the `services-config` element. For more information, see “Channels and endpoints” on page 38.

You use the `url` and `dynamic-url` elements to configure HTTP service URLs in a destination. These elements define which URLs are permitted for a destination. The following table describes those elements:

Element	Description
`url`	(Optional) Default URL.
`dynamic-url`	(Optional) HTTP service URL patterns. You can use more than one dynamic-url entry to specify multiple URL patterns. Flex matches these values against `url` property values that you specify in client-side service tags or ActionScript code.

You use the `wsdl` and `soap` elements to configure web service URLs in a destination. These elements define which URLs are permitted for a destination. The following table describes those elements:

Element	Description
`wsdl`	(Optional) Default WSDL URL.
`soap`	SOAP endpoint URL patterns that would typically be defined for each operation in the WSDL document. You can use more than one soap entry to specify multiple SOAP endpoint patterns. Flex matches these values against `endpointURI` property values that you specify in client-side service tags or ActionScript code.

Using HTTPService and WebService with the default destination

In the following situation, a component uses a default destination:

- You set the `useProxy` property to `true`
- You do not set the `destination` property
- You set the `HTTPService.url` property or the `WebService.wsdl` property

The default destinations are named `DefaultHTTP` and `DefaultHTTPS`. The `DefaultHTTPS` destination is used when your `url` or `wsdl` property specifies an HTTPS URL.

By setting the `url` or `wsdl` property, you let the component specify the URL of the remote service, rather than specifying it in the destination. However, since the request uses the default destination, you can take advantage of BlazeDS. Therefore, you can use basic or custom authentication to secure access to the destination. You can choose from several different transport channels, including secure channels, for sending data to and from destinations. Additionally, you can use the server-side logging capability to log remote service traffic.

Configure the default destinations in the proxy-config.xml file. Use one or more `dynamic-url` parameters to specify URL patterns for the HTTP service, or one or more `soap` parameters to specify URL patterns for the WSDL of the web service. The value of the `url` or `wsdl` property of the component must match the specified pattern.

The following example shows a default destination definition that specifies a `dynamic-url` value:

```
<service id="proxy-service" class="flex.messaging.services.HTTPProxyService">
    ...

    <destination id="DefaultHTTP">
        <channels>
            <channel ref="my-amf"/>
        </channels>

        <properties>
            <dynamic-url>http://mysite.com/myservices/*</dynamic-url>
        </properties>
        ...
    </destination>
</service>
```

Therefore, set the `HTTPService.url` property to a URL that begins with the pattern http://mysite.com/myservices/. Setting the `HTTPService.url` property to any other value causes a `fault` event.

The following table describes elements that you use to configure the default destinations:

Parameter	Description
`dynamic-url`	Specifies the URL pattern for the HTTP service. You can use one or more `dynamic-url` parameters to specify multiple patterns. Any request from the client must match one of the patterns.
`soap`	Specifies the URL pattern for a WSDL or the location of a web service. You can use one or more `soap` parameters to specify multiple patterns. The URL of the WSDL, or the URL of the web service, must match one of the patterns. If you specify a URL for the `wsdl` property, the URL must match a pattern,

Configuring the Proxy Service

Configure the Proxy Service by using the proxy-config.xml file. The `service` parameter contains a `properties` element that you use to configure the Apache connection manager, self-signed certificates for SSL, and external proxies. The following table describes elements that you use to configure the Proxy Service:

Element	Description
`connection-manager`	Contains the `max-total-connections` and `default-max-connections-per-host` elements. The `max-total-connections` element controls the maximum total number of concurrent connections that the proxy supports. If the value is greater than 0, BlazeDS uses a multithreaded connection manager for the underlying Apache HttpClient proxy. The `default-max-connections-per-host` element sets the default number of connections allowed for each host in an environment that uses hardware clustering.
`content-chunked`	Specifies whether to use chunked content. The default value is `false`. Flash Player does not support chunked content.
`allow-lax-ssl`	Set to `true` when using SSL to allow self-signed certificates. Do not set to `true` in a production environment.
`external-proxy`	Specifies the location of an external proxy, as well as a user name and a password, when the Proxy Service must contact an external proxy before getting access to the Internet. The properties of `external-proxy` depend on the external proxy and the underlying Apache HttpClient proxy. For more information on the Apache HttpClient, see the Apache website.

The following example shows a Proxy Service configuration:

```
<service id="proxy-service" class="flex.messaging.services.HTTPProxyService">

    <!-- Define channels and destinations. -->

    <properties>
        <connection-manager>
            <max-total-connections>100</max-total-connections>
            <default-max-connections-per-host>2
            </default-max-connections-per-host>
        </connection-manager>

        <!-- Allow self-signed certificates. Do not use in production -->
        <allow-lax-ssl>true</allow-lax-ssl>

        <!-- Connection settings for an external proxy. -->
        <external-proxy>
            <server>10.10.10.10</server>
            <port>3128</port>
            <nt-domain>mycompany</nt-domain>
            <username>flex</username>
            <password>flex</password>
        </external-proxy>
    </properties>
</service>
```

Using HTTPService and WebService without a destination

You can connect to HTTP services and web services without configuring a destination. To do so, you set the `HTTPService.url` property or the `WebService.wsdl` property instead of setting the `destination` property. Additionally, set the `useProxy` property of the component to `false` to bypass the HTTPProxyService. When the `useProxy` property is set to `false`, the component communicates directly with the service based on the `url` or `wsdl` property value.

When you set the `useProxy` property to `true` for the HTTPService component, you can use the HTTP HEAD, OPTIONS, TRACE, and DELETE methods. However, when you do not go through the HTTPProxyService, you can use only HTTP GET or POST methods. By default, the HTTPService method uses the GET method.

Note: *If you bypass the proxy, and the status code of the HTTP response is not a success code from 200 through 299, Flash Player cannot access any data in the body of the response. For example, a server sends a response with an error code of 500 with the error details in the body of the response. Without a proxy, the body of the response is inaccessible by the Flex application.*

Connecting to a service in this manner requires that at least one of the following is true:

- The service is in the same domain as your Flex application.
- A crossdomain.xml (cross-domain policy) file is installed on the web server hosting the RPC service that allows access from the domain of the application. For more information, see the Adobe Flex 3 documentation.

The following examples show MXML tags for declaring HTTPService and WebService components that directly reference RPC services. The `id` property is required for calling the services and handling service results. In these examples, the `useProxy` property is not set in the tags. Therefore, the components use the default `useProxy` value of `false` and contact the services directly:

```
<?xml version="1.0"?>
<!-- ds\rpc\RPCNoServer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HTTPService
        id="yahoo_web_search"
        url="http://api.search.yahoo.com/WebSearchService/V1/webSearch"/>

    <mx:WebService
        id="macr_news"
        wsdl="http://ws.invesbot.com/companysearch.asmx?wsdl"/>
</mx:Application>
```

Defining and invoking a service component

You define an RPC component in MXML or ActionScript. After defining the component, use the component to call the remote service and process any results.

Defining and invoking an HTTPService component

Use the HTTPService components in your client-side application to make an HTTP request to a URL. The following examples show HTTPService component declarations in MXML and in ActionScript. Regardless of how you define the component, you send a request to the destination by calling the `HTTPService.send()` method. To handle the results, you use the `result` and `fault` event handlers:

```
<?xml version="1.0"?>
<!-- ds\rpc\HttpService.mxml. -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="useHttpService();">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.rpc.http.mxml.HTTPService;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;
```

```
            private var asService:HTTPService

            // Define the HTTPService component in ActionScript.
            public function useHttpService():void {
                asService = new HTTPService();
                asService.method = "POST";
                asService.useProxy = true;
                asService.destination = "catalog";
                asService.addEventListener("result", httpResult);
                asService.addEventListener("fault", httpFault);
            }

            public function httpResult(event:ResultEvent):void {
                //Do something with the result.
            }

            public function httpFault(event:FaultEvent):void {
                var faultstring:String = event.fault.faultString;
                Alert.show(faultstring);
            }
        ]]>
    </mx:Script>

    <!-- Define the HTTPService component in MXML. -->
    <mx:HTTPService
        id="mxmlService"
        method="POST"
        useProxy="true"
        destination="catalog"
        result="httpResult(event);"
        fault="httpFault(event);"/>

    <mx:Button label="MXML" click="mxmlService.send();"/>
    <mx:Button label="AS" click="asService.send();"/>

</mx:Application>
```

Defining and invoking a WebService component

Use the WebService components in your client-side application to make a SOAP request. The following example defines a WebService component in MXML and ActionScript:

```
<?xml version="1.0"?>
<!-- ds\rpc\WebServiceExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="useWebService();">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.rpc.soap.mxml.WebService;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;

            private var asService:WebService;

            // Define the WebService component in ActionScript.
            public function useWebService():void {
                asService = new WebService();
                asService.destination = "ws-catalog";
```

```
                asService.addEventListener("result", wsResult);
                asService.addEventListener("fault", wsFault);
                asService.loadWSDL();
            }

            public function wsResult(event:ResultEvent):void {
                //Do something with the result.
            }

            public function wsFault(event:FaultEvent):void {
                var faultstring:String = event.fault.faultString;
                Alert.show(faultstring);
            }
        ]]>
    </mx:Script>

    <!-- Define the WebService component in MXML. -->
    <mx:WebService
        id="mxmlService"
        destination="ws-catalog"
        result="wsResult(event);"
        fault="wsFault(event);"/>

    <mx:Button label="MXML" click="mxmlService.getProducts();"/>
    <mx:Button label="AS" click="asService.getProducts();"/>

</mx:Application>
```

The destination specifies the WSDL associated with this web service. A web service can expose multiple methods, corresponding to multiple operations. In this example, you directly call the `getProducts()` operation of the web service in response to a `click` event of a Button control.

Notice that the ActionScript version of the WebService component calls the `WebService.loadWSDL()` method to load the WSDL. This method is called automatically when you define the component in MXML.

Using an Operation object with the WebService component

Because a single WebService component can invoke multiple operations on the web service, the component requires a way to represent information specific to each operation. Therefore, for every operation, the component creates an mx.rpc.soap.mxml.Operation object.

The name of the Operation object corresponds to the name of the operation. From the example shown in "Defining and invoking a WebService component" on page 84, access the Operation object that corresponds to the `getProducts()` operation by accessing the Operation object named `getProducts`, as the following code shows:

```
// The Operation object has the same name as the operation, without the trailing parentheses.
var myOP:Operation = mxmlService.getProducts;
```

The Operation object contains properties that you use to set characteristics of the operation, such as the arguments passed to the operation, and to hold any data returned by the operation. You access the returned data by using the `Operation.lastResult` property.

Invoke the operation by referencing it relative to the WebService component, as the following example shows:

```
mxmlService.getProducts();
```

Alternatively, invoke an operation by calling the `Operation.send()` method, as the following example shows:

```
mxmlService.getProducts.send();
```

Defining multiple operations for the WebService component

When a web service defines multiple operations, you define multiple operations for the WebService component and specify the attributes for each operation, as the following example shows:

```
<mx:WebService
    id="mxmlService"
    destination="ws-catalog"
    result="wsResult(event);">
        <mx:operation name="getProducts" fault="getPFault(event);"/>
        <mx:operation name="updateProdcut" fault="updatePFault(event);"/>
        <mx:operation name="deleteProduct" fault="deletePFault(event);"/>
</mx:WebService>
```

The `name` property of an `<mx:operation>` tag must match one of the web service operation names. The WebService component creates a separate Operation object for each operation.

Each operation can rely on the event handlers and characteristics defined by the WebService component. However, the advantage of defining the operations separately is that each operation can specify its own event handlers, its own input parameters, and other characteristics. In this example, the WebService component defines the result handler for all three operations, and each operation defines its own fault handler. For more information, see “Handling service events” on page 86 and “Passing parameters to a service” on page 90.

Handling service events

Calls to a remote service are asynchronous. After you invoke an asynchronous call, your application does not wait for the result, but continues to execute. Therefore, you typically use events to signal that the service call has completed.

When a service call completes, the WebService and HTTPService components dispatch one of the following events:

- A *result event* indicates that the result is available. A `result` event generates an mx.rpc.events.ResultEvent object. You can use the `result` property of the ResultEvent object to access the data returned by the service call.
- A *fault event* indicates that an error occurred. A `fault` event generates an mx.rpc.events.FaultEvent object. You can use the `fault` property of the FaultEvent object to access information about the failure.

You can also handle the `invoke` event, which is broadcast when an RPC component makes the service call. This event is useful if operations are queued and invoked at a later time.

You can handle these events at three different levels in your application:

- Handle them at the component level. Specify event handlers for the `fault` and `result` events for the component. By default, these event handlers are invoked for all service calls.
- For the WebService component only, handle them at the operation level. These event handlers override any event handler specified at the component level. The HTTPService component does not support multiple operations, so you use this technique only with the WebService component.
- Handle them at the call level. The `send()` method returns an AsyncToken object. You can assign event handlers to the AsyncToken object to define event handlers for a specific service call.

Handling events at the component level

Handle events at the component level by using the `fault` and `result` properties of the component to specify the event handlers, as the following example shows:

```
<?xml version="1.0"?>
<!-- ds\rpc\RPCIntroExample.mxml -->
```

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;
            import mx.controls.Alert;

            public function handleResult(event:ResultEvent):void {
                // Handle result by populating the DataGrid control.
                // The operation returns an Array containing product ID, name, and price.
                myDG.dataProvider=event.result;
            }

            public function handleFault(event:FaultEvent):void {
                // Handle fault.
                Alert.show(event.fault.faultString, "Fault");
            }
        ]]>
    </mx:Script>

    <!-- Define a WebService component and connect to a service destination. -->
    <mx:WebService
        id="adbe_news"
        useProxy="true"
        destination="ws-catalog"
        result="handleResult(event);"
        fault="handleFault(event);"/>

    <!-- Call the getProducts() operation of the web service.
      The operation takes no parameters. -->
    <mx:Button click="adbe_news.getProducts();"/>

    <!-- Define a DataGrid control to diplay the results of the web service. -->
    <mx:DataGrid id="myDG" width="100%" height="100%">
        <mx:columns>
            <mx:DataGridColumn dataField="productId" headerText="Product Id"/>
            <mx:DataGridColumn dataField="name" headerText="Name"/>
            <mx:DataGridColumn dataField="price" headerText="Price"/>
        </mx:columns>
    </mx:DataGrid>

</mx:Application>
```

In this example, all operations that you invoke through the WebService component use the same event handlers.

Handling events at the operation level for the WebService component

For the WebService component, you can handle events at the operation level. These event handlers override any event handler specified at the component level. When you do not specify event listeners for an operation, the events are passed to the component level.

In the following MXML example, the WebService component defines default event handlers, and the operation specifies its own handlers:

```
<?xml version="1.0"?>
<!-- ds\rpc\RPCResultFaultMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
```

```
        <![CDATA[
            import mx.rpc.soap.SOAPFault;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;
            import mx.controls.Alert;

            public function getProductsResult(event:ResultEvent):void {
                // Handle result.
            }

            public function getProductsFault(event:FaultEvent):void {
                // Handle operation fault.
                Alert.show(event.fault.faultString, "Error");
            }

            public function defaultResult(event:ResultEvent):void {
                // Handle result.
            }

            public function defaultFault(event:FaultEvent):void {
                // Handle service fault.
                if (event.fault is SOAPFault) {
                     var fault:SOAPFault=event.fault as SOAPFault;
                     var faultElement:XML=fault.element;
                     // You could use E4X to traverse the raw fault element
                     // returned in the SOAP envelope.
                }
                Alert.show(event.fault.faultString, "Error");
            }
        ]]>
    </mx:Script>

    <mx:WebService id="WeatherService"
        destination="ws-catalog"
        result="defaultResult(event);"
        fault="defaultFault(event);">
        <mx:operation name="getProducts"
            fault="getProductsFault(event);"
            result="getProductsResult(event);">
        </mx:operation>
    </mx:WebService>

    <mx:Button click="WeatherService.getProducts.send();"/>
</mx:Application>
```

Handling events at the call level

Sometimes, an application requires different event handlers for calls to the same service. For example, a service call uses one event handler at application startup, and a different handler for calls during application execution.

To define event handlers for a specific service call, you can use the AsyncToken object returned by the `send()` method. You then register event handlers on the AsyncToken object, rather than on the component, as the following ActionScript code shows:

```
<?xml version="1.0"?>
<!-- ds\rpc\RPCAsynchEventHandler.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.rpc.soap.SOAPFault;
```

```
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;
            import mx.controls.Alert;
            import mx.rpc.AsyncToken;
            import mx.rpc.AsyncResponder;

            // Define an instance of the AsyncToken class.
            public var serviceToken:AsyncToken;

            // Call the service, and then
            // assign the event handlers to the AsyncToken object.
            public function setCustomHandlers():void {
                // send() returns an instance of AsyncToken.
                serviceToken = WeatherService.getProducts.send();
                var asynchRes:AsyncResponder =
                    new AsyncResponder(getProductsResult, getProductsFault);
                serviceToken.addResponder(asynchRes);
            }

            // Use the token argument to pass additional information to the handler.
           public function getProductsResult(event:ResultEvent, token:Object = null):void {
                // Handle result.
            }

            // Use the token argument to pass additional information to the handler.
            public function getProductsFault(event:FaultEvent, token:Object = null):void {
                // Handle operation fault.
                Alert.show(event.fault.faultString, "Error");
            }

            public function defaultResult(event:ResultEvent):void {
                // Handle result.
            }

            public function defaultFault(event:FaultEvent):void {
                // Handle service fault.
                if (event.fault is SOAPFault) {
                     var fault:SOAPFault=event.fault as SOAPFault;
                     var faultElement:XML=fault.element;
                     // You could use E4X to traverse the raw fault element
                     // returned in the SOAP envelope.
                }
                Alert.show(event.fault.faultString, "Error");
            }
        ]]>
    </mx:Script>

    <mx:WebService id="WeatherService"
        destination="ws-catalog"
        result="defaultResult(event);"
        fault="defaultFault(event);">
        <mx:operation name="getProducts"/>
    </mx:WebService>

    <!-- Call the service using the default event handlers. -->
    <mx:Button label="Use Default Handlers" click="WeatherService.getProducts.send();"/>

    <!-- Call the service using the custom event handlers. -->
    <mx:Button label="Use Custom Handlers" click="setCustomHandlers();"/>

</mx:Application>
```

Notice that some properties are assigned to the token after the call to the remote service is made. In a multi-threaded language, there would be a race condition where the result comes back before the token is assigned. This situation is not a problem in ActionScript because the remote call cannot be initiated until the currently executing code finishes.

Passing parameters to a service

Flex provides two ways to pass parameters to a service call: *explicit parameter passing* and *parameter binding*. With explicit parameter passing, pass properties in the method that calls the service. With parameter binding, use data binding to populate the parameters from user interface controls or data models.

Using explicit parameter passing

When you use explicit parameter passing, you provide input to a service in the form of parameters to an ActionScript function. This way of calling a service closely resembles the way that you call methods in Java.

Explicit parameter passing with HTTPService components

When you use explicit parameter passing with an HTTPService component, you specify an object that contains name-value pairs as an argument to the `send()` method. A `send()` method parameter must be a simple base type such as Object. You cannot use complex nested objects because there is no generic way to convert them to name-value pairs.

The following examples show two ways to call an HTTP service using the `send()` method with a parameter.

```
<?xml version="1.0"?>
<!-- ds\rpc\RPCSend.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            public function callService():void {
                var params:Object = new Object();
                params.param1 = 'val1';
                myService.send(params);
            }
        ]]>
    </mx:Script>

    <mx:HTTPService
        id="myService"
        destination="catalog"
        useProxy="true"/>

    <!-- HTTP service call with a send() method that takes
      a variable as its parameter. The value of the variable is an Object. -->
    <mx:Button click="myService.send({param1: 'val1'});"/>

    <!-- HTTP service call with an object as a send() method parameter
      that provides query parameters. -->
    <mx:Button click="callService();"/>
</mx:Application>
```

Explicit parameter passing with WebService components

When using the WebService component, you call a service by directly calling the service method, or by calling the `Operation.send()` method of the Operation object that represents the operation. When you use explicit parameter passing, you specify the parameters as arguments to the method that you use to invoke the operation.

The following example shows MXML code for declaring a WebService component and calling a service using explicit parameter passing in the click event listener of a Button control. A ComboBox control provides data to the service, and event listeners handle the service-level result and fault events:

```
<?xml version="1.0"?>
<!-- ds\rpc\RPCParamPassingWS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
        ]]>
    </mx:Script>

    <mx:WebService
        id="employeeWS"
        destination="SalaryManager"/>

    <mx:ComboBox id="dept" width="150">
        <mx:dataProvider>
            <mx:ArrayCollection>
                <mx:source>
                    <mx:Object label="Engineering" data="ENG"/>
                    <mx:Object label="Product Management" data="PM"/>
                    <mx:Object label="Marketing" data="MKT"/>
                </mx:source>
            </mx:ArrayCollection>
        </mx:dataProvider>
    </mx:ComboBox>

    <mx:Button label="Get Employee List"
click="employeeWS.getList(dept.selectedItem.data);"/>
</mx:Application>
```

Using data binding to pass parameters

Parameter binding lets you copy data from user interface controls or models to request parameters. You typically declare data bindings in MXML. However, you can also define them in ActionScript. For more information about data binding, see the Flex Help Resource Center.

Binding with HTTPService components

Parameters to an HTTPService component correspond to query parameters of the requested URL. When an HTTP service takes query parameters, you can specify them by using the `request` property. The `request` property takes an Object of name-value pairs used as parameters to the URL. The names of the properties must match the names of the query parameters that the service expects. If the `HTTPService.contentType` property is set to `application/xml`, the `request` property must be an XML document.

When you use parameter binding, you call a service by using the `send()` method but specify no arguments to the method. The HTTPService component automatically adds the parameters specified by the `request` property to the request.

***Note:** If you do not specify a parameter to the* `send()` *method, the HTTPService component uses any query parameters specified in an* `<mx:request>` *tag.*

The following example binds the selected data of a ComboBox control to the `request` property:

```
<?xml version="1.0"?>
<!-- ds\rpc\HttpServiceParamBind.mxml. Compiles -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HTTPService
        id="employeeSrv"
        destination="catalog">
        <mx:request>
            <deptId>{dept.selectedItem.data}</deptId>
        </mx:request>
    </mx:HTTPService>

    <mx:HBox>
        <mx:Label text="Select a department:"/>
        <mx:ComboBox id="dept" width="150">
            <mx:dataProvider>
                <mx:ArrayCollection>
                    <mx:source>
                        <mx:Object label="Engineering" data="ENG"/>
                        <mx:Object label="Product Management" data="PM"/>
                        <mx:Object label="Marketing" data="MKT"/>
                    </mx:source>
                </mx:ArrayCollection>
            </mx:dataProvider>
        </mx:ComboBox>
        <mx:Button label="Get Employee List" click="employeeSrv.send();"/>
    </mx:HBox>

    <mx:DataGrid dataProvider="{employeeSrv.lastResult.employees.employee}"
        width="100%">
        <mx:columns>
            <mx:DataGridColumn dataField="name" headerText="Name"/>
            <mx:DataGridColumn dataField="phone" headerText="Phone"/>
            <mx:DataGridColumn dataField="email" headerText="Email"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

Binding with WebService components

When you use parameter binding with a WebService component, you typically declare an operation by using the `operation` property. Each `operation` property corresponds to an instance of the Operation class, which defines a `request` property that contains the XML nodes that the operation expects.

The following example binds the data of a selected ComboBox item to the `getList()` operation. When you use parameter binding, you call a service by using the `send()` method with no arguments:

```
<?xml version="1.0"?>
<!-- ds\rpc\WebServiceParamBind.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.utils.ArrayUtil;
            import mx.controls.Alert;
        ]]>
    </mx:Script>
```

```
    <mx:ArrayCollection
        id="employeeAC"
        source="{ArrayUtil.toArray(employeeWS.getList.lastResult)}"/>

    <mx:WebService
        id="employeeWS"
        destination="wsDest"
        showBusyCursor="true"
        fault="Alert.show(event.fault.faultString);">
        <mx:operation name="getList">
            <mx:request>
                <deptId>{dept.selectedItem.data}</deptId>
            </mx:request>
        </mx:operation>
    </mx:WebService>

    <mx:HBox>
        <mx:Label text="Select a department:"/>
        <mx:ComboBox id="dept" width="150">
            <mx:dataProvider>
                <mx:ArrayCollection>
                    <mx:source>
                        <mx:Object label="Engineering" data="ENG"/>
                        <mx:Object label="Product Management" data="PM"/>
                        <mx:Object label="Marketing" data="MKT"/>
                    </mx:source>
                </mx:ArrayCollection>
            </mx:dataProvider>
        </mx:ComboBox>
        <mx:Button label="Get Employee List" click="employeeWS.getList.send();"/>
    </mx:HBox>

    <mx:DataGrid dataProvider="{employeeAC}" width="100%">
        <mx:columns>
                <mx:DataGridColumn dataField="name" headerText="Name"/>
                <mx:DataGridColumn dataField="phone" headerText="Phone"/>
                <mx:DataGridColumn dataField=" to email" headerText="Email"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

You can manually specify an entire SOAP request body in XML with all of the correct namespace information defined in the `request` property. Set the value of the `format` attribute of the `request` property to `xml`, as the following example shows:

```
<?xml version="1.0"?>
<!-- ds\rpc\WebServiceSOAPRequest.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" verticalGap="10">
    <mx:WebService id="ws" wsdl="http://api.google.com/GoogleSearch.wsdl"
        useProxy="true">
        <mx:operation name="doGoogleSearch" resultFormat="xml">
            <mx:request format="xml">
                <ns1:doGoogleSearch xmlns:ns1="urn:GoogleSearch"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
                    <key xsi:type="xsd:string">XYZ123</key>
                    <q xsi:type="xsd:string">Balloons</q>
                    <start xsi:type="xsd:int">0</start>
                    <maxResults xsi:type="xsd:int">10</maxResults>
                    <filter xsi:type="xsd:boolean">true</filter>
                    <restrict xsi:type="xsd:string"/>
```

```
                    <safeSearch xsi:type="xsd:boolean">false</safeSearch>
                    <lr xsi:type="xsd:string" />
                    <ie xsi:type="xsd:string">latin1</ie>
                    <oe xsi:type="xsd:string">latin1</oe>
                </ns1:doGoogleSearch>
            </mx:request>
        </mx:operation>
    </mx:WebService>
</mx:Application>
```

Handling service results

Most service calls return data to the application. The way to access that data depends on which component you use:

- HTTPService

  Access the data by using the `HTTPService.lastResult` property, or in a `result` event by using the `ResultEvent.result` property.

- WebService

  The WebService component creates an Operation object for each operation supported by the associated web service. Access the data by using the `Operation.lastResult` property for the specific operation, or in a `result` event by using the `ResultEvent.result` property.

By default, the `resultFormat` property value of the HTTPService component and of the Operation class is `object`, and the data that is returned is represented as a simple tree of ActionScript objects. Flex interprets the XML data that a web service or HTTP service returns to appropriately represent base types, such as String, Number, Boolean, and Date. To work with strongly typed objects, populate those objects using the object tree that Flex creates.

WebService and HTTPService components both return anonymous Objects and Arrays that are complex types. If `makeObjectsBindable` is `true`, which it is by default, Objects are wrapped in mx.utils.ObjectProxy instances and Arrays are wrapped in mx.collections.ArrayCollection instances.

Note: *ColdFusion is not case-sensitive, so it internally represents all of its data in uppercase. Keep in mind case sensitivity when consuming a ColdFusion web service.*

When consuming data from a web service invocation, you can create an instance of a specific class instead of an Object or an Array. If you want Flex to create an instance of a specific class, use an mx.rpc.xml.SchemaTypeRegistry object and register a QName object with a corresponding ActionScript class. For more information, see "Customizing web service type mapping" on page 70.

Processing results in an event handler

Because calls to a remote service are asynchronous, you typically use events to signal that the service call has completed. A `result` event indicates that the result is available. The event object passed to the event handler is of type ResultEvent. Use the `result` property of the ResultEvent object to access the data returned by the service call, as the following example shows:

```
<?xml version="1.0"?>
<!-- ds\rpc\RPCResultEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            import mx.rpc.events.ResultEvent;
```

```
            import mx.rpc.events.FaultEvent;
            import mx.controls.Alert;

            // Handle result by populating the DataGrid control.
            // The operation returns an Array containing product ID, name, and price.
            public function handleResult(event:ResultEvent):void {
                // Make sure that the result can be cast to the correct type.
                if (event.result is ArrayCollection)
                {
                    // Cast the result to the correct type.
                    myDG.dataProvider=event.result as ArrayCollection;
                }
                else
                    myDG.dataProvider = null;
            }

            public function handleFault(event:FaultEvent):void {
                // Handle fault.
                Alert.show(event.fault.faultString, "Fault");
            }
        ]]>
    </mx:Script>

    <!-- Define a WebService component and connect to a service destination. -->
    <mx:WebService
        id="adbe_news"
        useProxy="true"
        destination="ws-catalog"
        result="handleResult(event);"
        fault="handleFault(event);"/>

    <!-- Call the getProducts() operation of the web service.
      The operation takes no parameters. -->
    <mx:Button click="adbe_news.getProducts();"/>

    <!-- Define a DataGrid control to diplay the reults of the web service. -->
    <mx:DataGrid id="myDG" width="100%" height="100%">
        <mx:columns>
            <mx:DataGridColumn dataField="productId" headerText="Product Id"/>
            <mx:DataGridColumn dataField="name" headerText="Name"/>
            <mx:DataGridColumn dataField="price" headerText="Price"/>
        </mx:columns>
    </mx:DataGrid>

</mx:Application>
```

The data type of the `ResultEvent.result` property is Object. Therefore, the event handler inspects the `ResultEvent.result` property to make sure that it can be cast to the required type. In this example, you want to cast it to ArrayCollection so that you can use it as the data provider for the DataGrid control. If the result cannot be cast to the ArrayCollection, set the data provider to null.

Binding a result to other objects

Rather than using an event handler, bind the results of an RPC component to the properties of other objects, including user interface components and data models. The `lastResult` object contains data from the last successful invocation of the component. Whenever a service request executes, the `lastResult` object is updated and any associated bindings are also updated.

In the following example, two properties of the `Operation.lastResult` object for the `GetWeather()` operation of a WebService component, `CityShortName` and `CurrentTemp`, are bound to the `text` properties of two TextArea controls. The `CityShortName` and `CurrentTemp` properties are returned when a user makes a request to the `MyService.GetWeather()` operation and provides a ZIP code as an operation request parameter.

```
<mx:TextArea text="{MyService.GetWeather.lastResult.CityShortName}"/>
<mx:TextArea text="{MyService.GetWeather.lastResult.CurrentTemp}"/>
```

Binding a result to an ArrayCollection object

You can bind the results to the `source` property of an ArrayCollection object, and use the ArrayCollection API to work with the data. You can then bind the ArrayCollection object to a complex property of a user interface component, such as a List, ComboBox, or DataGrid control.

In the following example, bind an `Operation.lastResult` object, `employeeWS.getList.lastResult`, to the `source` property of an ArrayCollection object. The ArrayCollection object is bound to the `dataProvider` property of a DataGrid control that displays the names, phone numbers, and e-mail addresses of employees.

```
<?xml version="1.0"?>
<!-- ds\rpc\BindingResultArrayCollection.mxml. Warnings on mx:Object  -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" verticalGap="10">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.utils.ArrayUtil;
        ]]>
    </mx:Script>

    <mx:WebService id="employeeWS" destination="employeeWS"
        showBusyCursor="true"
        fault="Alert.show(event.fault.faultString, 'Error');">
        <mx:operation name="getList">
            <mx:request>
                <deptId>{dept.selectedItem.data}</deptId>
            </mx:request>
        </mx:operation>
    </mx:WebService>

    <mx:ArrayCollection id="ac"
        source="{ArrayUtil.toArray(employeeWS.getList.lastResult)}"/>

    <mx:HBox>
        <mx:Label text="Select a department:"/>
        <mx:ComboBox id="dept" width="150">
            <mx:dataProvider>
                <mx:ArrayCollection>
                    <mx:source>
                        <mx:Object label="Engineering" data="ENG"/>
                        <mx:Object label="Product Management" data="PM"/>
                        <mx:Object label="Marketing" data="MKT"/>
                    </mx:source>
                </mx:ArrayCollection>
            </mx:dataProvider>
        </mx:ComboBox>
        <mx:Button label="Get Employee List" click="employeeWS.getList.send();"/>
    </mx:HBox>

    <mx:DataGrid dataProvider="{ac}" width="100%">
        <mx:columns>
                <mx:DataGridColumn dataField="name" headerText="Name"/>
```

```
                <mx:DataGridColumn dataField="phone" headerText="Phone"/>
                <mx:DataGridColumn dataField="email" headerText="Email"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

If you are unsure whether the result of a service call contains an Array or an individual object, you can use the `toArray()` method of the mx.utils.ArrayUtil class to convert it to an Array. If you pass the `toArray()` method to an individual object, it returns an Array with that object as the only Array element. If you pass an Array to the method, it returns the same Array.

For information about working with ArrayCollection objects, see the Flex Help Resource Center.

Binding a result to an XMLListCollection object

You can bind the results to an XMLListCollection object when the `resultFormat` property is set to `e4x`. When using an XMLListCollection object, you can use ECMAScript for XML (E4X) expressions to work with the data. You can then bind the XMLListCollection object to a complex property of a user interface component, such as a List, ComboBox, or DataGrid control.

In the following example, bind an `Operation.lastResult` object, `employeeWS.getList.lastResult`, to the `source` property of an XMLListCollection object. The XMLListCollection object is bound to the `dataProvider` property of a DataGrid control that displays the names, phone numbers, and e-mail addresses of employees.

Note: *To bind service results to an XMLListCollection, set the* `resultFormat` *property of your HTTPService or Operation object to* `e4x`*. The default value of this property is* `object`*.*

For more information on handling XML data, see "Handling results as XML with the E4X result format" on page 98.

```
<?xml version="1.0"?>
<!-- ds\rpc\BindResultXMLListCollection.mxml. -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
        ]]>
    </mx:Script>

    <mx:WebService id="employeeWS"
        destination="employeeWS"
        showBusyCursor="true"
        fault="Alert.show(event.fault.faultString, 'Error');">
        <mx:operation name="getList" resultFormat="e4x">
            <mx:request>
                <deptId>{dept.selectedItem.data}</deptId>
            </mx:request>
        </mx:operation>
    </mx:WebService>

    <mx:HBox>
        <mx:Label text="Select a department:"/>
        <mx:ComboBox id="dept" width="150">
            <mx:dataProvider>
                <mx:ArrayCollection>
                    <mx:source>
                        <mx:Object label="Engineering" data="ENG"/>
                        <mx:Object label="Product Management" data="PM"/>
                        <mx:Object label="Marketing" data="MKT"/>
                    </mx:source>
                </mx:ArrayCollection>
            </mx:dataProvider>
```

```
        </mx:ComboBox>

        <mx:Button label="Get Employee List"
            click="employeeWS.getList.send();"/>
    </mx:HBox>

    <mx:XMLListCollection id="xc"
        source="{employeeWS.getList.lastResult}"/>

    <mx:DataGrid dataProvider="{xc}" width="100%">
        <mx:columns>
            <mx:DataGridColumn dataField="name" headerText="Name"/>
            <mx:DataGridColumn dataField="phone" headerText="Phone"/>
            <mx:DataGridColumn dataField="email" headerText="Email"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

For information about working with XMLListCollection objects, see the Flex Help Resource Center.

Handling results as XML with the E4X result format

You can set the `resultFormat` property value of HTTPService components and WebService components to `e4x` to specify that the returned data is of type XML. Using a `resultFormat` of `e4x` is the preferred way to work with XML. You can also set the `resultFormat` property to `xml` to specify that the returned data is of type flash.xml.XMLNode, which is a legacy object for working with XML. Also, you can set the `resultFormat` property of HTTPService components to `flashvars` or `text` to create results as ActionScript objects that contain name-value pairs or as raw text, respectively. For more information, see *Adobe LiveCycle ES ActionScript Reference*.

When working with web service results that contain .NET DataSets or DataTables, it is best to set the `resultFormat` property to `object` to take advantage of specialized result handling for these data types. For more information, see "Handling web service results that contain .NET DataSets or DataTables" on page 101.

***Note:** If you want to use E4X syntax on service results, set the `resultFormat` property of your HTTPService or WebService component to `e4x`. The default value is `object`.*

When you set the `resultFormat` property of a WebService operation to `e4x`, you sometimes have to handle namespace information contained in the body of the SOAP envelope that the web service returns. The following example shows part of a SOAP body that contains namespace information. This data was returned by a web service that retrieves stock quotes. The namespace information is in boldface text.

```
<soap:Body>
    <GetQuoteResponse xmlns="http://ws.invesbot.com/">
        <GetQuoteResult>
            <StockQuote xmlns="">
                <Symbol>ADBE</Symbol>
                <Company>ADOBE SYSTEMS INC</Company>
                <Price>&lt;big&gt;&lt;b&gt;35.90&lt;/b&gt;&lt;/big&gt;</Price>
            </StockQuote>
        </GetQuoteResult>
    </GetQuoteResponse>
    ...
</soap:Body>
```

This `soap:Body` tag contains namespace information. Therefore, if you set the `resultFormat` property of the WebService operation to `e4x`, create a namespace object for the http://ws.invesbot.com/ namespace. The following example shows an application that does that:

```
<?xml version="1.0"?>
<!-- ds\rpc\WebServiceE4XResult1.mxml -->
```

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns="*">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;

            private namespace invesbot = "http://ws.invesbot.com/";
            use namespace invesbot;
        ]]>
    </mx:Script>

    <mx:WebService
        id="WS"
        destination="stockservice" useProxy="true"
        fault="Alert.show(event.fault.faultString, 'Error');">
        <mx:operation name="GetQuote" resultFormat="e4x">
            <mx:request>
                <symbol>ADBE</symbol>
            </mx:request>
        </mx:operation>
    </mx:WebService>

    <mx:HBox>
        <mx:Button label="Get Quote" click="WS.GetQuote.send();"/>
        <mx:Text text="{WS.GetQuote.lastResult.GetQuoteResult.StockQuote.Price}"/>
    </mx:HBox>
</mx:Application>
```

Optionally, you can create a variable for a namespace and access it in a binding to the service result, as the following example shows:

```
<?xml version="1.0"?>
<!-- ds\rpc\WebServiceE4XResult2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns="*">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;

            public var invesbot:Namespace =
            new Namespace("http://ws.invesbot.com/");
        ]]>
    </mx:Script>

    <mx:WebService
        id="WS"
        destination="stockservice" useProxy="true"
        fault="Alert.show(event.fault.faultString, 'Error');">
        <mx:operation name="GetQuote" resultFormat="e4x">
            <mx:request>
                <symbol>ADBE</symbol>
            </mx:request>
        </mx:operation>
    </mx:WebService>

    <mx:HBox>
        <mx:Button label="Get Quote" click="WS.GetQuote.send()"/>
       <mx:Text text="{WS.GetQuote.lastResult.invesbot::GetQuoteResult.StockQuote.Price}"/>
    </mx:HBox>
</mx:Application>
```

You use E4X syntax to access elements and attributes of the XML that is returned in a `lastResult` object. You use different syntax, depending on whether there is a namespace or namespaces declared in the XML.

No namespace specified

The following example shows how to get an element or attribute value when no namespace is specified on the element or attribute:

```
var attributes:XMLList = XML(event.result).Description.value;
```

The previous code returns `xxx` for the following XML document:

```
<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <Description>
        <value>xxx</value>
    </Description>
</RDF>
```

Any namespace specified

The following example shows how to get an element or attribute value when any namespace is specified on the element or attribute:

```
var attributes:XMLList = XML(event.result).*::Description.*::value;
```

The previous code returns `xxx` for either one of the following XML documents:

XML document one:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description>
        <rdf:value>xxx</rdf:value>
    </rdf:Description>
</rdf:RDF>
```

XML document two:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:cm="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <cm:Description>
        <rdf:value>xxx</rdf:value>
    </cm:Description>
</rdf:RDF>
```

Specific namespace specified

The following example shows how to get an element or attribute value when the declared rdf namespace is specified on the element or attribute:

```
var rdf:Namespace = new Namespace("http://www.w3.org/1999/02/22-rdf-syntax-ns#");
var attributes:XMLList = XML(event.result).rdf::Description.rdf::value;
```

The previous code returns `xxx` for the following XML document:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description>
        <rdf:value>xxx</rdf:value>
        <nsX:value>yyy</nsX:value>
    </rdf:Description>
</rdf:RDF>
```

The following example shows an alternate way to get an element or attribute value when the declared rdf namespace is specified on the element or attribute:

```
namespace rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#";
use namespace rdf;
```

```
var attributes:XMLList = XML(event.result).rdf::Description.rdf::value;
```

The previous code also returns `xxx` for the following XML document:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description>
        <rdf:value>xxx</rdf:value>
    </rdf:Description>
</rdf:RDF>
```

Handling web service results that contain .NET DataSets or DataTables

Web services written with the Microsoft .NET Framework can return special .NET DataSet or DataTable objects to the client. A .NET web service provides a basic WSDL document without information about the type of data that it manipulates. When the web service returns a DataSet or a DataTable, data type information is embedded in an XML Schema element in the SOAP message to specify how to process the rest of the message. To best handle results from this type of web service, set the `resultFormat` property of a Flex WebService operation to `object`. You can optionally set the `resultFormat` property of the operation to `e4x`. However, the XML and E4X formats are inconvenient because you must navigate through the unusual structure of the response and implement workarounds if you want to bind the data to another object.

When you set the `resultFormat` property of a WebService operation to `object`, a DataTable or DataSet returned from a .NET web service is automatically converted to an object with a Tables property, which contains a map of one or more DataTable objects. Each DataTable object from the Tables map contains two properties: Columns and Rows. The Rows property contains the data. The event.result object gets the following properties corresponding to DataSet and DataTable properties in .NET. Arrays of DataSets or DataTables have the same structures described here, but are nested in a top-level Array on the result object.

Property	Description
`result.Tables`	Map of table names to objects that contain table data.
`result.Tables["`*someTable*`"].Columns`	Array of column names in the order specified in the DataSet or DataTable schema for the table.
`result.Tables["`*someTable*`"].Rows`	Array of objects that represent the data of each table row. For example, {columnName1:value, columnName2:value, columnName3:value}.

The following MXML application populates a DataGrid control with DataTable data returned from a .NET web service.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns="*" xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical">
    <mx:WebService
        id="nwCL"
        wsdl="http://localhost/data/CustomerList.asmx?wsdl"
        result="onResult(event)"
        fault="onFault(event)" />
    <mx:Button label="Get Single DataTable" click="nwCL.getSingleDataTable()"/>
    <mx:Button label="Get MultiTable DataSet" click="nwCL.getMultiTableDataSet()"/>
    <mx:Panel id="dataPanel" width="100%" height="100%" title="Data Tables"/>

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.controls.DataGrid;
            import mx.rpc.events.FaultEvent;
            import mx.rpc.events.ResultEvent;
```

```
            private function onResult(event:ResultEvent):void {
                // A DataTable or DataSet returned from a .NET webservice is
                // automatically converted to an object with a "Tables" property,
                // which contains a map of one or more dataTables.
                if (event.result.Tables != null)
                {
                    // clean up panel from previous calls.
                    dataPanel.removeAllChildren();

                    for each (var table:Object in event.result.Tables)
                    {
                        displayTable(table);
                    }

                    // Alternatively, if a table's name is known beforehand,
                    // it can be accessed using this syntax:
                    var namedTable:Object = event.result.Tables.Customers;
                    //displayTable(namedTable);
                }
            }

            private function displayTable(tbl:Object):void {
                var dg:DataGrid = new DataGrid();
                dataPanel.addChild(dg);
                // Each table object from the "Tables" map contains two properties:
                // "Columns" and "Rows". "Rows" is where the data is, so we can set
                // that as the dataProvider for a DataGrid.
                dg.dataProvider = tbl.Rows;
            }

            private function onFault(event:FaultEvent):void {
                Alert.show(event.fault.toString());
            }
        ]]>
    </mx:Script>

</mx:Application>
```

The following example shows the .NET C# class that is the back-end web service implementation called by the Flex application. This class uses the Microsoft SQL Server Northwind sample database:

```
<%@ WebService Language="C#" Class="CustomerList" %>
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Web.Services.Description;
using System.Data;
using System.Data.SqlClient;
using System;

public class CustomerList : WebService {
    [WebMethod]
    public DataTable getSingleDataTable() {
        string cnStr = "[Your_Database_Connection_String]";
        string query = "SELECT TOP 10 * FROM Customers";
        SqlConnection cn = new SqlConnection(cnStr);
        cn.Open();
        SqlDataAdapter adpt = new SqlDataAdapter(new SqlCommand(query, cn));
        DataTable dt = new DataTable("Customers");

        adpt.Fill(dt);
        return dt;
```

```
    }

    [WebMethod]
    public DataSet getMultiTableDataSet() {
        string cnStr = "[Your_Database_Connection_String]";
        string query1 = "SELECT TOP 10 CustomerID, CompanyName FROM Customers";
        string query2 = "SELECT TOP 10 OrderID, CustomerID, ShipCity,
        ShipCountry FROM Orders";
        SqlConnection cn = new SqlConnection(cnStr);
        cn.Open();

        SqlDataAdapter adpt = new SqlDataAdapter(new SqlCommand(query1, cn));
        DataSet ds = new DataSet("TwoTableDataSet");
        adpt.Fill(ds, "Customers");

        adpt.SelectCommand = new SqlCommand(query2, cn);
        adpt.Fill(ds, "Orders");

        return ds;
    }
}
```

Using capabilities specific to WebService components

Flex applications can interact with web services that define their interfaces in a Web Services Description Language 1.1 (WSDL 1.1) document, which is available as a URL. WSDL is a standard format for describing the messages that a web service understands, the format of its responses to those messages, the protocols that the web service supports, and where to send messages. The Flex web service API generally supports SOAP 1.1, XML Schema 1.0 (versions 1999, 2000 and 2001), and WSDL 1.1 rpc-encoded, and rpc-literal, document-literal (bare and wrapped style parameters). The two most common types of web services use RPC-encoded or document-literal SOAP bindings; the terms *encoded* and *literal* indicate the type of WSDL-to-SOAP mapping that a service uses.

Note: *Flex does not support the following XML schema types: union, default, or list. Flex also does not support the following data types: duration, gMonth, gYear, gYearMonth, gDay, gMonthDay, Name, Qname, NCName, anyURI, or language. These data types are treated as Strings and not validated. Flex supports any URL but treats it like a String.*

Flex applications support web service requests and results that are formatted as Simple Object Access Protocol (SOAP) messages. SOAP provides the definition of the XML-based format that you can use for exchanging structured and typed information between a web service client, such as a Flex application, and a web service.

Adobe Flash Player operates within a security sandbox that limits what Flex applications and other Flash Player applications can access over HTTP. Flash Player applications are only allowed HTTP access to resources on the same domain and by the same protocol from which they were served. This restriction presents a problem for web services, because they are typically accessed from remote locations. The Flex proxy, available in BlazeDS, intercepts requests to remote web services, redirects the requests, and then returns the responses to the client.

If you are not using BlazeDS, you can access web services in the same domain as your Flex application. Or, a crossdomain.xml file that allows access from the domain of the application must be installed on the web server hosting the RPC service. For more information, see the Adobe Flex 3 documentation.

Note: *If you are using Flash Player version 9,0,124,0 or later, the crossdomain.xml file has a new tag called* `<allow-http-request-headers-from>` *that you use to set header-sending rights. For web services, make sure to set the* `headers` *attribute of the* `<allow-http-request-headers-from>` *tag to* `SOAPAction`*. For more information, see* *http://www.adobe.com/devnet/flashplayer/articles/flash_player9_security_update.html* *and* *http://kb.adobe.com/selfservice/viewContent.do?externalId=kb403185&sliceId=2*.

Reading WSDL documents

View a WSDL document in a web browser, a simple text editor, an XML editor, or a development environment such as Adobe Dreamweaver, which contains a built-in utility for displaying WSDL documents in an easy-to-read format.

For a complete description of the format of a WSDL document, see http://www.w3.org/TR/wsdl.

RPC-oriented operations and document-oriented operations

A WSDL file can specify either remote procedure call-oriented (RPC) or document-oriented (document/literal) operations. Flex supports both operation styles.

When calling an RPC-oriented operation, a Flex application sends a SOAP message that specifies an operation and its parameters. When calling a document-oriented operation, a Flex application sends a SOAP message that contains an XML document.

In a WSDL document, each `<port>` tag has a `binding` property that specifies the name of a particular `<soap:binding>` tag, as the following example shows:

```
<binding name="InstantMessageAlertSoap" type="s0:InstantMessageAlertSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
        style="document"/>
```

The `style` property of the associated `<soap:binding>` tag determines the operation style. In this example, the style is `document`.

Any operation in a service can specify the same style or override the style that is specified for the port associated with the service, as the following example shows:

```
<operation name="SendMSN">
    <soap:operation soapAction="http://www.bindingpoint.com/ws/imalert/
        SendMSN"style="document"/>
```

Stateful web services

BlazeDS can maintain the state of web service endpoints. If the web service uses cookies to store session information, BlazeDS uses Java server sessions to maintain the state of the web service. This capability acts as an intermediary between Flex applications and web services. It adds the identity of an endpoint to whatever the endpoint passes to a Flex application. If the endpoint sends session information, the Flex application receives it. This capability requires no configuration.

Working with SOAP headers

A SOAP header is an optional tag in a SOAP envelope that usually contains application-specific information, such as authentication information.

Adding SOAP headers to web service requests

Some web services require that you pass a SOAP header when you call an operation. Add a SOAP header to all web service operations or individual operations by calling the `addHeader()` or `addSimpleHeader()` method of the WebService or Operation object.

When you use the `addHeader()` method, you first create SOAPHeader and QName objects separately. The `addHeader()` method has the following signature:

```
addHeader(header:mx.rpc.soap.SOAPHeader):void
```

To create a SOAPHeader object, you use the following constructor:

```
SOAPHeader(qname:QName, content:Object)
```

The `content` parameter of the `SOAPHeader()` constructor is a set of name-value pairs based on the following format:

```
{name1:value1, name2:value2}
```

To create the QName object in the first parameter of the `SOAPHeader()` method, you use the following constructor:

```
QName(uri:String, localName:String)
```

The `addSimpleHeader()` method is a shortcut for a single name-value SOAP header. When you use the `addSimpleHeader()` method, you create SOAPHeader and QName objects in parameters of the method. The `addSimpleHeader()` method has the following signature:

```
addSimpleHeader(qnameLocal:String, qnameNamespace:String, headerName:String,
    headerValue:Object):void
```

The `addSimpleHeader()` method takes the following parameters:

- `qnameLocal` is the local name for the header QName.
- `qnameNamespace` is the namespace for the header QName.
- `headerName` is the name of the header.
- `headerValue` is the value of the header. This value can be a String if it is a simple value, an Object that undergoes basic XML encoding, or XML if you want to specify the header XML yourself.

The following calls to the `addSimpleHeader()` and `addSimpleHeader()` methods are equivalent:

```
addHeader(new Qname(qNs,qLocal), {name:val});
addSimpleHeader(qLocal, qNs, name, val);
```

Both methods add a SOAPHeader object to a collection of headers. Each SOAPHeader is encoded as follows:

```
<qnamePrefix:qnameLocal>
    content
</qnamePrefix:qnameLocal>
```

If the `content` parameter contains simple data, its String representation is used. If it contains an Object, its structure is converted to XML. For example, if the `content` parameter passed to the method contains the following data:

```
{name:value}
```

The SOAPHeader is encoded as follows:

```
<qnamePrefix:qnameLocal>
    <headerName>headerValue</headerName>
</qnamePrefix:qnameLocal>
```

If the `content` parameter contains a property with the same name as `qnameLocal`, the value of that property is used as the header content. Therefore, if `qnameLocal` equals `headerName`, the SOAPHeader object is encoded as follows:

```
<qnamePrefix:headerName>
    headerValue
</qnamePrefix:headerName>
```

The code in the following example shows how to use the `addHeader()` method and the `addSimpleHeader()` method to add a SOAP header. The methods are called in the `headers()` function, and the event listener is assigned in the `load` property of an WebService component:

```
<?xml version="1.0"?>
<!-- ds\rpc\WebServiceAddHeader.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.rpc.soap.SOAPHeader;

            private var header1:SOAPHeader;
            private var header2:SOAPHeader;
```

```
            public function headers():void {

                // Create QName and SOAPHeader objects.
                var q1:QName = new QName("http://soapinterop.org/xsd", "Header1");
                header1 = new SOAPHeader(q1, {string:"bologna",int:"123"});
                header2 = new SOAPHeader(q1, {string:"salami",int:"321"});

                // Add the header1 SOAP Header to all web service requests.
                ws.addHeader(header1);

                // Add the header2 SOAP Header to the getSomething operation.
                ws.getSomething.addHeader(header2);

                // Within the addSimpleHeader method,
                // which adds a SOAP header to web
                //service requests, create SOAPHeader and QName objects.
                ws.addSimpleHeader("header3", "http://soapinterop.org/xsd", "foo", "bar");
            }
        ]]>
    </mx:Script>

    <mx:WebService id="ws"
        destination="wsDest"
        load="headers();"/>
</mx:Application>
```

Clearing SOAP headers

Use the `clearHeaders()` method of a WebService or Operation object to remove SOAP headers that you added to the object, as the following example shows:

```
<?xml version="1.0"?>
<!-- ds\rpc\WebServiceClearHeader.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- The value of the destination property is for demonstration only
        and is not a real destination. -->

    <mx:Script>
        <![CDATA[
            import mx.rpc.*;
            import mx.rpc.soap.SOAPHeader;

            private function headers():void {
                // Create QName and SOAPHeader objects.
                var q1:QName = new QName("Header1", "http://soapinterop.org/xsd");
                var header1:SOAPHeader = new SOAPHeader(q1, {string:"bologna",int:"123"});
                var header2:SOAPHeader = new SOAPHeader(q1, {string:"salami",int:"321"});
                    // Add the header1 SOAP Header to all web service request.
                ws.addHeader(header1);
                // Add the header2 SOAP Header to the getSomething operation.
                ws.getSomething.addHeader(header2);

                // Within the addSimpleHeader method, which adds a SOAP header to all
                // web service requests, create SOAPHeader and QName objects.
                ws.addSimpleHeader("header3","http://soapinterop.org/xsd", "foo", "bar");
            }

            // Clear SOAP headers added at the WebService and Operation levels.
            private function clear():void {
```

```
                ws.clearHeaders();
                ws.getSomething.clearHeaders();
            }
        ]]>
    </mx:Script>

    <mx:WebService id="ws"
        destination="wsDest"
        load="headers();"/>

    <mx:HBox>
        <mx:Button label="Clear headers and run again"
            click="clear();"/>
    </mx:HBox>
</mx:Application>
```

Redirecting a web service to a different URL

Some web services require that you change to a different endpoint URL after you process the WSDL and make an initial call to the web service. For example, suppose you want to use a web service that requires you to pass security credentials. After you call the web service to send login credentials, it accepts the credentials and returns the actual endpoint URL that is required to use the business operations. Before calling the business operations, change the `endpointURI` property of your WebService component.

The following example shows a `result` event listener that stores the endpoint URL that a web service returns in a variable, and then sets the endpoint URL for subsequent requests:

```
public function onLoginResult(event:ResultEvent):void {

    //Extract the new service endpoint from the login result.
    var newServiceURL = event.result.serverUrl;

    // Redirect all service operations to the URL received in the login result.
    serviceName.endpointURI=newServiceURL;
}
```

A web service that requires you to pass security credentials can also return an identifier that you must attach in a SOAP header for subsequent requests; for more information, see "Working with SOAP headers" on page 104.

Handling asynchronous calls to services

Because ActionScript code executes asynchronously, if you allow concurrent calls to a service, ensure that your code handles the results appropriately. By default, making a request to a web service operation that is already executing does not cancel the existing request. In a Flex application in which a service can be called from multiple locations, the service might respond differently in different contexts.

When you design a Flex application, consider whether the application requires disparate data sources, and the number of types of services that the application requires. The answers to these questions help determine the level of abstraction that you provide in the data layer of the application.

In a simple application, user interface components call services directly. In applications that are slightly larger, business objects call services. In still larger applications, business objects interact with service broker objects that call services.

To understand the results of asynchronous service calls to objects in an application, you need a good understanding of scoping in ActionScript. For more information, see the *Adobe Flex 3 Developer Guide*.

Using the Asynchronous Completion Token design pattern

Flex is a service-oriented framework in which code executes asynchronously, therefore, it lends itself well to the Asynchronous Completion Token (ACT) design pattern. This design pattern efficiently dispatches processing within a client in response to the completion of asynchronous operations that the client invokes. For more information, see www.cs.wustl.edu/~schmidt/PDF/ACT.pdf.

When you use the ACT design pattern, you associate application-specific actions and state with responses that indicate the completion of asynchronous operations. For each asynchronous operation, you create an ACT that identifies the actions and state that are required to process the results of the operation. When the result is returned, you can use its ACT to distinguish it from the results of other asynchronous operations. The client uses the ACT to identify the state required to handle the result.

An ACT for a particular asynchronous operation is created before the operation is called. While the operation is executing, the client continues executing. When the service sends a response, the client uses the ACT that is associated with the response to perform the appropriate actions.

When you call a Flex remote object service, web service, or HTTP service, Flex returns an instance of the mx.rpc.AsyncToken class. If you use the default `concurrency` value of `multiple`, you can use the token returned by the data service's `send()` method to handle the specific results of each concurrent call to the same service.

You can add information to the token when it is returned, and then in a result event listener you can access the token as `event.token`. This situation is an implementation of the ACT design pattern that uses the token of each data service call as an ACT. How you use the ACT design pattern in your own code depends on your requirements. For example, you could attach simple identifiers to individual calls, or more complex objects that perform their own set of functionality, or functions that a central listener calls.

The following example shows a simple implementation of the ACT design pattern. This example uses an HTTP service and attaches a simple variable to the token.

```
<mx:HTTPService id="MyService" destination="httpDest" result="resultHandler(event)"/>

<mx:Script>
    <![CDATA[
        ...
        public function storeCall():void {
            // Create a variable called call to store the instance
            // of the service call that is returned.
            var call:Object = MyService.send();

            // Add a variable to the token that is returned.
            // You can name this variable whatever you want.
            call.marker = "option1";
            ...
        }

        // In a result event listener, execute conditional
        // logic based on the value of call.marker.
        private function resultHandler(event:ResultEvent):void {
            var call:Object = event.token
            if (call.marker == "option1") {
            //do option 1
            }
            else
            ...
        }
    ]]>
</mx:Script>
```

Making a service call when another call is completed

Another common requirement when using data services is the dependency of one service call on the result of another. Your code must not make the second call until the result of the first call is available. Make the second service call in the result event listener of the first, as the following example shows:

```
<mx:Script>
    <![CDATA[
        // Call the getForecastWithSalesInput operation with the result of the
        // getCurrentSales operation.
        public function resultHandler(evt:ResultEvent):void {
            ws.setForecastWithSalesInput(evt.token.currentsales);
            //Or some variation that uses data binding.
        }
    ]]>
</mx:Script>

<mx:WebService id="ws" destination="wsDest"...>
    <mx:operation name="getCurrentSales" result="resultHandler(event)"/>
    <mx:operation name="setForecastWithSalesInput"/>
</mx:WebService>
```

Chapter 8: Using the Remoting Service

The Remoting Service lets a client application access the methods of server-side Java objects, without configuring the objects as web services. To access the server-side Java object, the client application uses the RemoteObject component. Use the RemoteObject component instead of a WebService or HTTPService component when objects are not already published as web services, web services are not used in your environment, or you would rather use Java objects than web services or HTTP services.

The RemoteObject component is one of three Remote Procedure Call (RPC) components used by a client application. The others are the HTTPService component and WebService component. Before using the RemoteObject component, be familiar with the basic information about RPC components. For more information, see "Using HTTP and web services" on page 73.

Topics

RemoteObject component 110
Configuring a destination 114
Calling a service 115
Handling events. 117
Passing parameters 117
Handling results 119
Accessing EJBs and other objects in JNDI. 120

RemoteObject component

You declare RemoteObject components in MXML or ActionScript to connect to remote services. Use the RemoteObject component to call methods on a Java class or ColdFusion component.

Note: *This documentation describes how to connect to Java classes. For information on connecting to ColdFusion components, see the ColdFusion documentation.*

A destination for a RemoteObject component is a Java class defined as the source of a Remoting Service destination. Destination definitions provide centralized administration of remote services. They also enable you to use basic or custom authentication to secure access to destinations. You can choose from several different transport channels, including secure channels, for sending data to and from destinations. Additionally, you can use the server-side logging capability to log remote service traffic.

You can also use RemoteObject components with PHP and .NET objects in conjunction with third-party software, such as the open source projects AMFPHP and SabreAMF, and Midnight Coders WebORB. For more information, see the following websites:

- AMFPHP http://amfphp.sourceforge.net/
- SabreAMF http://www.osflash.org/sabreamf
- Midnight Coders WebORB http://www.themidnightcoders.com/

Remoting Service channels

With the Remoting Service, you often use an AMFChannel. The AMFChannel uses binary AMF encoding over HTTP. If binary data is not allowed, then you can use an HTTPChannel, which is AMFX (AMF in XML) over HTTP. Message channels are typically defined in the services-config.xml file, in the `channels` section under the `services-config` element. For more information on channels, see "BlazeDS architecture" on page 27.

Using a RemoteObject component

The following example shows a RemoteObject component that connects to a destination, sends a request to the data source in the `click` event of a Button control, and displays the result data in the `text` property of a TextArea control:

```
<?xml version="1.0"?>
<!-- ds\rpc\RPCIntroExample1.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;
            import mx.controls.Alert;

            public function handleResult(event:ResultEvent):void {
                // Handle result by populating the TextArea control.
                outputResult.text=remoteService.getData.lastResult.prop1;
            }

            public function handleFault(event:FaultEvent):void {
                // Handle fault.
                Alert.show(event.fault.faultString, "Fault");
            }
        ]]>
    </mx:Script>

    <!-- Connect to a service destination.-->
    <mx:RemoteObject id="remoteService"
        destination="census"
        result="handleResult(event);"
        fault="handleFault(event);"/>

    <!-- Provide input data for calling the service. -->
    <mx:TextInput id="inputText"/>

    <!-- Call the web service, use the text in a TextInput control as input data.-->
    <mx:Button click="remoteService.getData(inputText.text)"/>

    <!-- Display results data in the user interface. -->
    <mx:TextArea id="outputResult"/>
</mx:Application>
```

Defining remote Java objects

One difference between Remoting Service destinations and HTTP service and web service destinations is that in Remoting Service destinations you host the remote Java object in your BlazeDS web application and reference it by using a destination. With HTTP service and web service destinations, you typically configure the destination to access a remote service, external to the web application. A developer is responsible for writing and compiling the Java class and adding it to the web application classpath by placing it in the WEB-INF\classes or WEB-INF\lib directory.

You can use any plain old Java object (POJO) that is available in the web application classpath as the source of the Remoting Service destination. The class must have a zero-argument constructor so that BlazeDS can construct an instance.

The following example shows a Remoting Service destination definition in the remoting-config.xml file. The `source` element specifies the fully qualified name of a class in the classpath of the web application.

```
<destination id="census">
    <properties>
        <source>flex.samples.census.CensusService</source>
    </properties>
```

</destination>

The following example shows the corresponding source code of the Java class that is referenced in the destination definition:

```
package flex.samples.census;

import java.util.ArrayList;
import java.util.List;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import flex.samples.ConnectionHelper;

public class CensusService
{
    public List getElements(int begin, int count)
    {

        long startTime = System.currentTimeMillis();

        Connection c = null;
        List list = new ArrayList();

        String sql = "SELECT id, age, classofworker, education, maritalstatus, race,
            sex FROM census WHERE id > ? AND id <= ? ORDER BY id ";

        try {

            c = ConnectionHelper.getConnection();
            PreparedStatement stmt = c.prepareStatement(sql);
            stmt.setInt(1, begin);
            stmt.setInt(2, begin + count);
            ResultSet rs = stmt.executeQuery();

            while (rs.next()) {
                CensusEntryVO ce = new CensusEntryVO();
                ce.setId(rs.getInt("id"));
                ce.setAge(rs.getInt("age"));
                ce.setClassOfWorker(rs.getString("classofworker"));
                ce.setEducation(rs.getString("education"));
                ce.setMaritalStatus(rs.getString("maritalstatus"));
                ce.setRace(rs.getString("race"));
                ce.setSex(rs.getString("sex"));
                list.add(ce);
            }
        } catch (SQLException e) {
            e.printStackTrace();
```

```
        } finally {
            try {
                c.close();
            } catch (Exception ignored) {
            }
        }

        return list;
    }
}
```

Placing Java objects in the classpath

The Remoting Service lets you access stateless and stateful objects that are in the classpath of the BlazeDS web application. Place class files in the WEB-INF\classes directory to add them to the classpath. Place Java Archive (JAR) files in the WEB-INF\lib directory to add them to the classpath.

Specify the fully qualified class name in the `source` property of a Remoting Service destination in the remoting-config.xml file. The class also must define a constructor that takes no arguments.

Converting ActionScript data to and from Java data

When you send data from a Flex application to a Java object, the data is automatically converted from an ActionScript data type to a Java data type. An object returned from a Java method is converted from Java to ActionScript. For a complete description of how data is converted, see "Data serialization" on page 56.

Reserved method names for the RemoteObject component

If a remote method has the same name as a method defined by the RemoteObject class, or by any of its parent classes, then you cannot call the remote method directly. The RemoteObject class defines the following method names; do not use these names as method names in your Java class:

```
disconnect()
getOperation()
hasOwnProperty()
initialized()
isPrototypeOf()
logout()
propertyIsEnumerable()
setCredentials()
setPropertyIsEnumerable()
setRemoteCredentials()
toString()
valueOf()
```

For a complete list of reserved method names, see the *Adobe LiveCycle ES ActionScript Reference*. Also, do not begin Java method names with the underscore (_) character.

If a remote method name matches a reserved method name, you can use the following ActionScript method with a RemoteObject or WebService component to return an Operation object that represents the method:

```
public function getOperation(name:String):Operation
```

For example, if a remote method is called `hasOwnProperty()`, create an Operation object, as the following example shows:

```
public var myRemoteObject:RemoteObject = new RemoteObject();
myRemoteObject.destination = "ro-catalog";
public var op:Operation = myRemoteObject.getOperation("hasOwnProperty");
```

Invoke the remote method by using the `Operation.send()` method, as the following example shows:

```
op.send();
```

Configuring a destination

You configure RemoteObject destinations in the Remoting Service definition in the remoting-config.xml file. The following example shows a basic server-side configuration for a Remoting Service in the remoting-config.xml file:

```
<service id="remoting-service"
    class="flex.messaging.services.RemotingService">

    <adapters>
        <adapter-definition id="java-object"
            class="flex.messaging.services.remoting.adapters.JavaAdapter"
            default="true"/>
    </adapters>

    <default-channels>
        <channel ref="samples-amf"/>
    </default-channels>

    <destination id="restaurant">
        <properties>
            <source>samples.restaurant.RestaurantService</source>
            <scope>application</scope>
        </properties>
    </destination>
</service>
```

The `class` attribute of the `<service>` tag specifies the RemotingService class. RemoteObject components connect to RemotingService destinations.

The adapter is server-side code that interacts with the Java class. Because you set the JavaAdapter as the default adapter, all destinations use it unless the destination explicitly specifies another adapter.

Use the `source` and `scope` elements of a Remoting Service destination definition to specify the Java object that the destination uses. Additionally, specify whether the destination is available in the `request` scope (stateless), the `application` scope, or the `session` scope. The following table describes these properties:

Element	Description
`source`	Fully qualified class name of the Java object (remote object).
`scope`	Indicates whether the object is available in the `request` scope, the `application` scope, or the `session` scope. Use the `request` scope when you configure a Remoting Service destination to access stateless objects. With the `request` scope, the server creates an instance of the Java class on each request. Use the `request` scope if you are storing the object in the application or session scope causes memory problems. When you use the `session` scope, the server creates an instance of the Java object once on the server for the session. For example, multiple tabs in the same web browser share the same session. If you open a Flex application in one tab, any copy of that application running in another tab accesses the same Java object. When you use the `application` scope, the server creates an instance of the Java object once on the server for the entire application. The default value is `request`.

For Remoting Service destinations, you can declare destinations that only allow invocation of methods that are explicitly included in an include list. Any attempt to invoke a method that is not in the `include-methods` list results in a fault. For even finer grained security, you can assign a security constraint to one or more of the methods in the `include-methods` list. If a destination-level security constraint is defined, it is tested first. Following that, the method-level constraints are checked. For more information, see "Configuring a destination to use a security constraint" on page 159.

Calling a service

Define the RemoteObject components in your client-side Flex application in MXML or ActionScript. The following example defines a RemoteObject component using both techniques:

```
<?xml version="1.0"?>
<!-- ds\rpc\ROInAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="useRemoteObject();">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.rpc.remoting.mxml.RemoteObject;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;

            public var asService:RemoteObject;

            public function useRemoteObject():void {
                asService = new RemoteObject();
                asService.destination = "ro-catalog";
                asService.getList.addEventListener("result", getListResultHandler);
                asService.addEventListener("fault", faultHandler);
                asService.getList();
            }

            public function getListResultHandler(event:ResultEvent):void {
                 // Handle the result by accessing the event.result property.
            }

            public function faultHandler (event:FaultEvent):void {
             // Deal with event.fault.faultString, etc.
                Alert.show(event.fault.faultString, 'Error');
            }
        ]]>
    </mx:Script>

    <!-- Define the RemoteObject component in MXML. -->
    <mx:RemoteObject
        id="mxmlService"
        destination="ro-catalog"
        result="getListResultHandler(event);"
        fault="faultHandler(event);"/>

    <mx:Button label="MXML" click="mxmlService.getList();"/>
    <mx:Button label="AS" click="asService.getList();"/>

</mx:Application>
```

The destination specifies the Java class associated with the remote service. A Java class can expose multiple methods, corresponding to multiple operations. In this example, you directly call the `getList()` operation in response to a `click` event of a Button control.

Using the Operation class with the RemoteObject component

Because a single RemoteObject component can invoke multiple operations, the component requires a way to represent information specific to each operation. Therefore, for every operation, the component creates an mx.rpc.remoting.mxml.Operation object.

The name of the Operation object corresponds to the name of the operation. From the example shown in "Calling a service" on page 115, you access the Operation object that corresponds to the `getList()` operation by accessing the Operation object named `getList`, as the following code shows:

```
// The Operation object has the same name as the operation, without the trailing parentheses.
var myOP:Operation = mxmlService.getList;
```

The Operation object contains properties that you use to set characteristics of the operation, such as the arguments passed to the operation, and to hold any data returned by the operation. You access the returned data by using the `Operation.lastResult` property.

Invoke the operation by referencing it relative to the RemoteObject component, as the following example shows:

```
mxmlService.getList();
```

Alternatively, invoke an operation by calling the `Operation.send()` method, as the following example shows:

```
mxmlService.getList.send();
```

Defining multiple operations for the RemoteObject component

When a service defines multiple operations, you define multiple methods for the RemoteObject component and specify the attributes for each method, as the following example shows:

```
<mx:RemoteObject
    id="mxmlService"
    destination="ro-catalog"
    result="roResult(event);">
        <mx:method name="getList" fault="getLFault(event);"/>
        <mx:method name="updateList" fault="updateLFault(event);"/>
        <mx:method name="deleteListItem" fault="deleteLIFault(event);"/>
</mx:RemoteObject>
```

The `name` property of an `<mx:method>` tag must match one of the operation names. The RemoteObject component creates a separate Operation object for each operation.

Each operation can rely on the event handlers and characteristics defined by the RemoteObject component. However, the advantage of defining the methods separately is that each operation can specify its own event handlers, its own input parameters, and other characteristics. In this example, the RemoteObject component defines the result handler for all three operations, and each operation defines its own fault handler. For an example that defines input parameters, see "Using parameter binding to pass parameters to the RemoteObject component" on page 118.

Note: *The Flex compiler defines the method property of the RemoteObject class; it does not correspond to an actual property of the RemoteObject class.*

Setting the concurrency property

The `concurrency` property of the `<mx:method>` tag indicates how to handle multiple calls to the same method. By default, making a new request to an operation or method that is already executing does not cancel the existing request.

The following values of the `concurrency` property are permitted:

- `multiple` Existing requests are not canceled and the developer is responsible for ensuring the consistency of returned data by carefully managing the event stream. The default value is `multiple`.
- `single` Making only one request at a time is allowed on the method; multiple requests generate a fault.
- `last` Making a request cancels any existing request.

***Note:** The request referred to here is not the HTTP request. It is the client action request. HTTP requests are sent to the server and get processed on the server side. However, the result is ignored when a request is canceled (requests are canceled when you use the `single` or `last` value). The `last` request is not necessarily the last one that the server receives over HTTP.*

Handling events

Calls to a remote service are asynchronous. After you invoke an asynchronous call, your application does not wait for the result, but continues to execute. Therefore, you typically use events to signal that the service call has completed.

When a service call completes, the RemoteObject component dispatches one of the following events:

- A `result` event indicates that the result is available. A `result` event generates a mx.rpc.events.ResultEvent object. You can use the `result` property of the ResultEvent object to access the data returned by the service call.
- A `fault` event indicates that an error occurred. A `fault` event generates a mx.rpc.events.FaultEvent object. You can use the `fault` property of the FaultEvent object to access information about the failure.

For more information on handling these events, see “Handling service events” on page 86.

Passing parameters

Flex provides two ways to pass parameters to a service call: *explicit parameter passing* and *parameter binding*. With explicit parameter passing, pass properties in the method that calls the service. With parameter binding, use data binding to populate the parameters from user interface controls or data models.

Explicit parameter passing with the RemoteObject component

The following example shows MXML code for declaring a RemoteObject component and calling a service using explicit parameter passing in the `click` event listener of a Button control. A ComboBox control provides data to the service.

```
<?xml version="1.0"?>
<!-- ds\rpc\RPCParamPassing.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            [Bindable]
```

```
            public var empList:Object;
        ]]>
    </mx:Script>

    <mx:RemoteObject
        id="employeeRO"
        destination="SalaryManager"
        result="empList=event.result;"
        fault="Alert.show(event.fault.faultString, 'Error');"/>

    <mx:ComboBox id="dept" width="150">
        <mx:dataProvider>
            <mx:ArrayCollection>
                <mx:source>
                    <mx:Object label="Engineering" data="ENG"/>
                    <mx:Object label="Product Management" data="PM"/>
                    <mx:Object label="Marketing" data="MKT"/>
                </mx:source>
            </mx:ArrayCollection>
        </mx:dataProvider>
    </mx:ComboBox>

    <mx:Button label="Get Employee List"
        click="employeeRO.getList(dept.selectedItem.data);"/>
</mx:Application>
```

Using parameter binding to pass parameters to the RemoteObject component

Parameter binding lets you copy data from user interface controls or models to request parameters. When you use parameter binding with RemoteObject components, you always declare operations in a RemoteObject component's `<mx:method>` tag. You then declare `<mx:arguments>` tags under an `<mx:method>` tag.

The order of the `<mx:arguments>` tags must match the order of the method parameters of the service. You can name argument tags to match the actual names of the corresponding method parameters as closely as possible, but it is not necessary.

Note: *Defining multiple argument tags with the same name in an `<mx:arguments>` tag creates the argument as an Array with the specified name. The service call fails if the remote method is not expecting an Array as the only input parameter. No warning about this situation occurs when the application is compiled.*

The following example uses parameter binding in a RemoteObject component's `<mx:method>` tag to bind the data of a selected ComboBox item to the `employeeRO.getList` operation when the user clicks a Button control. When you use parameter binding, you call a service by using the `send()` method with no parameters.

```
<?xml version="1.0"?>
<!-- ds\rpc\ROParamBind2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.utils.ArrayUtil;
        ]]>
    </mx:Script>

    <mx:ArrayCollection id="employeeAC"
        source="{ArrayUtil.toArray(employeeRO.getList.lastResult)}"/>

    <mx:RemoteObject
        id="employeeRO"
```

```
        destination="roDest"
        showBusyCursor="true"
        fault="Alert.show(event.fault.faultString, 'Error');">
        <mx:method name="getList">
            <mx:arguments>
                <deptId>{dept.selectedItem.data}</deptId>
            </mx:arguments>
        </mx:method>
    </mx:RemoteObject>

    <mx:HBox>
        <mx:Label text="Select a department:"/>
        <mx:ComboBox id="dept" width="150">
            <mx:dataProvider>
                <mx:ArrayCollection>
                    <mx:source>
                        <mx:Object label="Engineering" data="ENG"/>
                        <mx:Object label="Product Management" data="PM"/>
                        <mx:Object label="Marketing" data="MKT"/>
                    </mx:source>
                </mx:ArrayCollection>
            </mx:dataProvider>
        </mx:ComboBox>
        <mx:Button label="Get Employee List" click="employeeRO.getList.send();"/>
    </mx:HBox>

    <mx:DataGrid dataProvider="{employeeAC}" width="100%">
        <mx:columns>
            <mx:DataGridColumn dataField="name" headerText="Name"/>
            <mx:DataGridColumn dataField="phone" headerText="Phone"/>
            <mx:DataGridColumn dataField="email" headerText="Email"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

If you are unsure whether the result of a service call contains an Array or an individual object, use the `toArray()` method of the mx.utils.ArrayUtil class to convert it to an Array, as this example shows. If you pass the `toArray()` method an individual object, it returns an Array with that object as the only Array element. If you pass an Array to the method, it returns the same Array. For information about working with ArrayCollection objects, see the Flex Help Resource Center.

Handling results

Most service calls return data to the application. The RemoteObject component creates an Operation object for each operation supported by the associated Java class. Access the data by using the `Operation.lastResult` property for the specific operation, or in a `result` event by using the `ResultEvent.result` property.

By default, the `resultFormat` property value of the Operation class is `object`, and the data that is returned is represented as a simple tree of ActionScript objects. Flex interprets XML data to appropriately represent base types, such as String, Number, Boolean, and Date. To work with strongly typed objects, populate those objects using the object tree that Flex creates.

The RemoteObject component returns anonymous Objects and Arrays that are complex types. If `makeObjectsBindable` is `true`, which it is by default, Objects are wrapped in mx.utils.ObjectProxy instances and Arrays are wrapped in mx.collections.ArrayCollection instances.

You handle results for the RemoteObject component in much the same way as you do for the WebService component. For more information, see “Handling service results” on page 94.

Accessing EJBs and other objects in JNDI

Access Enterprise JavaBeans (EJBs) and other objects stored in the Java Naming and Directory Interface (JNDI) by calling methods on a destination that is a service facade class that looks up an object in JNDI and calls its methods.

You can use stateless or stateful objects to call the methods of Enterprise JavaBeans and other objects that use JNDI. For an EJB, you can call a service facade class that returns the EJB object from JNDI and calls a method on the EJB.

In your Java class, you use the standard Java coding pattern, in which you create an initial context and perform a JNDI lookup. For an EJB, you also use the standard coding pattern in which your class contains methods that call the EJB home object's `create()` method and the resulting business methods of the EJB.

The following example uses a method called `getHelloData()` on a facade class destination:

```
<mx:RemoteObject id="Hello" destination="roDest">
    <mx:method name="getHelloData"/>
</mx:RemoteObject>
```

On the Java side, the `getHelloData()` method could easily encapsulate everything necessary to call a business method on an EJB. The Java method in the following example performs the following actions:

- Creates new initial context for calling the EJB
- Performs a JNDI lookup that gets an EJB home object
- Calls the EJB home object's `create()` method
- Calls the `sayHello()` method of the EJB

```
public void getHelloData() {
    try
    {
        InitialContext ctx = new InitialContext();
        Object obj = ctx.lookup("/Hello");
        HelloHome ejbHome = (HelloHome)
        PortableRemoteObject.narrow(obj, HelloHome.class);
        HelloObject ejbObject = ejbHome.create();
        String message = ejbObject.sayHello();
    }
    catch (Exception e);
}
```

Part 4: Messaging Service

Using the Messaging Service 122
Connecting to the Java Message Service (JMS) 141

Chapter 9: Using the Messaging Service

The Messaging Service expands the core messaging framework to add support for publish-subscribe messaging among multiple Flex clients through the BlazeDS server. A Flex application uses the client-side messaging API to send messages to, and receive messages from, a destination defined by the server. Messages are sent over a channel and processed by a server endpoint, both of which are protocol-specific.

The Messaging Service also supports bridging to JMS topics and queues on an embedded or external JMS server by using the JMSAdapter. For more information, see "Connecting to the Java Message Service (JMS)" on page 141.

Topics

Using the Messaging Service 122
Working with Producer components. 125
Working with Consumer components. 128
Using a pair of Producer and Consumer components in an application 131
Message filtering 132
Configuring the Messaging Service 136

Using the Messaging Service

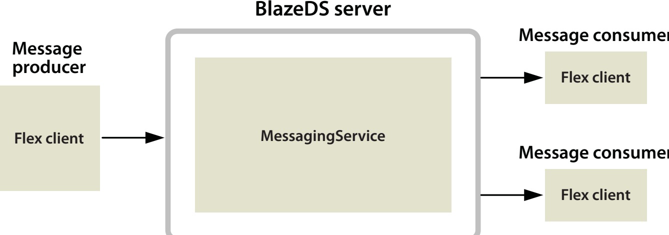
You use the Flex client-side API and the corresponding server-side Messaging Service to create messaging applications. Messaging lets a Flex application connect to a message destination on the server, send messages to the server, and receive messages from other messaging clients. Messages sent to the server are routed to other Flex applications that have subscribed to the same destination.

The server can also push messages to clients on its own. In the server push scenario, the server initiates the message and broadcasts it to a destination. All Flex applications that have subscribed to the destination receive the message. For an example, see the Trader Desktop sample application that ships with BlazeDS. For more information on the running the sample applications, see "Running the BlazeDS sample applications" on page 14.

The Messaging Service lets separate applications communicate asynchronously as peers by passing messages back and forth through the server. A message defines properties such as a unique identifier, BlazeDS headers, any custom headers, and a message body. The names of BlazeDS headers are prefixed by the string "DS".

The most well-known example of the type of application that can use the Messaging Service is an instant messaging application. In that application, one client sends a message to the server, and the server then routes the message to any subscribed clients. You can create other types of applications to implement broadcast messaging to simultaneously send messages to multiple clients, set up a system to send alert messages, or implement other types of messaging applications.

The following image shows the flow of messages from one Flex client to another. On the Flex client that sends the message, the message is routed over a channel to a destination on the server. The server then routes the message to other Flex clients, which perform any necessary processing of the received message.



Client applications that send messages are called message *producers*. You define a producer in a Flex application by using the Producer component. Client applications that receive messages are called message *consumers*. You define a consumer in a Flex application by using the Consumer component.

Producers send messages to specific destinations on the server. A Consumer component subscribes to a server-side destination, and the server routes any messages sent to that destination to the consumer. In most messaging systems, producers and consumers do not know anything about each other.

A Flex application using the Messaging Service often contains at least one pair of Producer and Consumer components. This configuration enables each application to send messages to a destination and receive messages that other applications send to that destination.

Types of messaging

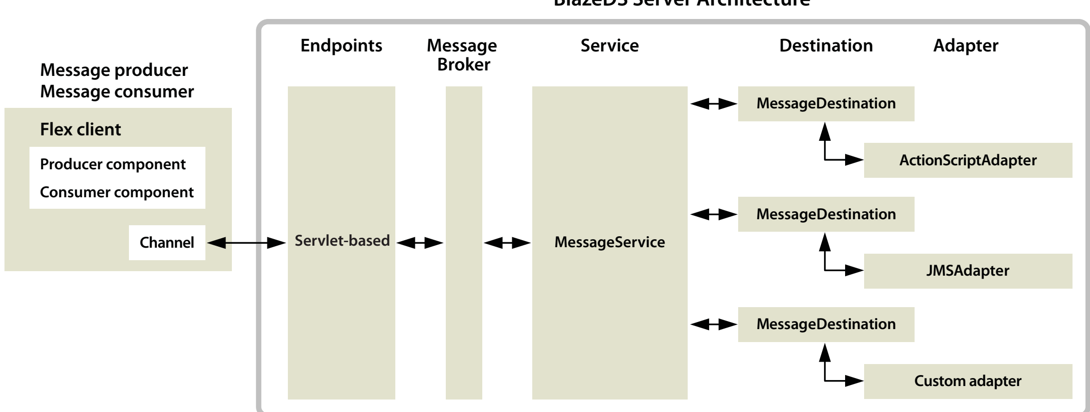
The Messaging Service supports publish-subscribe messaging. In publish-subscribe messaging, each message can have multiple consumers. You use this type of messaging when you want more than one consumer to receive the same message. Examples of applications that can use publish-subscribe messaging are auction sites, stock quote services, and other applications that require one message to be sent to many subscribers.

The Messaging Service lets you target messages to specific consumers that are subscribed to the same destination. Consumers can apply a selector expression to the message headers, or Producers and Consumers can add subtopic information to messages to target them. Therefore, even if multiple consumers are subscribed to the same destination, you can target a message to a single consumer or to a subset of all consumers. For more information, see “Message filtering” on page 132.

***Note:** You can support point-to-point messaging, also known as queue-based messaging, between Flex clients by using the JMSAdapter and bridging to a JMS queue. For more information, see “Connecting to the Java Message Service (JMS)” on page 141.*

The Messaging Service architecture

The components of the Messaging Service include channels, destinations, adapters, producers, and consumers. The following image shows the messaging architecture:



Channels

Flex applications can access the Message Service over several different message channels. The Flex client tries the channels in the order specified in the configuration files, until an available channel is found or all channels have been tried.

Each channel corresponds to a specific network protocol and has a corresponding server-side endpoint. Use a real-time channel with messaging. Real-time channels include AMFChannel and HTTPChannel with polling enabled, and StreamingAMFChannel and StreamingHTTPChannel. The AMF and HTTP channels with polling enabled poll the server for new messages when one or more Consumer components on the client have an active subscription.

Message Service

The Message Service maintains a list of message destinations and the clients subscribed to each destination. You configure the Message Service to transport messages over one or more channels, where each channel corresponds to a specific transport protocol.

Destinations

A destination is the server-side code that you connect to using Producer and Consumer components. When you define a destination, you reference one or more message channels that transport messages. You also reference a message adapter or use an adapter that is configured as the default adapter.

Adapters

BlazeDS provides two adapters to use with the Messaging Service and lets you create your own custom adapter:

- The ActionScriptAdapter is the server-side code that facilitates messaging when your application uses ActionScript objects only or interacts with another system. The ActionScriptAdapter lets you use messaging with Flex clients as the sole producers and consumers of the messages.
- The JMSAdapter lets you bridge destinations to JMS destinations, topics, or queues on a JMS server so that Flex clients can send messages to and receive messages from the JMS server.
- A custom adapter lets you create an adapter to interact with other messaging implementations, or for situations where you need functionality not provided by either of the standard adapters.

You reference adapters and specify adapter-specific settings in a destination definition of the configuration files.

Messaging Service configuration

You configure the Messaging Service by editing the messaging-config.xml file or the services-config.xml file. As a best practice, use the messaging-config.xml file for your configuration to keep it separate from other types of configurations.

Within the services-config.xml file, you specify the channels to use with a destination, set properties of the destination, enable logging, configure security, and specify other properties. For more information, see "Configuring the Messaging Service" on page 136.

Working with Producer components

You use the Producer component in a Flex application to enable the application to send messages. You can create Producer components in MXML or ActionScript.

To send a message from a Producer component to a destination, you create an mx.messaging.messages.AsyncMessage object, populate the body of the AsyncMessage object, and then call the `Producer.send()` method. You can create text messages and messages that contain objects.

You can optionally specify `acknowledge` and `fault` event handlers for a Producer component. An `acknowledge` event is dispatched when the Producer receives an acknowledge message to indicate that the destination successfully received a message that the Producer sent. A `fault` event is dispatched when a destination cannot successfully process a message due to a connection-, server-, or application-level failure.

For reference information about the Producer class, see *Adobe LiveCycle ES ActionScript Reference*.

Creating a Producer component in MXML

You use the `<mx:Producer>` tag to create a Producer component in MXML. The tag must contain an `id` value. The component typically specifies a `destination` that is defined in the messaging-config.xml file or the services-config.xml file. The following code shows an `<mx:Producer>` tag that specifies a destination and `acknowledge` and `fault` event handlers:

```
<?xml version="1.0"?>
<!-- ds\messaging\CreateProducerMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.rpc.events.FaultEvent;
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            private function acknowledgeHandler(event:MessageAckEvent):void {
                // Handle acknowledge message event.
            }

            private function faultHandler(event:MessageFaultEvent):void {
                // Handle message fault event.
            }

            // Compose the message as an instance of AsyncMessage,
            // then use the Producer.send() method to send it.
            private function sendMessage():void {
                var message:AsyncMessage = new AsyncMessage();
                message.body = userName.text + ": " + input.text;
```

```
                producer.send(message);
            }
        ]]>
    </mx:Script>

    <mx:Producer id="producer"
        destination="chat"
        acknowledge="acknowledgeHandler(event);"
        fault="faultHandler(event);"/>

    <mx:TextInput id="userName"/>
    <mx:TextInput id="input"/>
    <mx:Button label="Send"
        click="sendMessage();"/>
</mx:Application>
```

In this example, the Producer component sets the `destination` property to chat, which is a destination defined in the messaging-config.xml file. You can also specify the destination at run time by setting the `destination` property in your ActionScript code.

Creating a Producer component in ActionScript

You can create a Producer component in ActionScript. The following code shows a Producer component that is created in a method in an `<mx:Script>` tag:

```
<?xml version="1.0"?>
<!-- ds\messaging\CreateProducerAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="createProducer();">

    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            //Define a variable of type Producer.
            private var producer:Producer;

            // Create the Producer.
            private function createProducer():void {
               producer = new Producer();
               producer.destination = "chat";
              producer.addEventListener(MessageAckEvent.ACKNOWLEDGE,  acknowledgeHandler);
               producer.addEventListener(MessageFaultEvent.FAULT, faultHandler);
            }

            private function acknowledgeHandler(event:MessageAckEvent):void{
                // Handle message acknowledge event.
            }

            private function faultHandler(event:MessageFaultEvent):void{
                // Handle message fault event.
            }

            // Compose the message as an instance of AsyncMessage,
            // then use the Producer.send() method to send it.
            private function sendMessage():void {
                var message:AsyncMessage = new AsyncMessage();
                message.body = userName.text + ": " + input.text;
                producer.send(message);
```

```
            }
        ]]>
    </mx:Script>

    <mx:TextInput id="userName"/>
    <mx:TextInput id="input"/>
    <mx:Button label="Send"
        click="sendMessage();"/>
</mx:Application>
```

Resending messages and timing-out requests

A Producer component sends a message once. If the delivery of a message is in doubt, the Producer component dispatches a `fault` event to indicate that it never received an acknowledgment from the destination. When the event is dispatched, the event handler can then make a decision to attempt to resend the message if it is safe to do so.

***Note:** If a message has side effects, be careful about automatically trying to resend it. If it doesn't have side effects, then you can resend it safely. For example, an HTTP GET operation is read only so it can be safely resent. However, HTTP POST, PUT, and DELETE operations have side effects on the server that make them hard to resend.*

Two situations can trigger a fault that indicates delivery is in doubt. It can be triggered when the value of the `Producer.requestTimeout` property is exceeded, or the underlying message channel becomes disconnected before the acknowledgment message is received. The `fault` handler code can detect this scenario by inspecting the `ErrorMessage.faultCode` property of the associated event object for the `ErrorMessage.MESSAGE_DELIVERY_IN_DOUBT` value.

The `Producer.connected` property is set to `true` when the Flex client is connected to the server. The `Producer.send()` method automatically checks this property before attempting to send a message. Two properties control the action of the Producer component when it becomes disconnected from the server:

- `resubscribeAttempts`

  Specifies the number of times the component attempts to reconnect to the server before dispatching a `fault` event. The component makes the specified number of attempts over each available channel. A value of -1 specifies to continue indefinitely and a value of zero disables attempts.

- `resubscribeInterval`

  Specifies the interval, in milliseconds, between attempts to reconnect. Setting the value to 0 disables reconnection attempts.

Working with Consumer components

You can create Consumer components in MXML or ActionScript. To subscribe to a destination, you call the `Consumer.subscribe()` method.

You can also specify `message` and `fault` event handlers for a Consumer component. A Consumer component broadcasts a `message` event when a message is sent to a destination and the message has been routed to a consumer subscribed to that destination. A `fault` event is broadcast when the channel to which the Consumer component is subscribed cannot establish a connection to the destination, or the subscription request is denied.

For reference information about the Consumer class, see the *Adobe LiveCycle ES ActionScript Reference*.

Creating a Consumer component in MXML

You use the `<mx:Consumer>` tag to create a Consumer component in MXML. The tag must contain an `id` value. It typically specifies a `destination` that is defined in the server-side services-config.xml file.

The following code shows an `<mx:Consumer>` tag that specifies a destination and `acknowledge` and `fault` event handlers:

```
<?xml version="1.0"?>
<!-- ds\messaging\CreateConsumerMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="logon();">

    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            // Subscribe to destination.
            private function logon():void {
                consumer.subscribe();
            }

            // Write received message to TextArea control.
            private function messageHandler(event:MessageEvent):void {
                // Handle message event.
                ta.text += event.message.body + "\n";
            }

            private function faultHandler(event:MessageFaultEvent):void {
                // Handle message fault event.
            }
        ]]>
    </mx:Script>

    <mx:Consumer id="consumer"
        destination="chat"
        message="messageHandler(event);"
        fault="faultHandler(event);"/>
    <mx:TextArea id="ta" width="100%" height="100%"/>
</mx:Application>
```

You can unsubscribe a Consumer component from a destination by calling the component's `unsubscribe()` method.

Creating a Consumer component in ActionScript

You can create a Consumer component in ActionScript. The following code shows a Consumer component created in a method in an `<mx:Script>` tag:

```
<?xml version="1.0"?>
<!-- ds\messaging\CreateConsumerAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="logon();">

    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            // Create a variable of type Consumer.
            private var consumer:Consumer;

            // Create the Consumer.
            private function logon():void {
                consumer = new Consumer();
                consumer.destination = "chat";
                consumer.addEventListener
                    (MessageEvent.MESSAGE, messageHandler);
                consumer.addEventListener
                    (MessageFaultEvent.FAULT, faultHandler);
                consumer.subscribe();
            }

            // Write received message to TextArea control.
            private function messageHandler(event:MessageEvent):void {
                // Handle message event.
                ta.text += event.message.body + "\n";
            }

            private function faultHandler(event:MessageFaultEvent):void{
                // Handle message fault event.
            }
        ]]>
    </mx:Script>

    <mx:TextArea id="ta" width="100%" height="100%"/>
</mx:Application>
```

Sending and receiving an object in a message

You can send and receive objects as part of a message. The following example sends and receives a message that contains an object:

```
<?xml version="1.0"?>
<!-- ds\messaging\SendObjectMessage.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="logon();">

    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;
```

```
            // Subscribe to destination.
            private function logon():void {
                consumer.subscribe();
            }

            // Create message from TextInput controls.
            private function sendMessage():void {
                var message:AsyncMessage = new AsyncMessage();
                message.body = new Object();
                message.body.uName = userName.text;
                message.body.uInput = input.text;
                message.body.theCollection = ['b','a',3,new Date()];
                producer.send(message);
            }

            // Write received message to TextArea control.
            private function messageHandler(event:MessageEvent):void {
                // Handle message event.
                ta.text = String(event.message.body.uName) + " ," +
                String(event.message.body.uInput);
            }
        ]]>
    </mx:Script>

    <mx:Producer id="producer"
        destination="chat"/>
    <mx:Consumer id="consumer"
        destination="chat"
        message="messageHandler(event);"/>

    <!-- User input controls. -->
    <mx:TextInput id="userName"/>
    <mx:TextInput id="input"/>
    <mx:Button label="Send"
    click="sendMessage();"/>

    <!-- Display received message. -->
    <mx:TextArea id="ta"/>
</mx:Application>
```

Handling a network disconnection

Use the `Consumer.subscribed` and `Consumer.connected` properties to monitor the status of the Consumer component. The `connected` property is set to `true` when the Flex client is connected to the server. The `subscribed` property is set to `true` when the Flex client is subscribed to a destination.

Two properties control the action of the Consumer component when the destination becomes unavailable, or the subscription to the destination fails:

- `resubscribeAttempts`

  Specifies the number of times the component attempts to resubscribe to the server before dispatching a `fault` event. The component makes the specified number of attempts over each available channel. You can set the `Channel.failoverURIs` property to the URI of a computer to attempt to resubscribe to if the connection is lost. You typically use this property when operating in a clustered environment. For more information, see "Clustering" on page 167.

  A value of -1 specifies to continue indefinitely, and a value of 0 disables attempts.

- `resubscribeInterval`

  Specifies the interval, in milliseconds, between attempts to resubscribe. Setting the value to 0 disables resubscription attempts.

Calling the receive method

Typically, you use a real-time polling or streaming channel with the Consumer component. In both cases, the Consumer receives messages from the server without having to initiate a request. For more information, see "Channels" on page 124.

You can use a non-real-time channel, such as an AMFChannel with `polling-enabled` set to `false`, with a Consumer component. In that case, call the `Consumer.receive()` method directly to initiate a request to the server to receive any queued messages. Before you call the `receive()` method, call the `subscribe()` method to subscribe to the destination. A `fault` event is broadcast if a failure occurs when the `Consumer.receive()` method is called.

Using a pair of Producer and Consumer components in an application

A Flex application often contains at least one pair of Producer and Consumer components. This configuration enables each application to send messages to a destination and receive messages that other applications send to that destination.

To act as a pair, Producer and Consumer components in an application must use the same message destination. Producer component instances send messages to a destination and Consumer component instances receive messages from that destination.

The following code shows a simple chat application that contains a pair of Producer and Consumer components. The user types messages in a TextInput control; the Producer component sends the message when the user presses the keyboard Enter key or clicks the Button control labeled Send. The user views messages from other users in the `ta` TextArea control. If you open this application in two browser windows, any message sent by one instance of the application appears in both.

```
<?xml version="1.0"?>
<!-- ds\messaging\ProducerConsumer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="logon();">

    <mx:Script>
        <![CDATA[

            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            // Subscribe to destination.
            private function logon():void {
               consumer.subscribe();
            }

            // Write received message to TextArea control.
            private function messageHandler(event: MessageEvent):void {
               ta.text += event.message.body + "\n";
            }

            // Compose the message as an instance of AsyncMessage,
```

```
            // then use the Producer.send() method to send it.
            private function sendMessage():void {
               var message: AsyncMessage = new AsyncMessage();
               message.body = userName.text + ": " + msg.text;
               producer.send(message);
               msg.text = "";
            }
        ]]>
    </mx:Script>

    <mx:Producer id="producer" destination="chat"/>
    <mx:Consumer id="consumer" destination="chat"
        message="messageHandler(event)"/>

    <mx:TextArea id="ta" width="100%" height="100%"/>

    <mx:TextInput id="userName" width="100%"/>
    <mx:TextInput id="msg" width="100%"/>
    <mx:Button label="Send"
    click="sendMessage();"/>
</mx:Application>
```

Message filtering

The Messaging Service provides functionality for Producer components to add information to message headers and to add subtopic information. Consumer components can then specify filtering criteria based on this information so that only message that meet the filtering criteria are received by the consumer.

The Consumer component sends the filtering criteria to the server when the Consumer calls the `subscribe()` method. Therefore, while the Consumer component defines the filtering criteria, the actual filtering is done on the server before a message is sent to the consumer.

Note: *Filter messages based on message headers or subtopics. However, do not filter messages using both techniques at the same time.*

Using selectors

A Producer component can include extra information in a message in the form of message headers. A Consumer component then uses the `selector` property to filter messages based on message header values.

Use the `AsyncMessage.headers` property of the message to specify the message headers. The headers are contained in an associative Array where the key is the header name and the value is either a String or a number.

Note: *Do not start message header names with the text "JMS" or "DS". These prefixes are reserved.*

The following code adds a message header called prop1 and sets its value:

```
<?xml version="1.0"?>
<!-- ds\messaging\SendMessageHeader.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            private function sendMessage():void {
```

```
                var message:AsyncMessage = new AsyncMessage();
                message.headers = new Array();
                message.headers["prop1"] = 5;
                message.body = input.text;
                producer.send(message);
            }
        ]]>
    </mx:Script>

    <mx:Producer id="producer"
        destination="chat"/>

    <mx:TextInput id="userName"/>
    <mx:TextInput id="input"/>
    <mx:Button label="Send"
        click="sendMessage();"/>
</mx:Application>
```

To filter messages based on message headers, use the `Consumer.selector` property to specify a message selector. A message selector is a String that contains a SQL conditional expression based on the SQL92 conditional expression syntax. The Consumer component receives only messages with headers that match the selector criteria.

The following code sets the `Consumer.selector` property so that the Consumer only receives messages where the value of prop1 in the message header is greater than 4:

```
<?xml version="1.0"?>
<!-- ds\messaging\CreateConsumerMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="logon();">

    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            // Subscribe to destination.
            private function logon():void {
                consumer.subscribe();
            }

            // Write received message to TextArea control.
            private function messageHandler(event:MessageEvent):void {
                // Handle message event.
                ta.text += event.message.body + "\n";
            }

        ]]>
    </mx:Script>

    <mx:Consumer id="consumer"
        destination="chat"
        selector="prop1 > 4"
        message="messageHandler(event);"/>

    <mx:TextArea id="ta" width="100%" height="100%"/>
</mx:Application>
```

***Note:** For advanced messaging scenarios, you can use the mx.messaging.MultiTopicConsumer and mx.messaging.MultiTopicProducer classes.*

Using subtopics

A Producer can send a message to a specific category or categories, called subtopics, within a destination. You then configure a Consumer component to receive only messages assigned to a specific subtopic or subtopics.

***Note:** You cannot use subtopics with a JMS destination. However, you can use message headers and Consumer selector expressions to achieve similar functionality when using JMS. For more information, see* *"Using selectors" on page 132**.*

In a Producer component, use the `subtopic` property to assign a subtopic to messages. Define a subtopic as a dot (.) delimited String, in the form:

```
mainToken[.secondaryToken][.additionalToken][...]
```

For example, you can define a subtopic in the form "chat", "chat.fds", or "chat.fds.newton". The dot (.) delimiter is the default; use the `<subtopic-separator>` property in the configuration file to set a different delimiter. For more information, see "Setting server properties in the destination" on page 138.

In the Consumer component, use the `subtopic` property to define the subtopic that a message must be sent to for it to be received. You can specify a literal String value for the `subtopic` property. Use the wildcard character (*) in the `Consumer.subtopic` property to receive messages from more than one subtopic.

The Messaging Service supports single-token wildcards in the subtopic String. If the wildcard character is the last character in the String, it matches any tokens in that position or in any subsequent position. For example, the Consumer component specifies the subtopic as "foo.*". It matches the subtopics "foo.bar" and "foo.baz", and also "foo.bar.aaa" and "foo.bar.bbb.ccc".

If the wildcard character is in any position other than the last position, it only matches a token at that position. For example, a wildcard character in the second position matches any tokens in the second position of a subtopic value, but it does not apply to multiple tokens. Therefore, if the Consumer component specifies the subtopic as "foo.*.baz", it matches the subtopics "foo.bar.baz" and "foo.aaa.baz", but not "foo.bar.cookie".

To send a message from a Producer component to a destination and a subtopic, set the `destination` and `subtopic` properties, and then call the `send()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- ds\messaging\Subtopic1.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            private function useSubtopic():void {
                var message:AsyncMessage = new AsyncMessage();
                producer.subtopic = "chat.fds.newton";
                // Generate message.
                producer.send(message);
            }
        ]]>
    </mx:Script>

    <mx:Producer id="producer"
        destination="chat"/>
</mx:Application>
```

To subscribe to a destination and a subtopic with a Consumer component, set the `destination` and `subtopic` properties and then call the `subscribe()` method, as the following example shows. This example uses a wildcard character (*) to receive all messages sent to all subtopics under the chat.fds subtopic.

```
<?xml version="1.0"?>
```

```
<!-- ds\messaging\Subtopic2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="logon();">

    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            private function messageHandler(event:MessageEvent):void {
                // Handle message event.
                ta.text += event.message.body + "\n";
            }

            private function logon():void {
                consumer.subtopic = "chat.fds.*";
                consumer.subscribe();
            }
        ]]>
    </mx:Script>

    <mx:Consumer id="consumer"
    destination="chat"
        message="messageHandler(event);"/>
    <mx:TextArea id="ta" width="100%" height="100%"/>
</mx:Application>
```

To allow subtopics for a destination, set the `allow-subtopics` element to `true` in the destination definition in the messaging-config.xml file. The `subtopic-separator` element is optional and lets you change the separator character; the default value is `"."` (period).

```
<destination id="ChatTopic">
    <properties>
        <network>
            <subscription-timeout-minutes>0</subscription-timeout-minutes>
        </network>
        <server>
            <message-time-to-live>0</message-time-to-live>
            <allow-subtopics>true</allow-subtopics>
            <subtopic-separator>.</subtopic-separator>
        </server>
    </properties>
    <channels>
        <channel ref="my-streaming-amf"/>
    </channels>
</destination>
```

For more information on configuration, see “Configuring the Messaging Service” on page 136.

Configuring the Messaging Service

The most common tasks that you perform when configuring the Messaging Service are defining message destinations and applying security to message destinations. Typically, you configure the Messaging Service in the messaging-config.xml file. The services-config.xml file includes the messaging-config.xml file by reference.

The following example shows a basic Message Service configuration in the messaging-config.xml file. It contains the following information:

- A service definition of the Message Service
- A reference to the MessageService class.
- A definition of the ActionScriptAdapter.
- A destination definition that references two channels. You define channels outside the destination definition.

```
<service id="message-service"
    class="flex.messaging.services.MessageService">
    <adapters>
        <adapter-definition
            id="actionscript"
            class="flex.messaging.services.messaging.adapters.ActionScriptAdapter"
            default="true"/>
        <adapter-definition id="jms"
            class="flex.messaging.services.messaging.adapters.JMSAdapter"/>
    </adapters>
    <destination id="chat-topic">
        <properties>
            <server>
                <message-time-to-live>0</message-time-to-live>
            </server>
        </properties>
        <channels>
            <channel ref="samples-rtmp"/>
            <channel ref="samples-amf-polling"/>
        </channels>
    </destination>
</service>
```

Configuring the adapter

To use a Message Service adapter, such as the ActionScript, JMS, or ColdFusion Event Gateway Adapter, you reference the adapter in a destination definition. If you do not explicitly specify an adapter, the destination uses the default adapter as defined in the `<adapters>` tag. In addition to referencing an adapter, you also set its properties in a destination definition.

The following example shows two adapter definitions: the ActionScriptAdapter and the JMSAdapter. In this example, the ActionScriptAdapter is defined as the default adapter:

```
<service id="message-service"
    class="flex.messaging.services.MessageService">
    ...
    <adapters>
        <adapter-definition
            id="actionscript"
            class="flex.messaging.services.messaging.
            adapters.ActionScriptAdapter" default="true"/>
        <adapter-definition
            id="jms"
            class="flex.messaging.services.messaging.adapters.JMSAdapter"/>
    </adapters>
```

```
    ...
    <destination id="chat-topic">
        ...
        <adapter ref="actionscript"/>
        ...
    </destination>
    ...
</service>
```

Since ActionScriptAdapter is defined as the default adapter, you can omit the `<adapter>` tag from the destination definition,

Defining the destination

You perform most of the configuration for the Messaging Service in the destination definition, including specifying the adapter and channels used by the destination, and any network and server properties.

Setting network properties in the destination

A destination contains a set of properties for defining client-server messaging behavior. The following example shows the network-related properties of a destination:

```
<destination id="chat-topic">
    <properties>
        <network>
            <throttle-inbound policy="ERROR" max-frequency="50"/>
            <throttle-outbound policy="ERROR" max-frequency="500"/>
        </network>
    </properties>
</destination>
```

Message Service destinations use the following network-related properties:

Property	Description
`subscription-timeout-minutes`	Subscriptions that receive no pushed messages in this time interval, in minutes, are automatically unsubscribed. When the value is set to 0 (zero), subscribers are not forced to unsubscribe automatically. The default value is 0.
`throttle-inbound`	The `max-frequency` attribute controls how many messages per second the message destination accepts. The `policy` attribute indicates what to do when the message limit is reached: • A `policy` value of `NONE` specifies no throttling policy (same as frequency of zero). • A `policy` value of `ERROR` specifies that when the frequency is exceeded, throttle the message and send an error to the client. • A `policy` value of `IGNORE` specifies that when the frequency is exceeded, throttle the message but don't send an error to the client.
`throttle-outbound`	The `max-frequency` attribute controls how many messages per second the server can route to subscribed consumers. The `policy` attribute indicates what to do when the message limit is reached: • A `policy` value of `NONE` specifies no throttling policy (same as frequency of zero). • A `policy` value of `ERROR` specifies that when the frequency is exceeded, throttle the message and send an error to the client. • A `policy` value of `IGNORE` specifies that when the frequency is exceeded, throttle the message but don't send an error to the client.

Setting server properties in the destination

A destination contains a set of properties for controlling server-related parameters. The following example shows server-related properties of a destination:

```
<destination id="chat-topic">
    <properties>
    ...
        <server>
            <message-time-to-live>0</message-time-to-live>
        </server>
    </properties>
</destination>
```

Message Service destinations use the following server-related properties:

Property	Description
`allow-subtopics`	(Optional) The subtopic feature lets you divide the messages that a Producer component sends to a destination into specific categories in the destination. You can configure a Consumer component that subscribes to the destination to receive only messages sent to a specific subtopic or set of subtopics. You use wildcard characters (*) to subscribe for messages from more than one subtopic.
`cluster-message-routing`	(Optional) Determines whether a destination in an environment that uses software clustering uses `server-to-server` (default) or `broadcast` messaging. With `server-to-server` mode, data messages are routed only to servers with active subscriptions, but subscribe and unsubscribe messages are broadcast across the cluster. With broadcast messaging, all messages are broadcast across the cluster. For more information, see "Clustering" on page 167.
`message-time-to-live`	The number of milliseconds that a message is kept on the server pending delivery before being discarded as undeliverable. A value of 0 means the message is not expired.
`send-security-constraint`	(Optional) Security constraints apply to the operations performed by the messaging adapter. The `send-security-constraint` property applies to send operations.
`subscribe-security-constraint`	(Optional) Security constraints apply to the operations performed by the messaging adapter. The `subscribe-security-constraint` property applies to subscribe, multi-subscribe, and unsubscribe operations.
`subtopic-separator`	(Optional) Token that separates a hierarchical subtopic value. For example, for the subtopic 'foo.bar' the dot (.) is the subtopic separator. The default value is the dot (.) character.

Referencing message channels in the destination

The following example shows a destination referencing a channel. Because the `samples-rtmp` channel is listed first, it is used first and only if a connection cannot be established does the client attempt to connect over the rest of the channels in order of definition.

```
<destination id="chat-topic">
    ...
    <channels>
        <channel ref="samples-rtmp"/>
        <channel ref="samples-amf-polling"/>
    </channels>
    ...
</destination>
```

For more information about message channels, see "Channels and endpoints" on page 38.

Applying security to the destination

One way to secure a destination is by using a *security constraint*, which defines the access privileges for the destination. You use a security constraint to authenticate and authorize users before allowing them to access a destination. You can specify whether to use basic or custom authentication, and indicate the roles required for authorization.

Two security properties that you can set for a messaging destination include the following:

- `send-security-constraint`

  Specifies the security constraint for a Producer component sending a message to the server.

- `subscribe-security-constraint`

  Specifies the security constraint for a Consumer component subscribing to a destination on the server.

You use these properties in a destination definition, as the following example shows:

```
<destination id="chat">
    ...
    <properties>
        <server>
            <send-security-constraint ref="sample-users"/>
            <subscribe-security-constraint ref="sample-users"/>
        </server>
    </properties>
    ...
</destination>
```

In this example, the properties reference the sample-users security constraint defined in the services-config.xml file, which specifies to use custom authentication:

```
<security>
    <login-command class="flex.messaging.security.TomcatLoginCommand" server="Tomcat">
        <per-client-authentication>false</per-client-authentication>
    </login-command>
    <security-constraint id="basic-read-access">
        <auth-method>Basic</auth-method>
        <roles>
            <role>guests</role>
            <role>accountants</role>
        </roles>
    </security-constraint>
    <security-constraint id="sample-users">
        <auth-method>Custom</auth-method>
        <roles>
            <role>sampleusers</role>
        </roles>
    </security-constraint>
</security>
```

For more information about security, see “Securing BlazeDS” on page 156.

Creating a custom Message Service adapter

You can create a custom Message Service adapter for situations where you need functionality not provided by either of the standard adapters. A Message Service adapter class must extend the flex.messaging.services.MessageServiceAdapter class. An adapter calls methods on an instance of a flex.messaging.MessageService object. Both ServiceAdapter and MessageService are included in the public BlazeDS Javadoc documentation.

The primary method of any Message Service adapter class is the `invoke()` method, which is called when a client sends a message to a destination. In the `invoke()` method, you can include code to send messages to all subscribing clients or to specific clients by evaluating selector statements included with a message.

To send a message to clients, you call the `MessageService.pushMessageToClients()` method in your adapter's `invoke()` method. This method takes a message object as its first parameter. Its second parameter is a Boolean value that indicates whether to evaluate message selector statements. You can call the `MessageService.sendPushMessageFromPeer()` method in your adapter's `invoke()` method to broadcast messages to peer server nodes in a clustered environment.

```
package customclasspackage;
{

    import flex.messaging.services.MessageServiceAdapter ;
    import flex.messaging.services.MessageService;
    import flex.messaging.messages.Message;
    import flex.messaging.Destination;

    public class SimpleCustomAdapter extends MessageServiceAdapter {

        public Object invoke(Message message) {
            MessageService msgService = (MessageService)service;
            msgService.pushMessageToClients(message, true);
            msgService.sendPushMessageFromPeer(message, true);
            return null;
        }
    }
}
```

Optionally, a Message Service adapter can manage its own subscriptions by overriding the `ServiceAdapter.handlesSubscriptions()` method and return `true`. You also must override the `ServiceAdapter.manage()` method, which is passed CommandMessages for subscribe and unsubscribe operations.

The `ServiceAdapter.getAdapterState()` and `ServiceAdapter.setAdapterState()` methods are for adapters that maintain an in-memory state that must be replicated across a cluster. When an adapter starts up, it gets a copy of that state from another cluster node when another node is running.

To use an adapter class, specify it in an `adapter-definition` element in the messaging-config.xml file, as the following example shows:

```
<adapters>
...
    adapter-definition id="cfgateway" class="foo.bar.SampleMessageAdapter"/>
...
</adapters>
```

Optionally, you can implement MBean component management in an adapter. This implementation lets you expose properties to a JMX server that can be used as an administration console. For more information, see "Monitoring and managing services" on page 153.

Chapter 10: Connecting to the Java Message Service (JMS)

The Messaging Service supports bridging BlazeDS to Java Message Service (JMS) messaging destinations by using the JMSAdapter. The JMSAdapter lets Flex clients publish messages to and consume messages from a JMS server. For more information on the Messaging Service, see “Using the Messaging Service” on page 122.

Topics

About JMS 141
Configuring the Messaging Service to connect to a JMSAdapter 143

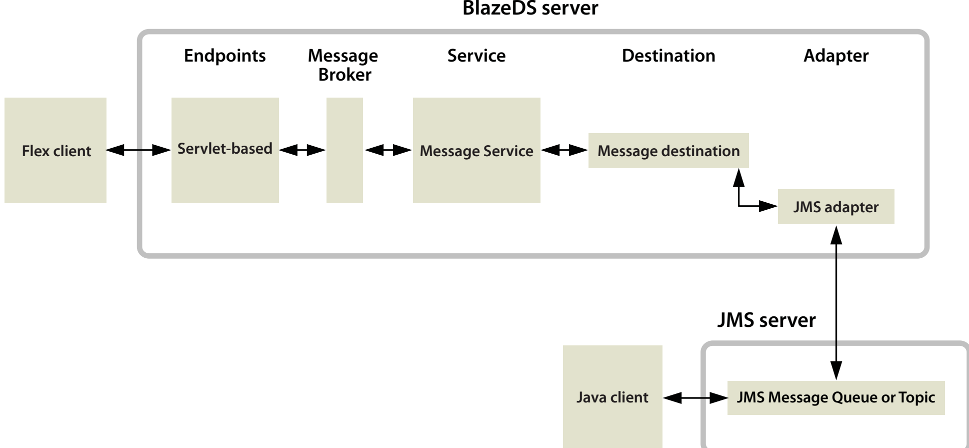
About JMS

Java Message Service (JMS) is a Java API that lets applications create, send, receive, and read messages. Flex applications can then exchange messages with Java client applications.

In a standard configuration of the Messaging Service, a destination references the ActionScriptAdapter. The ActionScriptAdapter lets you use messaging with Flex clients as the sole producers and consumers of the messages.

You use the JMSAdapter to connect BlazeDS to JMS topics or queues. The JMSAdapter supports topic-based and queue-based messaging. The JMSAdapter class lets Flex applications participate in existing messaging-oriented middleware (MOM) systems. Therefore, a Java application can publish messages to and respond to messages from Flex applications.

To connect BlazeDS to a JMS server, you create a destination that references the JMSAdapter. The following image shows BlazeDS using the JMSAdapter:



The Flex client sends and receives messages through a destination that references the JMSAdapter. The JMSAdapter then connects to the destination on the JMS server. Since the destination is accessible by a Java client, the Java client can exchange messages with the Flex client.

Writing client-side code to use JMS

The Flex client uses the Consumer and Producer components to send and receive messages through the JMSAdapter, just as it would for an application using the ActionScriptAdapter. For example, if the name of a destination that references the JMSAdapter is messaging_JMS_Topic, you reference it from a Producer component as the following example shows:

```
<mx:Producer id="producer"
    destination="messaging_JMS_Topic"
    acknowledge="acknowledgeHandler(event);"
    fault="faultHandler(event);"/>
```

JMS topics and queues

The JMSAdapter supports both JMS topics and JMS queues. The JMSAdapter supports the use of message headers and selectors for JMS topics, but hierarchical topics and subtopics are not supported.

Topics support dynamic client subscribe and unsubscribe, and therefore do not require the same level of administration as JMS queues. When using JMS queues, define a unique queue for each client.

If two Flex clients listen to the same JMS queue, and the JMS server sends a message to the queue, only one of the clients receives the message at a given time. This operation is expected because JMS queues are meant to be consumed by one consumer.

JMS queues are point-to-point, unlike topics which are one-to-many. However, due to the administrative overhead of JMS queues, the JMSAdapter is not the best choice for point-to-point messages between clients. A better choice for point-to-point messaging is to use the ActionScriptAdapter in conjunction with message filtering on the client side. For more information, see “Message filtering” on page 132.

Setting up your system to use the JMSAdapter

You can use any JMS server with BlazeDS that implements the JMS specification. To use the JMSAdapter to connect to a JMS server, you perform several types of configurations, including the following:

1 Configure your web application so that it has access to the JMS server. For example, if you are using Tomcat, you might have to add the following Resource definitions to the application to add support for Apache ActiveMQ, which supports JMS version 1.1:

```
<Context docBase="${catalina.home}/../../apps/team" privileged="true"
    antiResourceLocking="false" antiJARLocking="false" reloadable="true">
    <!-- Resourced needed for JMS -->
    <Resource name="jms/flex/TopicConnectionFactory"
        type="org.apache.activemq.ActiveMQConnectionFactory"
        description="JMS Connection Factory"
        factory="org.apache.activemq.jndi.JNDIReferenceFactory"
        brokerURL="vm://localhost"
        brokerName="LocalActiveMQBroker"/>
    <Resource name="jms/topic/flex/simpletopic"
        type="org.apache.activemq.command.ActiveMQTopic"
        description="my Topic"
        factory="org.apache.activemq.jndi.JNDIReferenceFactory"
        physicalName="FlexTopic"/>
    <Resource name="jms/flex/QueueConnectionFactory"
        type="org.apache.activemq.ActiveMQConnectionFactory"
        description="JMS Connection Factory"
        factory="org.apache.activemq.jndi.JNDIReferenceFactory"
        brokerURL="vm://localhost"
        brokerName="LocalActiveMQBroker"/>
    <Resource name="jms/queue/flex/simplequeue"
        type="org.apache.activemq.command.ActiveMQQueue"
```

```
        description="my Queue"
        factory="org.apache.activemq.jndi.JNDIReferenceFactory"
        physicalName="FlexQueue"/>
    <Valve className="flex.messaging.security.TomcatValve"/>
</Context>
```

The JMS server is often embedded in your J2EE server, but you can interact with a JMS server on a remote computer accessed by using JNDI. For more information, see "Using a remote JMS provider" on page 146.

2 Create a JMSAdapter definition in the messaging-config.xml file:

```
<adapters>
    <adapter-definition id="jms"
        class="flex.messaging.services.messaging.adapters.JMSAdapter"/>
</adapters>
```

For more information on configuring the JMSAdapter, see "Configure the JMSAdapter" on page 143.

3 Create a destination that references the adapter in the messaging-config.xml file:

```
<destination id="messaging_AMF_Poll_JMS_Topic" channels="my-amf-poll">
    <adapter ref="jms"/>
    <properties>
        <jms>
            <connection-factory>
                java:comp/env/jms/flex/TopicConnectionFactory
            </connection-factory>
            <destination-type>Topic</destination-type>
            <destination-jndi-name>
                java:comp/env/jms/topic/flex/simpletopic
            </destination-jndi-name>
            <message-type>javax.jms.TextMessage</message-type>
        </jms>
    </properties>
    <channels>
        <channel ref="samples-rtmp"/>
        <channel ref="samples-amf-polling"/>
    </channels>
</destination>
```

This destination references the first Apache ActiveMQ Resource, which supports topic-based messaging.

Note: *Since the channel only defines how the Flex client communicates with the server, you do not have to perform any special channel configuration to use the JMSAdapter.*

For more information, see "Configuring a destination to use the JMSAdapter" on page 144.

4 Compile your Flex application against the services-config.xml file, which includes the messaging-config.xml file by reference.

Configuring the Messaging Service to connect to a JMSAdapter

Typically, you configure the Messaging Service, including the JMSAdapter, in the messaging-config.xml file.

Configure the JMSAdapter

You configure the JMSAdapter individually for the destinations that use it, as the following example shows:

```
<adapters>
    <adapter-definition id="jms"
        class="flex.messaging.services.messaging.adapters.JMSAdapter"/>
</adapters>
```

Configuring a destination to use the JMSAdapter

You perform most of the configuration of the JMSAdapter in the destination definition. Configure the adapter with the proper JNDI information and JMS ConnectionFactory information to look up the connection factory in JNDI.

The following example shows a destination that uses the JMSAdapter:

```
<destination id="chat-topic-jms">
    <properties>
    ...
        <jms>
            <destination-type>Topic</destination-type>
            <message-type>javax.jms.TextMessage</message-type>
            <connection-factory>jms/flex/TopicConnectionFactory</connection-factory>
            <destination-jndi-name>jms/topic/flex/simpletopic</destination-jndi-name>
            <delivery-mode>NON_PERSISTENT</delivery-mode>
            <message-priority>DEFAULT_PRIORITY</message-priority>
            <preserve-jms-headers>"true"</preserve-jms-headers>
            <acknowledge-mode>AUTO_ACKNOWLEDGE</acknowledge-mode>
            <connection-credentials username="sampleuser" password="samplepassword"/>
            <max-producers>1</max-producers>
        </jms>
    </properties>
    ...
    <adapter ref="jms"/>
</destination>
```

The JMSAdapter accepts the following configuration properties. For more specific information about JMS, see the Java Message Service specification or your application server documentation.

Property	Description
acknowledge-mode	Not used with JMSAdapter.
connection-credentials	(Optional) The username and password used while creating the JMS connection for example: `<connection-credentials username="sampleuser" password="samplepassword"/>` Use only if JMS connection level authentication is being used.
connection-factory	Name of the JMS connection factory in JNDI.
delivery-mode	JMS DeliveryMode for producers. The valid values are PERSISTENT and NON_PERSISTENT. The PERSISTENT mode specifies that all sent messages be stored by the JMS server, and then forwarded to consumers. This configuration adds processing overhead but is necessary for guaranteed delivery. The NON_PERSISTENT mode does not require that messages be stored by the JMS server before forwarding to consumers, so they can be lost if the JMS server fails while processing the message. This setting is suitable for notification messages that do not require guaranteed delivery.

Property	Description
delivery-settings/mode	(Optional) Specifies the message delivery mode used to deliver messages from the JMS server. If you specify async mode, but the application server cannot listen for messages asynchronously (that is javax.jms.MessageConsumer.setMessageListener is restricted), or the application server cannot listen for connection problems asynchronously (for example, javax.jms.Connection.setExceptionListener is restricted), you get a configuration error asking the user to switch to sync mode. The default value is sync.
delivery-settings/sync-receive-interval-millis	(Optional) Default value is 100. The interval of the receive message calls. Only available when the mode value is sync.
delivery-settings/sync-receive-wait-millis	(Optional) Default value is 0 (no wait). Determines how long a JMS proxy waits for a message before returning. Using a high sync-receive-wait-millis value along with a small thread pool can cause messages to back up if many proxied consumers are not receiving a steady flow of messages. Only available when the mode value is sync.
destination-jndi-name	Name of the destination in the JNDI registry.
destination-type	(Optional) Type of messaging that the adapter is performing. Valid values are topic for publish-subscribe messaging and queue for point-to-point messaging. The default value is topic.
initial-context-environment	A set of JNDI properties for configuring the InitialContext used for JNDI lookups of your ConnectionFactory and Destination. Lets you use a remote JNDI server for JMS. For more information, see “Using a remote JMS provider” on page 146.
max-producers	The maximum number of producer proxies that a destination uses when communicating with the JMS server. The default value is 1, which indicates that all clients using the destination share the same connection to the JMS server.
message-priority	JMS priority for messages that producers send. The valid values are DEFAULT_PRIORITY or an integer value indicating the priority. The JMS API defines ten levels of priority value, with 0 as the lowest priority and 9 as the highest. Additionally, clients should consider priorities 0-4 as gradations of normal priority, and priorities 5-9 as gradations of expedited priority.
message-type	Type of message to use when transforming Flex messages into JMS messages. Supported types are javax.jms.TextMessage and javax.jms.ObjectMessage. If the client-side Publisher component sends messages as objects, set the message-type to javax.jms.ObjectMessage.
message-type	The javax.jms.Message type which the adapter uses for this destination. Supported types are javax.jms.TextMessage and javax.jms.ObjectMessage.
preserve-jms-headersBlazeDS	(Optional) Defaults to true. Determines whether the adapter preserves all standard JMS headers from JMS messages to BlazeDS messages. Every JMS message has a set of standard headers: JMSDestination, JMSDeliveryMode, JMSMessageID, JMSTimestamp, JMSExpiration, JMSRedelivered, JMSPriority, JMSReplyTo, JMSCorrelationID, and JMSType. The JMS server sets these headers when the message is created and they are passed to BlazeDS. BlazeDS converts the JMS message into a BlazeDS message and sets JMSMessageID and JMSTimestamp on the BlazeDS message as messageId and timestamp, but the rest of the JMS headers are ignored. Setting the preserve-jms-headers property to true preserves all of the headers.

Configuring a server for the JMSAdapter

When a destination specifies a <destination-type> of topic for the JMSAdapter, you can set the durable property in the server definition. When true, the durable property specifies that messages are saved in a durable message store to ensure that they survive connection outages and reach destination subscribers. The default value is false.

***Note:** This property does not guarantee durability between Flex clients and the JMSAdapter, but between the JMSAdapter and the JMS server.*

The following example sets the `durable` property to `true`:

```
<server>
    <durable>true</durable>
</server>
```

Using a remote JMS provider

In many cases, the JMS server is embedded in your J2EE server. However, you can also interact with a JMS server on a remote computer accessed by using JNDI.

You can use JMS on a remote JNDI server by configuring the optional `initial-context-environment` element in the `jms` section of a message destination that uses the JMSAdapter. The `initial-context-environment` element takes `property` subelements, which in turn take `name` and `value` subelements. To establish the desired JNDI environment, specify the javax.naming.Context constant names and corresponding values in the `name` and `value` elements, or specify String literal names and corresponding values.

The bold-faced code in the following example is an `initial-context-environment` configuration:

```
<destination id="chat-topic-jms">
    <properties>
        ...
        <jms>
            <destination-type>Topic</destination-type>
            <message-type>javax.jms.TextMessage</message-type>
            <connection-factory>jms/flex/TopicConnectionFactory</connection-factory>
            <destination-jndi-name>jms/topic/flex/simpletopic</destination-jndi-name>
            <delivery-mode>NON_PERSISTENT</delivery-mode>
            <message-priority>DEFAULT_PRIORITY</message-priority>
            <acknowledge-mode>AUTO_ACKNOWLEDGE</acknowledge-mode>

            <!-- (Optional) JNDI environment. Use when using JMS on a remote JNDI server. -->
            <initial-context-environment>
                <property>
                    <name>Context.SECURITY_PRINCIPAL</name>
                    <value>anonymous</value>
                </property>
                <property>
                    <name>Context.SECURITY_CREDENTIALS</name>
                    <value>anonymous</value>
                </property>
                <property>
                    <name>Context.PROVIDER_URL</name>
                    <value>http://{server.name}:1856</value>
                </property>
                <property>
                    <name>Context.INITIAL_CONTEXT_FACTORY</name>
                    <value>fiorano.jms.runtime.naming.FioranoInitialContextFactory</value>
                </property>
            </initial-context-environment>
        </jms>
    </properties>
    ...
    <adapter ref="jms"/>
</destination>
```

Flex treats name element values that begin with the text "Context." as constants defined by javax.naming.Context and verifies that the Context class defines the constants that you specify. Some JMS providers also allow custom properties to be set in the initial context. You can specify these properties by using the string literal name and corresponding value that the provider requires. For example, the FioranoMQ JMS provider configures failover to back up servers with the following property:

```
<property>
     <name>BackupConnectURLs</name>
     <value>http://backup-server:1856;http://backup-server-2:1856</value>
</property>
```

If you do not specify the initial-context-environment properties in the jms section of a destination definition, the default JNDI environment is used. The default JNDI environment is configured in a jndiprovider.properties application resource file and or a jndi.properties file.

Naming conventions across JNDI providers for topic connection factories and destinations can vary. Depending on your JNDI environment, the connection-factory and destination-jndi-name elements must correctly reference the target named instances in the directory. Include the client library JAR files for your JMS provider in the WEB-INF/lib directory of your web application, or in another location from which the class loader loads them. Even when using an external JMS provider, BlazeDS uses the connection-factory and destination-jndi-name configuration properties to look up the necessary connection factory and destination instances.

J2EE restrictions on JMS

Section 6.6 of the J2EE 1.4 specification limits some of the JMS APIs that can be used in a J2EE environment. The following table lists the restricted APIs that are relevant to the JMSAdapter and the implications of the restrictions.

API	Implication of restriction	Alternative
javax.jms.MessageConsumer.get/setMessageListener	No asynchronous message delivery	Set the mode attribute of the delivery-settings configuration property to sync.
javax.jms.Connection.setExceptionListener	No notification of connection problems	Set the mode attribute of the delivery-settings configuration property to sync.
javax.jms.Connection.setClientID	No durable subscribers	Set the durable configuration property to false.
javax.jms.Connection.stop	Not an important implication	None

The JMSAdapter handles these restrictions by doing the following:

- Providing an explicit choice between synchronous and asynchronous message delivery.
- When asynchronous message delivery is specified and setMessageListener is restricted, a clear error message is sent to ask the user to switch to synchronous message delivery.
- When asynchronous message delivery is specified and setExceptionListener is restricted, a clear error message is sent to ask the user to switch to synchronous message delivery.
- Providing fine-grained tuning for synchronous message delivery so asynchronous message delivery is not needed.
- Providing clear error messages when durable subscribers cannot be used due to setClientID being restricted and asking the user to switch durable setting to false.

Part 6: Administering BlazeDS applications

Logging 149
Security 156
Clustering 167

Chapter 11: Logging

One tool that can help in debugging applications is the logging mechanism. You can perform both client-side and server-side logging. Client-side logging writes log messages from the Flex client to a file on the client computer. Server-side logging writes log messages from the BlazeDS server to a designated logging target, which can be the log location for the servlet container, System.out, or a custom location.

Topics

Client-side logging 149
Server-side logging 150
Monitoring and managing services 153

Client-side logging

For client-side logging, you can directly write messages to the log file, or configure the application to write messages generated by Flex to the log file. The Flash Debug Player has two primary methods of writing messages to a log file:

- The global `trace()` method

  The global `trace()` method prints a String to the log file. Messages can contain checkpoint information to signal that your application reached a specific line of code, or the value of a variable.

- Logging API

  The logging API, implemented by the TraceTarget class, provides a layer of functionality on top of the `trace()` method. For example, you can use the logging API to log debug, error, and warning messages generated by Flex during application execution.

The Flash Debug Player sends logging information to the flashlog.txt file. The operating system determines the location of this file, as the following table shows:

Operating system	Log file location
Windows 95/98/ME/2000/XP	`C:\Documents and Settings\`*`username`*`\Application Data\Macromedia\Flash Player\Logs`
Windows Vista	`C:\Users\`*`username`*`\AppData\Roaming\Macromedia\Flash Player\Logs`
Mac OS X	`/Users/`*`username`*`/Library/Preferences/Macromedia/Flash Player/Logs/`
Linux	`/home/`*`username`*`/.macromedia/Flash_Player/Logs/`

Use settings in the mm.cfg text file to configure the Flash Debug Player for logging. If this file does not exist, you can create it when you first configure the Flash Debug Player. The following table shows where to create the mm.cfg file for different operating systems:

Operating system	Create file in ...
Mac OS X	`/Library/Application Support/Macromedia`
Windows 95/98/ME	`%HOMEDRIVE%\%HOMEPATH%`
Windows 2000/XP	`C:\Documents and Settings\`*`username`*

Operating system	Create file in ...
Windows Vista	C:\Users*username*
Linux	/home/*username*

The mm.cfg file contains many settings that you can use to control logging. The following sample mm.cfg file enables error reporting and trace logging:

```
ErrorReportingEnable=1
TraceOutputFileEnable=1
```

Once logging is enabled, you can call the `trace()` method to write a String to the flashlog.txt file, as the following example shows:

```
trace("Got to checkpoint 1.");
```

To enable the logging of all Flex-generated debug messages to flashlog.txt, insert the following MXML line:

```
<mx:TraceTarget loglevel="2"/>
```

For information about client-side logging, see *Building and Deploying Adobe Flex 3 Applications*.

Server-side logging

You perform server-side logging for requests to and responses from the server. The following example shows a log message generated by the server:

```
[BlazDS] 05/13/2008 14:27:18.842 [ERROR] [Message.General] Exception when invoking service: 
(none) with message: Flex Message (flex.messaging.messages.AsyncMessageExt)
    clientId = 348190FC-2308-38D7-EB10-57541CC2440A
    correlationId =
    destination = foo
    messageId = E0BFF004-F697-611B-3C79-E38956BAB21B
    timestamp = 1210703238842
    timeToLive = 0
    body = dsafasdasd: asdasd
    hdr(DSEndpoint) = my-rtmp
    hdr(DSId) = 348190D5-130A-1711-B193-25C4415DFCF5
    hdr(DSValidateEndpoint) = true
exception: flex.messaging.MessageException: No destination with id 'chat' is registered with 
any service.
```

You can configure the logging mechanism to specify the following information in the message:

- The type of messages to log, called the log *level*. The available levels include `All`, `Debug`, `Error`, `Info`, `None`, and `Warn`. For example, you can choose to log `Error` messages, but not `Info` messages. For more information, see “Setting the logging level” on page 151.
- The optional String prefixed to every log message. In this example, the String is `[BlazDS]`. For more information, see “Setting logging properties” on page 152.
- The display of the date and time of the log message. In this example, the message contains the date and time: `05/13/2008 14:27:18.842`. For more information, see “Setting logging properties” on page 152.
- The display of the level of the log message. In this example, the message contains the level: `[ERROR]`. For more information, see “Setting logging properties” on page 152.

- The display of the category of the log message. The category provides information about the area of BlazeDS that generated the message. In this example, the message contains the level: `[Message.General]`. For more information, see "Setting logging properties" on page 152.
- The target of the log messages. By default, log messages are written to System.out. For more information, see "Setting the logging target" on page 152.

Configuring server-side logging

Configure server-side logging in the `logging` section of the Flex services-config.xml configuration file. After you edit services-config.xml, restart the BlazeDS server.

The following example shows a configuration that sets the logging level to `Debug`:

```
<logging>
    <target class="flex.messaging.log.ConsoleTarget" level="Debug">
        <properties>
            <prefix>[BlazeDS]</prefix>
            <includeDate>false</includeDate>
            <includeTime>false</includeTime>
            <includeLevel>false</includeLevel>
            <includeCategory>false</includeCategory>
        </properties>
        <filters>
            <pattern>Endpoint.*</pattern>
            <pattern>Service.*</pattern>
            <pattern>Configuration</pattern>
        </filters>
    </target>
</logging>
```

Setting the logging level

The `level` defines the types of messages written to the log. The following table describes the logging levels:

Logging level	Description
`All`	Logs all messages.
`Debug`	Logs debug message. Debug messages indicate internal Flex activities. Select the `Debug` logging level to include `Debug`, `Info`, `Warn`, and `Error` messages in your log files.
`Error`	Logs error messages. Error messages indicate when a critical service is not available or a situation restricts use of the application.
`Info`	Logs information messages. Information messages indicate general information to the developer or administrator. Select the `Info` logging level to include `Info` and `Error` messages in your log files.
`None`	No messages are logged.
`Warn`	Logs warning messages. Warning messages indicate that Flex encountered a problem with the application, but the application does not stop running. Select the `Warn` logging level to include `Warn` and `Error` messages in your log files.

In a production environment, you typically set the logging level to `Warn` to capture both warnings and error messages. If you prefer to ignore warning messages, set the level to `Error` to display only error messages.

Setting the logging target

By default, the server writes log messages to System.out. In the `class` attribute of the `target` element, you can specify `flex.messaging.log.ConsoleTarget` (default) to log messages to the standard output, or the `flex.messaging.log.ServletLogTarget` to log messages to the default logging mechanism for servlets for your application server.

Setting logging properties

The following table describes the logging properties:

Property	Description
`includeCategory`	Determines whether the log message includes the category. The category provides information about the area of BlazeDS that generated the message. The default value is `false`.
`includeDate`	Determines whether the log message includes the date. The default value is `false`.
`includeLevel`	Determines whether the log message includes the log level. The categories are Debug, Error, Info, and Warn. Specifies to include the message category in the logging message. The default value is `false`.
`includeTime`	Determines whether the log message includes the time. The default value is `false`.
`filters`	Specifies a pattern that defines the categories to log. The category of a log message must match the specified pattern to be written to the log. For more information, see "Setting a filtering pattern" on page 152.
`prefix`	Specifies the String prefixed to log messages. The default value is an empty String.

In the following example, you set the configuration properties to display the category, date, level, time, and set the prefix to `[BlazeDS]`:

```
<logging>
    <target class="flex.messaging.log.ConsoleTarget" level="Debug">
        <properties>
            <prefix>[BlazeDS]</prefix>
            <includeDate>true</includeDate>
            <includeTime>true</includeTime>
            <includeLevel>true</includeLevel>
            <includeCategory>true</includeCategory>
        </properties>
    </target>
</logging>
```

Setting a filtering pattern

The `<filters>` property lets you filter log messages based on the message category. If you omit a setting for the `<filters>` property, messages for all categories are written to the log.

The following example shows the first line of log messages from different categories:

```
[BlazeDS] 05/14/2008 12:52:52.606 [DEBUG] [Endpoint.HTTP] Received command: TCCommand
...
[BlazeDS] 05/14/2008 12:52:52.606 [DEBUG] [Message.General] Before invoke service: message-
service
...
[BlazeDS] 05/14/2008 12:52:52.606 [DEBUG] [Service.Message] Sending message: Flex Message
...
[BlazeDS] 05/14/2008 12:52:52.606 [DEBUG] [Message.Timing] After invoke service: message-
service;
```

To filter messages so only those messages in the Message.General and Endpoint categories appear, set the `<filters>` property as the following example shows:

```
<logging>
    <target class="flex.messaging.log.ConsoleTarget" level="Debug">
        <properties>
            <prefix>[BlazeDS]</prefix>
            <includeDate>false</includeDate>
            <includeTime>false</includeTime>
            <includeLevel>false</includeLevel>
            <includeCategory>false</includeCategory>
        </properties>
        <filters>
            <pattern>Endpoint.*</pattern>
            <pattern>Message.General</pattern>
        </filters>
    </target>
</logging>
```

Use the wildcard character (*) in the pattern to log messages from more than one category. To see messages for all endpoints, specify a pattern of `Endpoint.*`. To see messages for only an HTTP endpoint, specify a pattern of `Endpoint.HTTP`. To see all messages for all categories, specify a pattern of `*`.

You can use many different patterns as the value of the `pattern` element, such as the following:

- Client.*
- Client.FlexClient
- Client.MessageClient
- Configuration
- Endpoint.*
- Endpoint.General
- Endpoint.AMF
- Endpoint.FlexSession
- Endpoint.HTTP
- Endpoint.StreamingAMF
- Endpoint.StreamingHTTP
- Endpoint.Type
- Executor
- Message.*
- Message.General
- Message.Command.*

For the complete list of filter patterns, see the services-config.xml file in the *install_root*/resources/config directory.

Monitoring and managing services

BlazeDS uses Java Management Beans (MBeans) to provide run-time monitoring and management of the services configured in the services configuration file. The run-time monitoring and management console is an example of a Flex client application that provides access to the run-time MBeans. The application calls a Remoting Service destination, which is a Java class that makes calls to the MBeans.

About the run-time monitoring and management console

The run-time monitoring and management console is a Flex client application that provides access to the run-time MBeans in the data services web applications running on an application server. The console application calls a Remoting Service destination, which is a Java class that makes calls to the MBeans.

The console is in the ds-console web application; you run it by opening http://*server:port*/ds-console when the web application is running, where *server:port* contains your server and port names.

The tabs in the console provide several different views of run-time data for the data service applications running in the application server. You use the Application combobox to select the web application you want to monitor. You can use a slider control to modify the frequency with which the console polls the server for new data.

The general administration tab provides a hierarchical tree of all the MBeans in the selected web application. Other views target specific types of information. For example, tabs provide server, channel endpoint, and destination information. These tabs include dynamic data graphs for applicable properties.

One log manager tab shows the log categories set in the services-config.xml file and lets you add or remove log categories at run time. You change the log level (debug, info, warn, and so forth) for the log categories. For information about logging, see "Server-side logging" on page 150.

Note: *The run-time monitoring and management console exposes administrative functionality without authorization checks. Deploy the ds-console web application to the same application server that your Flex web application is deployed to, and lock down the ds-console web application by using J2EE security or some other means to protect access to it. For more information about J2EE security, see your application server documentation and http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Security.html.*

MBean creation and registration

The APIs exposed by the run-time MBeans do not affect or interact with the BlazeDS configuration files. You can use them for operations such as ending a stale connection or monitoring message throttling, but you cannot use them for operations such as registering a new service or altering the settings for an existing server component.

The run-time MBeans are instantiated and registered with the local MBean server by their corresponding managed resource. For example, when a MessageBroker is instantiated, it creates and registers a corresponding MessageBrokerControlMBean with the MBean server. The underlying resource sets the attributes for the run-time MBeans and they are exposed in a read-only manner by the MBean API. In some cases, an MBean can poll its underlying resource for an attribute value.

MBean naming conventions

The run-time MBean model starts at the MessageBrokerControlMBean. You can traverse the model by using attributes on MBeans that reference other MBeans. For example, to access an EndpointControlMBean, you start at the MessageBrokerControlMBean and get the `Endpoints` attribute. This attribute contains the ObjectNames of all endpoints that are registered with the management broker. It lets any management client generate a single ObjectName for the root MessageBrokerControlMBean, and from there, navigate to and manage any of the other MBeans in the system without having their ObjectNames.

The run-time MBean model is organized hierarchically; however, each registered MBean in the system has an MBean ObjectName and can be fetched or queried directly if necessary. The run-time MBeans follow ObjectName conventions to simplify registration, lookup, and querying. An ObjectName for an MBean instance is defined as follows:

```
{domain}:{key}={value}[,{keyN}={valueN}]*
```

You can provide any number of additional key-value pairs to uniquely identify the MBean instance.

All of the run-time MBeans belong to the flex.run-time domain. If an application name is available, it is also included in the domain as follows: `flex.runtime.application-name`.

Each of the run-time MBean ObjectNames contains the following keys:

Key	Description
type	Short type name of the resource managed by the MBean. The MessageBrokerControlMBean manages the flex.messaging.MessageBroker, so its type is MessageBroker.
id	The `id` value of the resource managed by the MBean. If no `id` is available on the resource, an `id` is created according to this strategy: `id = {type} + N` where *N* is a numeric increment for instances of this type.

An ObjectName can also contain additional optional keys.

The run-time MBeans are documented in the public BlazeDS Javadoc documentation. The Javadoc also includes documentation for the flex.management.jmx.MBeanServerGateway class, which the run-time monitoring and management console uses as a Remoting Service destination.

Creating a custom MBean for a custom ServiceAdapter class

You can write a custom MBean to expose metrics or management APIs for a custom ServiceAdapter. The following method of the ServiceAdapter class lets you connect your MBean.

```
public void setupAdapterControl(Destination destination);
```

Your custom ServiceAdapter control MBean should extend flex.management.runtime.messaging.services.ServiceAdapterControl. Your MBean must implement your custom MBean interface, which in turn must extend flex.management.runtime.messaging.services.ServiceAdapterControlMBean. ServiceAdapterControlMBean lets you add your custom MBean into the hierarchy of core BlazeDS MBeans.

The code in the following example shows how to implement a `setupAdapterControl()` method in your custom ServiceAdapter, where `controller` is an instance variable of type CustomAdapterControl (your custom MBean implementation class):

```
public void setupAdapterControl(Destination destination) {
    controller = new CustomAdapterControl(getId(), this, destination.getControl());
    controller.register();
    setControl(controller);
}
```

Your custom adapter can update metrics stored in its corresponding MBean by invoking methods or updating properties of `controller`, the MBean instance. Your custom MBean is passed a reference to your custom adapter in its constructor. Access this reference in your custom MBean by using the `protected ServiceAdapter serviceAdapter` instance variable to query its corresponding service adapter for its state, or to invoke methods on it.

Chapter 12: Security

BlazeDS security lets you control access to server-side destinations. A Flex client application may only connect to the server over a secure destination after its credentials have been validated (authentication), and may only perform authorized operations. By default, BlazeDS uses the security framework of the underlying J2EE application server to support authentication and authorization. However, you can define custom logic to perform authentication and authorization that does not rely on the application server.

Topics

Securing BlazeDS 156
Configuring security. 158
Basic authentication 162
Custom authentication. 162
Passing credentials to a proxy service 166

Securing BlazeDS

Authentication is the process by which users validate their identity to a system. *Authorization* is the process of determining what types of activities a user is permitted to perform on a system. After users are authenticated, they can be authorized to access specific resources.

In BlazeDS, a destination defines the server-side code through which a client connects to the server. You restrict access to a destination by applying a security constraint to the destination.

Security constraints

A security constraint ensures that a user is authenticated before accessing the destination. A security constraint can also require that the user is authorized against roles defined in a user store to determine if the user is a member of a specific role.

You can configure a security constraint to use either basic or custom authentication. The type of authentication you specify determines the type of response the server sends back to the client when the client attempts to access a secured destination with no credentials or invalid credentials. For basic authentication, the server sends an HTTP 401 error to indicate that authentication is required, which causes a browser challenge in the form of a login dialog box. For custom authentication, the server sends a fault to the client to indicate that authentication is required.

By default, security constraints uses custom authentication.

Login commands

BlazeDS uses a login command to check credentials and log the user into the application server. The way a J2EE application server implements security is specific to each server type. Therefore, BlazeDS includes login command implementations for Apache Tomcat, JBoss, Oracle Application Server, BEA WebLogic, IBM WebSphere, and Adobe JRun.

You can use a login command without roles to support authentication only. If you also want to use authorization, link the specified role references to roles that are defined in the user store of your application server.

A login command must implement the flex.messaging.security.LoginCommand interface. You can create a custom login command by implementing the LoginCommand interface. A custom login command can utilize the underlying J2EE application server, or implement its own validation mechanism.

Classes that implement the LoginCommand interface should also implement the LoginCommandExt interface in the uncommon scenario where the name stored in the java.security.Principal instance that is created upon successful authentication differs from the username passed into the authentication. Implementing the LoginCommandExt interface lets the LoginCommand instance return the resulting username so that it can be compared to the one stored in the Principal instance.

Secure channels and endpoints

In addition to providing security that lets you authenticate and authorize users, you can also use secure channels and endpoints to implement a secure transport mechanism. For example, the following AMF channel definitions use a nonsecure transport connection:

```
<!-- Non-polling AMF -->
<channel-definition id="samples-amf"
    class="mx.messaging.channels.AMFChannel">
    <endpoint url="http://{server.name}:8400/myapp/messagebroker/amf"
        type="flex.messaging.endpoints.AmfEndpoint"/>
</channel-definition>
```

BlazeDS provides a secure version of the AMF and HTTP channels and endpoints that transport data over an HTTPS connection. The names of the secure channels and endpoints all start with the string "Secure". The following example redefines the AMF channel to use a secure transport mechanism:

```
<!-- Non-polling secure AMF -->
<channel-definition id="my-secure-amf"
    class="mx.messaging.channels.SecureAMFChannel">
    <endpoint url="https://{server.name}:9100/dev/messagebroker/amfsecure"
        class="flex.messaging.endpoints.SecureAMFEndpoint"/>
</channel-definition>
```

For more information on using the secure transport mechanism, see “Channels and endpoints” on page 38.

Setting up security constraints

The steps to create security constraints for basic and custom authentication are almost the same. The difference between the two types of authentication is how you pass credentials to the server from the client application. The steps to set up security are as follows:

1 In the services-config.xml file, specify the login command for your application server by using the `<security>` tag. For more information, see “Configuring security” on page 158.

2 In the services-config.xml file, define a security constraint. A security constraint can specify basic or custom authentication, and it can define one or more roles. For more information, see “Configuring security” on page 158.

3 In your destination definition, reference the security constraint. For more information, see “Configuring a destination to use a security constraint” on page 159.

4 If you use custom authentication, pass the credentials to the server by calling the `ChannelSet.login()` method. A `result` event indicates that the login occurred successfully, and a `fault` event indicates the login failed. For more information, see “Custom authentication” on page 162. For basic authentication, you do not have to modify your Flex application. The browser opens a login dialog box when you first attempt to connect to the destination. Use that dialog box to pass credentials.

Configuring security

Typically, you configure security in the services-config.xml file. However, you can also add security settings specific to other services in the data-management-config.xml, messaging-config.xml, and other configuration files.

You reference the security constraints when you define a destination. Multiple destinations can share the same security constraints. The following example shows a security definition in the services-config.xml file that defines two security constraints:

```
<services-config>
    <security>
        <login-command class="flex.messaging.security.TomcatLoginCommand" server="Tomcat">
            <per-client-authentication>false</per-client-authentication>
        </login-command>

        <security-constraint id="trusted">
            <auth-method>Basic</auth-method>
            <roles>
                <role>guests</role>
                <role>accountants</role>
                <role>employees</role>
                <role>managers</role>
            </roles>
        </security-constraint>

        <security-constraint id="sample-users">
            <auth-method>Custom</auth-method>
            <roles>
                <role>sampleusers</role>
            </roles>
        </security-constraint>
    </security>
    ...
</services-config>
```

This example sets `class` and `server` properties for the `login-command` for the Tomcat server.

The following table describes the properties that you use to configure security:

Property	Description
`security`	The top-level tag in a security definition. Use this tag as a child tag of the `<services-config>` tag.
`login-command`	Specifies the login command and application server. BlazeDS supplies the following login commands: `TomcatLoginCommand` (for both Tomcat and JBoss), `JRunLoginCommand`, `WeblogicLoginCommand`, `WebSphereLoginCommand`, `OracleLoginCommand`. For JBoss, set the `class` property to `flex.messaging.security.TomcatLoginCommand`, but set the `server` property to `JBoss`. If you set the `server` property to `all`, the login command is used regardless of your application server.
`per-client-authentication`	Set to true to enable per-client authentication. The default value is `false`. For more information, see "Using per-client and per-session authentication" on page 159.
`security-constraint`	Define a security constraint.

Property	Description
auth-method	Specifies the authentication method as either Basic or Custom. The default value is Custom.
roles	Specifies the authorization roles. The roles specified by the destination must be enabled to perform the request operation on the application server. Roles are application-server specific. For example, in Tomcat, they are defined in conf/tomcat-user.xml file.

Using per-client and per-session authentication

By default, authentication is performed on a per-session basis and authentication information is stored in the FlexSession object associated with the Flex application. Therefore, if two client-side applications share the same session, when one is authenticated for the session, the other one is also authenticated. For example, multiple tabs in the same web browser share the same session. If you open a Flex application in one tab, and then authenticate, any copy of that application running in another tab is also authenticated.

Alternatively, you can enable per-client authentication by setting the `per-client-authentication` property of the `login-command` to `true`. Authentication information is then stored in the FlexClient object associated with the Flex application. Setting it to `true` allows multiple clients sharing the same session to have distinct authentication states. For example, two tabs of the same web browser could authenticate users independently.

Another reason to set the `per-client-authentication` property to `true` is to configure the server to reset the authentication state based on a browser refresh. Refreshing a browser does not create a new server session. Therefore, by default, the authentication state of a Flex application persists across a browser refresh. Setting `per-client-authentication` to `true` resets the authentication state based on the browser state.

The login commands that ship with BlazeDS rely on authentication information stored in the J2EE session on the server. If you use one of these login commands with per-client authentication, the authentication information for the different clients conflicts in the J2EE session. Therefore, to use per-client authentication, create a custom login command that does not store authentication information in the J2EE session.

Configuring a destination to use a security constraint

Define a security constraint in the `security` section of the Flex services configuration file and reference it in destinations. The following example shows a security constraint that is referenced in two destination definitions:

```
<security>
    <login-command class="flex.messaging.security.TomcatLoginCommand" server="Tomcat">
        <per-client-authentication>false</per-client-authentication>
    </login-command>

    <security-constraint id="trusted">
        <auth-method>Basic</auth-method>
        <roles>
            <role>employees</role>
            <role>managers</role>
        </roles>
    </security-constraint>
</security>

<service>
    <destination id="SecurePojo1">
        ...
        <security>
            <security-constraint ref="trusted"/>
```

```
        </security>
    </destination>

    <destination id="SecurePojo2">
        ...
        <security>
            <security-constraint ref="trusted"/>
        </security>
    </destination>
    ...
</service>
```

When a security constraint applies to only one destination, you can declare the security constraint in the destination definition, as shown in the following example:

```
<destination id="roDest">
    ...
    <security>
        <security-constraint>
            <auth-method>Custom</auth-method>
            <roles>
                <role>roDestUser</role>
            </roles>
        </security-constraint>
    </security>
</destination>
```

You can set a default security constraint at the service level. Any destination of a service that does not have an explicit security constraint uses the default security constraint. The following example shows a default security constraint configuration:

```
<service>
...
    <default-security-constraint ref="sample-users"/>
</service>
```

The following table describes the properties that you use to configure security for a destination:

Property	Description
`destination`	
`security`	Specifies a security constraint for the destination. Use the `ref` attribute to specify the name of a security constraint. Or, you can define a local security constraint by using the syntax described in the section “Configuring security” on page 158.
`properties`	

Property	Description
server	Specifies security constraints specific to the adapter used by the destination. Use the `ref` attribute to specify the name of a security constraint. For the ASObjectAdapter adapter, you can use the following properties to specify a security constraint: • `count-security-constraint` • `create-security-constraint` • `read-security-constraint` • `update-security-constraint` • `delete-security-constraint` For the ActionScriptAdapter used with the Messaging Service, you can use the following properties to specify a security constraint: • `send-security-constraint` • `subscribe-security-constraint` For more information, see "Using the Messaging Service" on page 122.
include-methods	For Remoting Service destinations, specifies the list of methods that the destination is able to call. For more information, see "Restricting method access on a Remoting Service destination" on page 161.
exclude-methods	For Remoting Service destinations, specifies the list of methods that the destination is prohibited from calling. For more information, see "Restricting method access on a Remoting Service destination" on page 161.
service	
default-security-constraint	Specifies the default security constraint for all destinations that do not explicitly define a security constraint.

Restricting method access on a Remoting Service destination

For Remoting Service destinations, you can declare destinations that only allow invocation of methods that are explicitly included in an include list. You can use this feature with or without a security constraint on the destination. Any attempt to invoke a method that is not in the `include-methods` list results in a `fault` event. For even more granular security, you can assign a security constraint to one or more methods in the `include-methods` list. If a destination-level security constraint is defined, it is tested first and method-level constraints are checked second.

The following example shows a destination that contains an `include-methods` list and a method-level security constraint on one of the methods in the list:

```
<destination id="sampleIncludeMethods">
    <properties>
        <source>my.company.SampleService</source>
        <include-methods>
            <method name="fooMethod"/>
            <method name="barMethod" security-constraint="admin-users"/>
        </include-methods>
    </properties>
    <security>
        <security-constraint ref="sample-users"/>
    </security>
</destination>
```

You can also use an `exclude-methods` element, which is like the `include-methods` element, but operates in the reverse direction. All public methods on the source class are visible to clients except for the methods in this list. Use `exclude-methods` if only a few methods on the source class must be hidden and no methods require a tighter security constraint than the destination.

Basic authentication

Basic authentication relies on standard J2EE basic authentication from the application server. When you use basic authentication to secure access to destinations, you typically secure the endpoints of the channels that the destinations use in the web.xml file. You then configure the destination to access the secured resource to be challenged for a user name (principal) and password (credentials).

For basic authentication, BlazeDS checks that a currently authenticated principal exists before routing any messages to the destination. If no authenticated principal exists, the server returns an HTTP 401 error message to indicate that authentication is required. In response to the HTTP 401 error message, the browser prompts the user to enter a user name and password. The web browser performs the challenge independently of the Flex client application. After the user successfully logs in, they remain logged in until the browser is closed.

The following example shows a security constraint definition that specifies roles for authorization:

```
<security-constraint id="privileged-users">
    <auth-method>Basic</auth-method>
    <roles>
        <role>privilegedusers</role>
        <role>admins</role>
    </roles>
</security-constraint>
```

Custom authentication

For custom authentication, the client application passes credentials to the server without relying on the browser. Although you apply security constraints to a destination, you actually log in and log out of the channels associated with the destination. Therefore, to send authentication credentials to a destination that uses custom authentication, you specify a user name and password as arguments to the `ChannelSet.login()` method. You remove credentials by calling the `ChannelSet.logout()` method.

***Note:** In the previous release of BlazeDS, you sent credentials to a destination by calling the `setCredentials()` method. The `setCredentials()` method did not actually pass the credentials to the server until the first attempt by the component to connect to the server. Therefore, if the component issued a `fault` event, you could not be certain whether the fault happened because of an authentication error, or for another reason. The `ChannelSet.login()` method connects to the server when you call it so that you can handle an authentication issue immediately. Although you can continue to use the `setCredentials()` method, Adobe recommends that you use the `ChannelSet.login()` method.*

Because multiple destinations can use the same channels, and corresponding ChannelSet object, logging in to one destination logs the user in to any other destination that uses the same channel or channels. If two components apply different credentials to the same ChannelSet object, the last credentials applied are used. If multiple components use the same authenticated ChannelSet object, calling the `logout()` method logs all components out of the destinations.

The `login()` and `logout()` methods return an AsyncToken object. Assign event handlers to the AsyncToken object for the `result` event to handle a successful call, and for the `fault` event to handle a failure.

How the server processes the `logout()` method depends on the setting of the `per-client-authentication` property:

- If the `per-client-authentication` property is `false` (default), the `logout()` method invalidates the current session. A new FlexSession object is automatically created to replace it, and the new session does not contain any attributes or an authenticated information. If authentication information is stored at the session level, and a client disconnect results in the FlexSession being invalidated, authenticated information is also cleared.
- If the `per-client-authentication` property is `true`, the `logout()` method clears the authentication information stored in the FlexClient object, but does not invalidate the FlexClient object or FlexSession object.

***Note:** Calling the `login()`, `setCredentials()`, or `setRemoteCredentials()` method has no effect when the `useProxy` property of a component is set to `false`.*

Custom authentication example

The following example uses the `ChannelSet.login()` and `ChannelSet.logout()` methods with a RemoteObject control. This application performs the following actions:

- Creates a ChannelSet object in the `creationComplete` handler that represents the channels used by the RemoteObject component.
- Passes credentials to the server by calling the ROLogin() function in response to a Button `click` event.
- Uses the RemoteObject component to send a String to the server in response to a Button `click` event. The server returns the same String back to the RemoteObject component.
- Uses the result event of the RemoteObject component to display the String in a TextArea control.
- Logs out of the server by calling the ROLogout() function in response to a Button `click` event.

```
<?xml version="1.0"?>
<!-- security/SecurityConstraintCustom.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="100%" height="100%"
    creationComplete="creationCompleteHandler();">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.messaging.config.ServerConfig;
            import mx.rpc.AsyncToken;
            import mx.rpc.AsyncResponder;
            import mx.rpc.events.FaultEvent;
            import mx.rpc.events.ResultEvent;
            import mx.messaging.ChannelSet;
```

```
// Define a ChannelSet object.
public var cs:ChannelSet;

// Define an AsyncToken object.
public var token:AsyncToken;

// Initialize ChannelSet object based on the
// destination of the RemoteObject component.
private function creationCompleteHandler():void {
    if (cs == null)
     cs = ServerConfig.getChannelSet(remoteObject.destination);
}

// Login and handle authentication success or failure.
private function ROLogin():void {
    // Make sure that the user is not already logged in.
    if (cs.authenticated == false) {
        token = cs.login("sampleuser", "samplepassword");
        // Add result and fault handlers.
     token.addResponder(new AsyncResponder(LoginResultEvent, LoginFaultEvent));
    }
}

// Handle successful login.
private function LoginResultEvent(event:ResultEvent, token:Object=null):void  {
    switch(event.result) {
        case "success":
            authenticatedCB.selected = true;
        break;
        default:
    }
}

// Handle login failure.
private function LoginFaultEvent(event:FaultEvent, token:Object=null):void {
    switch(event.fault.faultCode) {
        case "Client.Authentication":
            default:
                authenticatedCB.selected = false;
                Alert.show("Login failure: " + event.fault.faultString);
    }
}

// Logout and handle success or failure.
private function ROLogout():void {
    // Add result and fault handlers.
    token = cs.logout();
    token.addResponder(new AsyncResponder(LogoutResultEvent,LogoutFaultEvent));
}

// Handle successful logout.
private function LogoutResultEvent(event:ResultEvent, token:Object=null):void {
    switch (event.result) {
        case "success":
            authenticatedCB.selected = false;
            break;
        default:
    }
}
```

```
            // Handle logout failure.
            private function LogoutFaultEvent(event:FaultEvent, token:Object=null):void {
                Alert.show("Logout failure: " + event.fault.faultString);
            }

            // Handle message recevied by RemoteObject component.
            private function resultHandler(event:ResultEvent):void {
                ta.text += "Server responded: "+ event.result + "\n";
            }

            // Handle fault from RemoteObject component.
            private function faultHandler(event:FaultEvent):void {
                ta.text += "Received fault: " + event.fault + "\n";
            }
        ]]>
    </mx:Script>

    <mx:HBox>
        <mx:Label text="Enter a text for the server to echo"/>
        <mx:TextInput id="ti" text="Hello World!"/>
        <mx:Button label="Login"
            click="ROLogin();"/>
        <mx:Button label="Echo"
            enabled="{authenticatedCB.selected}"
            click="remoteObject.echo(ti.text);"/>
        <mx:Button label="Logout"
            click="ROLogout();"/>
        <mx:CheckBox id="authenticatedCB"
            label="Authenticated?"
            enabled="false"/>
    </mx:HBox>
    <mx:TextArea id="ta" width="100%" height="100%"/>

    <mx:RemoteObject id="remoteObject"
        destination="remoting_AMF_SecurityConstraint_Custom"
        result="resultHandler(event);"
        fault="faultHandler(event);"/>
</mx:Application>
```

Configure Tomcat for custom authentication

Perform the following configuration steps to use custom authentication with Tomcat:

1 Place the flex-tomcat-common.jar file in the tomcat/lib/blazeds directory.

2 Place the flex-tomcat-server.jar file in the tomcat/lib/blazeds directory.

3 Edit the tomcat/conf/catalina.properties file to add the following path to the `common.loader` property:

```
${catalina.home}/lib/blazeds/*.jar
```

4 Edit the tomcat/conf/context.xml file to add the following tag to the Context descriptors:

```
<Valve className="flex.messaging.security.TomcatValve"/>
```

5 Restart Tomcat.

You can now authenticate against the current Tomcat realm. Typically user information is in the conf/tomcat-users.xml file. For more information, see the Tomcat documentation.

Passing credentials to a proxy service

A security constraint defines the security restrictions on a BlazeDS destination. However, you can also use BlazeDS as a proxy to a remote HTTP service or web service.

If the remote HTTP service or web service requires credentials to control access, use the `setRemoteCredentials()` method to pass the credentials. For example, when using an HTTPService component, you can pass credentials to a secured JSP page. Passing remote credentials is distinct from passing credentials to satisfy a security constraint that you defined on the destination. However, you can use the two types of credentials in combination. The credentials are sent to the destination in message headers.

You can also call the `setRemoteCredentials()` method for Remoting Service destinations managed by an external service that requires user name and password authentication, such as a ColdFusion Component (CFC).

The following example shows ActionScript code for sending remote user name and remote password values to a destination. The destination configuration passes these credentials to the actual JSP page that requires them.

```
var employeeHTTP:mx.rpc.http.HTTPService = new HTTPService();
employeeHTTP.destination = "secureJSP";
employeeHTTP.setRemoteCredentials("myRemoteUserName", "myRemotePassword");
employeeHTTP.send({param1: 'foo'});
```

As an alternative to setting remote credentials on a client at run time, you can set remote credentials in `remote-username` and `remote-password` elements in the properties section of a server-side destination definition. The following example shows a destination that specifies these properties:

```
<destination id="samplesProxy">
    <channels>
        <channel ref="samples-amf"/>
    </channels>
    <properties>
        <url>
            http://someserver/SecureService.jsp
        </url>
        <remote-username>johndoe</remote-username>
        <remote-password>opensaysme</remote-password>
    </properties>
</destination>
```

Chapter 13: Clustering

Clusters are a group of servers that function as a single server. BlazeDS supports message and data routing across a cluster so that clients connected to different servers in the cluster can exchange information. BlazeDS also supports channel failover across the cluster. If a server in the cluster fails, a client connected to that server automatically attempts to connect to another server in the cluster.

Topics

Server clustering 167
Handling channel failover 167
Cluster-wide message and data routing 169
Configuring clustering 170

Server clustering

BlazeDS provides a software clustering capability where you install BlazeDS on each server in the cluster. BlazeDS clustering provides two main features:

- Cluster-wide message and data routing

  A MessageService or DataService client connected to one server can exchange messages and data with a client connected to another server. By distributing client connections across the cluster, one server does not become a processing bottleneck.

- Channel failover support

  If the channel over which a Flex client communicates with the server fails, the client reconnects to the same destination on a different server or channel. Clustering handles failover for all channel types for RPC (remote procedure call) and Messaging Service destinations.

BlazeDS supports both vertical and horizontal clusters (where multiple instances of BlazeDS are deployed on the same or different physical servers). When a BlazeDS instance starts, clustered destinations broadcast their availability and reachable channel endpoint URIs to all peers in the cluster.

You can also implement hardware clustering, which requires additional hardware. Hardware clustering typically uses a set of load balancers positioned in front of servers. Load balancers direct requests to individual servers and pin client connections from the same client to that server. Although hardware clustering is possible without using BlazeDS software clustering, it can also work in conjunction with software clustering.

Handling channel failover

When a client-side application initially connects through a channel to a destination on a clustered server, it receives a set of channel endpoint URIs for that channel and destination across all servers in the cluster. These endpoints are assigned to the `Channel.failoverURIs` property.

If a connection through a channel to a destination fails, the client uses the `Channel.failoverURIs` property to reconnect to the same channel and destination on a different server. The client attempts to connect to each server in the cluster until it successfully connects.

A destination can define multiple channels. If the client fails to connect to a server in the cluster using the original channel and destination, the client tries to reconnect to the original server and destination through a different channel. If that fails, the client then attempts to connect to a different server. This process repeats until the client has attempted to connect to all servers over all channels defined for the destination.

***Note:** The list of URIs across the cluster for each channel is keyed by the channel id. Therefore, the destination configuration on all servers in the cluster must define the same set of channels.*

The crossdomain.xml file

A crossdomain.xml file provides a way for a server to indicate that its data and documents are available to SWF files served from specific domains, or from all domains. All servers in the cluster must have compatible crossdomain.xml files so that if one server fails and its clients are redirected to another server in the cluster, the second server can communicate with it.

For more information on the crossdomain.xml file, see the *Adobe Flex 3 Developer Guide*.

Setting component properties for clustering

Different client-side components, such as Producer, Consumer, DataService, and the RPC components, define properties that control the action of the component when a channel connection fails. The following table describes these properties:

Component	Property	Description
Producer	`reconnectAttempts` `reconnectInterval`	`reconnectAttempts` specifies the number of times the component attempts to reconnect to a channel following a connection failure. If the component fails to connect after the specified number of attempts, it dispatches a `fault` event. The `fault` event does not occur until the component attempts to connect to all servers in the cluster, across all available channels. A value of -1 specifies to continue indefinitely and a value of 0 disables attempts. The default value is 0. `reconnectInterval` specifies the interval, in milliseconds, between attempts to reconnect. Setting the value to 0 disables reconnection attempts. Set the value to a period of time long enough for the underlying protocol to complete its attempt and time out if necessary. The default is value 5000 (5 seconds).
Consumer DataService	`resubscribeAttempts` `resubscribeInterval`	`resubscribeAttempts` specifies the number of times the component attempts to resubscribe to the server following a connection failure. If the component fails to subscribe to a server after the specified number of attempts, it dispatches a `fault` event. The `fault` event does not occur until the component attempts to connect to all servers in the cluster, across all available channels. A value of -1 specifies to continue indefinitely and a value of 0 disables attempts. For the Consumer component, the default value is 5. For the DataService component, the default value is -1. `resubscribeInterval` specifies the interval, in milliseconds, between attempts to resubscribe. Setting the value to 0 disables resubscription attempts. Set the value to a period of time long enough for the underlying protocol to complete its attempt and time out if necessary. The default is value 5000 (5 seconds).

Component	Property	Description
HTTPService RemoteObject WebService Producer Consumer DataService	`requestTimeout`	The request timeout, in seconds, for sent messages. If an acknowledgment, response, or fault is not received from the remote destination before the timeout is reached, the component dispatches a `fault` event on the client. A value less than or equal to zero prevents request timeout. For more information, see "Cluster-wide message and data routing" on page 169.
Channel services-config.xml file	`connectTimeout` `connect-timeout-seconds`	`Channel.connectTimeout` and the `connect-timeout-seconds` property in the services-config.xml file specify the connect timeout, in seconds, for the channel. A value of 0 or below indicates that a connect attempt never times out on the client. For channels that are configured to failover, this value is the total time to wait for a connection to be established. It is not reset for each failover URI that the channel attempts to connect to.

To speed up the process when channels are failing over, or to detect a hung connection attempt and advance past it, you can set the `Channel.connectTimeout` property on your channels in ActionScript, or set the `<connect-timeout-seconds>` property in your channel configuration, as the following example shows:

```
<channel-definition id="triage-amf" class="mx.messaging.channels.AMFChannel">
    <endpoint url="…" class="flex.messaging.endpoints.AMFEndpoint"/>
    <properties>
        <serialization>
            <ignore-property-errors>true</ignore-property-errors>
            <log-property-errors>true</log-property-errors>
        </serialization>
        <connect-timeout-seconds>2</connect-timeout-seconds>
    </properties>
</channel-definition>
```

Cluster-wide message and data routing

Messaging and data service destinations broadcast messages or changes to the corresponding destination on the other servers in the cluster. Therefore, MessageService or DataService components in one client application connected to one server can receive messages or changes that a different client sends or commits to a different server.

When a client application subscribes to a particular service on one server in a cluster, and the server fails, the client can direct further messages to another server in the same cluster. This feature is available for all service types defined in the services configuration file.

For the Message Service, both failover and replication of application messaging state are supported. For the Remoting Service and Proxy Service, failover only is supported. When Remoting Service and Proxy Service use the AMF channel, the first request that fails generates a message `fault` event, which invalidates the current channel. The next time a request is sent, the client looks for another channel, which triggers a failover event followed by a successful send to the secondary server that is still running.

The `requestTimeout` property causes the client to dispatch a `fault` event if it has not received a response (either a response, acknowledgement, or fault) from the remote destination within the interval. This action is more useful with read-only calls over HTTP if the regular network request timeout is longer than desirable.

Use the `requestTimeout` property only with calls that create, update, or delete data on the server. If a request timeout occurs, query the server for its current state before you decide whether to retry the operation. Messages sent by Producer components or calls made by RPC services are not automatically queued or resent. These messages are not *idempotent*, which means they do not produce the same result no matter how many times they are performed. The application must determine whether it can safely resend a message or call the RPC service again.

Configuring clustering

To configure clustering, configure JGroups by using the jgroups-tcp.xml file, and configure BlazeDS by using the services-config.xml file.

Configuring JGroups

BlazeDS uses JGroups, an open source clustering platform to implement clustering. By default, a BlazeDS web application is not configured to support JGroups. The jgroups.jar file and the jgroups-*.xml properties files are located in the resources/clustering folder of the BlazeDS installation. Before you can run an application that uses JGroups, copy the jgroups.jar file to the WEB-INF/lib, and copy the jgroups-*.xml files to the WEB-INF/flex directory.

The JGroups configuration file determines how a clustered data service destination on one server broadcasts changes to corresponding destinations on other servers. JGroups supports UDP multicast or TCP multicast to a list of configured peer server addresses. Adobe recommends using the TCP option (TCP and TCPPING options in jgroups-tcp.xml). A unique version of this file is required for each server in the cluster. In the TCP element, use the `bind_addr` attribute to reference the current server by domain name or IP. In the TCPPING element, you reference the other peer servers.

For example, if the cluster contains four servers, the elements in the first jgroups-tcp.xml config file would look like the following example:

```
<TCP bind_addr="64.13.226.47" start_port="7800" loopback="false"/>
<TCPPING timeout="3000"
    initial_hosts="data1xd.com[7800],data2xd.com[7800],data3xd.com[7800]"
    port_range="1"
    num_initial_members="3"/>
<FD_SOCK/>
<FD timeout="10000" max_tries="5" shun="true"/>
<VERIFY_SUSPECT timeout="1500" down_thread="false" up_thread="false"/>
<MERGE2 max_interval="10000" min_interval="2000"/>
<pbcast.NAKACK gc_lag="100" retransmit_timeout="600,1200,2400,4800"/>
<pbcast.STABLE stability_
    delay="1000" desired_avg_gossip="20000" down_thread="false"
    max_bytes="0" up_thread="false"/>
<pbcast.GMS
    print_local_addr="true" join_timeout="5000" join_retry_timeout="2000" shun="true"/>
```

- The TCP `bind_addr` and `initial_hosts` attributes vary for each server in the cluster. The `bind_addr` property must be set to the IP or DNS name depending on how the /etc/hosts file is set up on that computer.
- The TCPPING `initial_hosts` setting lists all other nodes in the cluster, not including the node corresponding to the configuration file. Set the `num_initial_members` to the number of other nodes in the cluster.
- The FD (Failure Detection) setting specifies to use sockets to other nodes for failure detection. This node does not consider other nodes to be out of the cluster unless the socket to the node is closed.

For information about JGroups, see http://www.jgroups.org/javagroupsnew/docs/index.html.

Configuring BlazeDS

Define one or more software clusters in the `clusters` section of the services-config.xml file. Each `cluster` element must have an `id` attribute, and a `properties` attribute that points to a JGroups properties file. The following example shows the definition of a cluster in the services-config.xml file:

```
<?xml version="1.0"?>
<services-config>
    ...
```

```
    <clusters>
        <cluster id="default-cluster" properties="jgroups-tcp.xml"/>
    </clusters>
    ...
</services-config>
```

The cluster element can take the following attributes:

- `id` Identifier of the cluster definition. Use the same `id` value across all members of a cluster.
- `properties` The name of a JGroups properties file.
- `default` A value of `true` sets a cluster definition as the default cluster. All destinations use this cluster unless otherwise specified by the `cluster` property in the destination definition. The `default` attribute lets you enable clustering for all destinations without having to modify your destination definitions.
- `url-load-balancing` The server gathers the endpoint URLs from all servers in the cluster and sends them to clients during the initial connection handshake. Therfore, clients can implement failover between servers with different domain names. If you are using hardware load balancing for HTTP connections, you can disable this logic by setting the `url-load-balancing` attribute to `false`. In this mode, the client connects to each channel using the same URL and the load balancer routes you to an available server. When `url-load-balancing` is `true` (the default), you cannot use `{server.name}` and `{server.port}` tokens in channel endpoint URLs. Instead, specify the unique server name and port clients use to reach each server.

The following example shows a channel definition with a valid server name:

```
<channel-definition id="my-streaming-amf"
    class="mx.messaging.channels.StreamingAMFChannel">
    <endpoint
        url="http://companyserver:8400/blazeds/messagebroker/streamingamf"
        class="flex.messaging.endpoints.StreamingAMFEndpoint"/>
</channel-definition>
```

Notice that the channel endpoint omits the `{server.name}` and `{server.port}` tokens in the endpoint URL.

Reference the cluster in the `network` section of a destination. The value of the `ref` attribute must match the value of the `id` attribute of the cluster, as the following example shows:

```
<destination id="MyDestination">
    ...
    <properties>
        <network>
            <cluster ref="default-cluster"/>
        </network>
    </properties>
    ...
</destination>
```

If you do not specify a `ref` value, the destination uses the default cluster definition, if one is defined.

Configuring cluster message routing

For publish-subscribe messaging, you have the option of using the server-to-server messaging feature to route messages sent on any publish-subscribe messaging destination. Set server-to-server messaging in a configuration property on a messaging destination definition to make each server store the subscription information for all clients in the cluster. When you enable server-to-server messaging, data messages are routed only to the servers with active subscriptions, but subscribe and unsubscribe messages are broadcast across the cluster.

Server-to-server messaging provides the same functionality as the default messaging adapter, but improves the scalability of the messaging system for many use cases in a clustered environment. In many messaging applications, the number of subscribe and unsubscribe messages is much lower than the number of data messages that are sent. To enable server-to-server messaging, you set the value of the `cluster-message-routing` property in the server section of a messaging destination to `server-to-server`. The default value is `broadcast`. The following example shows a `cluster-message-routing` property set to `server-to-server`:

```
<destination id="MyTopic">
    <properties>
        ...
        <server>
            ...
            <cluster-message-routing>server-to-server</cluster-message-routing>
        </server>
    </properties>
    ...
</destination>
```

Viewing cluster information in the server-side log

The server-side logging mechanism captures log messages from the BlazeDS server. You can set the `pattern` attribute in the services-config.xml file to the wildcard character (*) to get all logging messages, including clustering. Alternatively, you can use the `pattern` parameter to filter messages. The following example filters log messages to write out only clustering and endpoint messages:

```
<logging>
    <target class="flex.messaging.log.ConsoleTarget" level="Debug">
        <properties>
            <prefix>[BlazeDS]</prefix>
            <includeDate>false</includeDate>
            <includeTime>false</includeTime>
            <includeLevel>false</includeLevel>
            <includeCategory>false</includeCategory>
        </properties>
        <filters>
            <pattern>Service.Cluster</pattern>
            <pattern>Endpoint.*</pattern>
        </filters>
    </target>
</logging>
```

For more information on logging, see “Logging” on page 149.

Part 7: Additional programming topics

Run-time configuration 174
The Ajax client library 179
Extending applications with factories 188
Message delivery with adaptive polling 193
Measuring message processing performance 199

Chapter 14: Run-time configuration

Using run-time configuration provides server APIs that let you create, modify, and delete services, destinations, and adapters dynamically on the server without the need for any Data Services configuration files.

Topics

About run-time configuration 174
Configuring components with a bootstrap service 175
Configuring components with a remote object 175
Accessing dynamic components with a Flex client application 177
Accessing dynamic components with a Flex client application 177

About run-time configuration

Using run-time configuration provides server-side APIs that let you create and delete data services, adapters, and destinations, which are collectively called components. You can create and modify components even after the server is started.

There are many reasons why you might want to create components dynamically. For example, consider the following use cases:

- You want a separate destination for each doctor's office that uses an application. Instead of manually creating destinations in the configuration files, you want to create them dynamically based on information in a database.
- You want a configuration application to dynamically create, delete, or modify destinations in response to some user input.

There are two primary ways to perform dynamic configuration. The first way is to use a custom bootstrap service class that the MessageBroker calls to perform configuration when the BlazeDS server starts up. This is the preferred way to perform dynamic configuration. The second way is to use a RemoteObject instance in a Flex client to call a remote object (Java class) on the server that performs dynamic configuration.

The Java classes that are configurable are MessageBroker, AbstractService and its subclasses, Destination and its subclasses, and ServiceAdapter and its subclasses. For example, you use the flex.messaging.services.HTTPProxy-Service class to create an HTTP proxy service, the flex.messaging.services.http.HTTPProxyAdapter class to create an HTTP proxy adapter, and the flex.messaging.services.http.HTTPProxyDestination class to create an HTTP proxy destination. Some properties (such as `id`) cannot be changed when the server is running.

The API documentation for these classes is included in the public BlazeDS Javadoc documentation.

Configuring components with a bootstrap service

To dynamically configure components at server startup, you create a custom Java class that extends the flex.messaging.services.AbstractBootstrapService class and implements the `initialize()` method of the AbstractBootstrapService class. You can also implement the `start()`, and `stop()` methods of the AbstractBootstrapService class; these methods provide hooks to server startup and shutdown in case you need to do special processing, such as starting or stopping the database as the server starts or stops. The following table describes these methods:

Method	Descriptions
`public abstract void initialize(String id, ConfigMap properties)`	Called by the MessageBroker after all of the server components are created, but just before they are started. Components that you create in this method are started automatically. Usually, you use this method rather than the `start()` and `stop()` methods because you want the components configured before the server starts. The `id` parameter specifies the ID of the AbstractBootstrapService. The `properties` parameter specifies the properties for the AbstractBootstrapService.
`public abstract void start()`	Called by the MessageBroker as server starts. You must manually start components that you create in this method.
`public abstract void stop()`	Called by the MessageBroker as server stops.

You must register custom bootstrap classes in the services section of the services-config.xml file, as the following example shows. Services are loaded in the order specified. You generally place the static services first, and then place the dynamic services in the order in which you want them to become available.

```
<services>
    <service-include file-path="remoting-config.xml"/>
    <service-include file-path="proxy-config.xml"/>
    <service-include file-path="messaging-config.xml"/>
    <service class="dev.service.MyBootstrapService1" id="bootstrap1"/>
    <service id="bootstrap2" class="my.company.BootstrapService2">
    <!-- Bootstrap services can also have custom properties that can be
        processed in initialize method -->
        <properties>
            <prop1>value1</prop1>
            <prop2>value2</prop2>
        </properties>
    </service>>
</services>
```

***Note:** The resources/config/bootstrapservices folder of the BlazeDS installation contains sample bootstrap services.*

Configuring components with a remote object

You can use a remote object to configure server components from a Flex client at run time. In this case, you write a Java class that calls methods directly on components, and you expose that class as a remote object (Remoting Service destination) that you can call from a RemoteObject in a Flex client application. The component APIs you use are identical to those you use in a bootstrap service, but you do not extend the `AbstractBootstrapService` class.

The following example shows a Java class that you could expose as a remote object to modify a Message Service destination from a Flex client application:

```
package runtimeconfig.remoteobjects;
```

```
/*
 * The purpose of this class is to dynamically change a destination that
 * was created at startup.
 */
import flex.messaging.MessageBroker;
import flex.messaging.MessageDestination;
import flex.messaging.config.NetworkSettings;
import flex.messaging.config.ServerSettings;
import flex.messaging.config.ThrottleSettings;
import flex.messaging.services.MessageService;

public class ROMessageDestination
{
    public ROMessageDestination()
    {
    }

    public String modifyDestination(String id)
    {
        MessageBroker broker = MessageBroker.getMessageBroker(null);
        //Get the service
        MessageService service = (MessageService) broker.getService(
            "message-service");

        MessageDestination msgDest =
            (MessageDestination)service.createDestination(id);
            NetworkSettings ns = new NetworkSettings();
            ns.setSessionTimeout(30);
            ns.setSharedBackend(true);
            ...
            msgDest.start();
            return "Destination " + id + " successfully modified";
    }
}
```

The following example shows an MXML file that uses a RemoteObject component to call the `modifyDestination()` method of an ROMessageDestination instance:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" creationComplete="run()">
    <mx:RemoteObject destination="ROMessageDestination" id="ro"
        fault="handleFault(event)"
        result="handleResult(event)"/>
    <mx:Script>
        <![CDATA[
            import mx.rpc.events.*;
            import mx.rpc.remoting.*;
            import mx.messaging.*;
            import mx.messaging.channels.*;

            public var faultstring:String = "";
            public var noFault:Boolean = true;
            public var result:Object = new Object();
            public var destToModify:String = "MessageDest_runtime";

            private function handleResult(event:ResultEvent):void {
                result = event.result;
                output.text += "-> remoting result: " + event.result + "\n";
            }

            private function handleFault(event:FaultEvent):void {
```

```
                //noFault = false;
                faultstring = event.fault.faultString;
                output.text += "-> remoting fault: " + event.fault.faultString +
                    "\n";
            }

            private function run():void {
                ro.modifyDestination(destToModify);
            }
        ]]>
    </mx:Script>
    <mx:TextArea id="output" height="200" percentWidth="80" />
</mx:Application>
```

Accessing dynamic components with a Flex client application

Using a dynamic destination with a client-side data services component, such as an HTTPService, RemoteObject, WebService, Producer, or Consumer component, is essentially the same as using a destination configured in the services-config.xml file.

It is a best practice to specify a ChannelSet and channel when you use a dynamic destination, and this is required when there are not application-level default channels defined in the services-config.xml file. If you do not specify a ChannelSet and Channel when you use a dynamic destination, BlazeDS attempts to use the default application-level channel assigned in the `default-channels` element in the services-config.xml file. The following example shows a default channel configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
    <services>
        ...
        <default-channels>
            <channel ref="my-polling-amf" />
        </default-channels>
        ...
    </services>
    ...
</services-config>
```

You have the following options for adding channels to a ChannelSet:

- Create channels on the client, as the following example shows:

```
...
var cs:ChannelSet = new ChannelSet();
var pollingAMF:AMFChannel = new AMFChannel("my-amf",
    "http://servername:8400/messagebroker/amfpolling");
cs.addChannel(pollingAMF);
...
```

- If you have compiled your application with the server-config.xml file, use the `ServerConfig.getChannel()` method to retrieve the channel definition, as the following example shows:

```
var cs:ChannelSet = new ChannelSet();
var pollingAMF:AMFChannel = ServerConfig.getChannel("my-amf");
cs.addChannel(pollingAMF);
```

Usually, there is no difference in the result of either of these options, but there are some properties that are set on the channel and used by the client code. When you create your channel using the first option, you should set the following properties depending on the type of channel you are creating: `pollingEnabled` and `pollingIntervalMillis` on AMFChannel and HTTPChannel, and `connectTimeoutSeconds` on Channel. So, when you create a polling AMFChannel on the client, you should set the `pollingEnabled` and `pollingInterval` properties, as the following example shows:

```
...
var cs:ChannelSet = new ChannelSet();
var pollingAMF:AMFChannel = new AMFChannel("my-amf",
    "http://servername:8400/eqa/messagebroker/amfpolling");
pollingAMF.pollingEnabled = true;
pollingAMF.pollingInterval = 8000;
cs.addChannel(pollingAMF);
...
```

The second option, using the `ServerConfig.getChannel()` method, retrieves these properties, so you do not need to set them in your client code. You should use this option when you use a configuration file to define channels with properties.

For components that use clustered destinations, you must define a ChannelSet and set the `clustered` property of the ChannelSet to `true`.

The following example shows MXML code for declaring a RemoteObject component and specifying a ChannelSet and Channel:

```
<RemoteObject id="ro" destination="Dest">
    <mx:channelSet>
        <mx:ChannelSet>
            <mx:channels>
                <mx:AMFChannel id="myAmf"
                    uri="http://myserver:2000/myapp/messagebroker/amf"/>
            </mx:channels>
        </mx:ChannelSet>
    </mx:channelSet>
</RemoteObject>
```

The following example shows equivalent ActionScript code:

```
private function run():void {
    ro = new RemoteObject();
    var cs:ChannelSet = new ChannelSet();
    cs.addChannel(new AMFChannel("myAmf",
        "http://{server.name}:{server.port}/eqa/messagebroker/amf"));
    ro.destination = "RemotingDest_runtime";
    ro.channelSet = cs;
}
```

One slight difference is that when you declare your Channel in MXML, you cannot have the dash (-) character in the value of `id` attribute of the corresponding channel that is defined on the server. For example, you would not be able to use a channel with an id value of `message-dest`. This is not an issue when you use ActionScript instead of MXML.

Chapter 15: The Ajax client library

The Ajax client library for BlazeDS is a set of JavaScript APIs that lets Ajax developers access the messaging capabilities of BlazeDS directly from JavaScript. It lets you use Flex clients and Ajax clients that share data in the same messaging application.

Topics

About the Ajax client library 179
Using the Ajax client library 179
Ajax client library API reference. 183

About the Ajax client library

The Ajax client library for BlazeDS is a JavaScript library that lets Ajax developers access the messaging capabilities of BlazeDS directly from JavaScript. Using the Ajax client library, you can build sophisticated applications that more deeply integrate with back-end data and services. BlazeDS provides integrated publish-subscribe messaging, and real-time data streaming. The Ajax client library lets you use Flex clients and Ajax clients that share data in the same messaging application.

When to use the Ajax client library

The Ajax client library is useful for enhancing any Ajax application with the capabilities of the Messaging Service. Integration with BlazeDS can provide data push and access to data sources outside of strictly XML over HTTP. Because Ajax provides better affordance for HTML-based applications that are not strictly read-only, many Ajax applications are facilitating round tripping of data.

The Ajax client library does not support the Flex RPC service capabilities, which include remote objects, HTTP services, and web services.

Requirements for using the Ajax client library

To use the Ajax client library, you must have the following:

- The Ajax client library, which is included in the following directory of the BlazeDS installation:

  installation_dir\resources\fds-ajax-bridge
- A supported Java application server or servlet container
- Adobe Flex Software Development Kit (SDK) included with BlazeDS
- Adobe Flash Player 9
- Microsoft Internet Explorer, Mozilla Firefox, or Opera with JavaScript enabled

Using the Ajax client library

To use the Ajax client library in an Ajax application, you must include the following JavaScript libraries in script tags on the HTML page:

- FDMSLib.js This file contains the definition of BlazeDS APIs as JavaScript. Include this file in any HTML page that requires BlazeDS. This file requires the FABridge.js file.
- FABridge.js This file provides the Flex Ajax Bridge (FABridge) library, the JavaScript gateway to Flash Player.

You must have the following ActionScript libraries to compile the client SWF file:

- FABridge.as The client Flex Ajax Bridge library. Import FABridge.as into your client application.
- FDMSBase.as The base class of the client application.

To initialize the library, you call a convenience method, `FDMSLib.load()`. This method creates the appropriate HTML to embed Flash Player and load the specified shim SWF file. This SWF file is typically compiled using the FDMSBridge.as application file, but can be any SWF file or MXML application that contains the Flex Ajax Bridge library (FABridge) and references the appropriate BlazeDS classes. After Flash Player loads the shim SWF file that contains the bridge, the specified callback is invoked to notify the application that Data Services are available and ready for use.

You use the following command line (on a single line) to compile the FDMSBridge.swf file:

```
mxmlc
    -verbose-stacktraces
    -o=<path_to_store_compiled_swf>\FDMSBridge.swf
    <path_to_FDMSBridge.as>\FDMSBridge.as
```

This command line assumes that you added mxmlc to the system path, or you run the command from the sdk\bin directory. The source code for FDMSBase.as and FDMSBridge.as is located in the directory *install_root*\resources\fds-ajax-bridge. The source code for FABridge.as is located in the directory *install_root*\resources\fds-ajax-bridge\bridge.

Initializing the Ajax client library

To set up an HTML page for BlazeDS integration, you must include the FDMSLib.js on the page to initialize it. Typically, Flash Player is hidden on the page, because the browser performs all the rendering. The JavaScript code in the following HTML code inserts the hidden Flash Player and initializes the library:

```
<script type="text/javascript" src="include/FABridge.js"></script>
<script type="text/javascript" src="include/FDMSLib.js"></script>
 ...
 FDMSLibrary.load("/ajax/Products/FDMSBridge.swf", fdmsLibraryReady);

<!DOCTYPE html PUBLIC "-//W3 C//DTD XHTML 1.0 Transitional//EN"
http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1"/>
<title>Products</title>
<link href="css/accordion.css" rel="stylesheet" type="text/css"/>
<link href="../css/screen.css" rel="stylesheet" type="text/css"/>
<script type="text/javascript" src="include/FABridge.js"></script>
<script type="text/javascript" src="include/FDMSLib.js"></script>
</head>
<body>
```

```
<script>
    FDMSLibrary.load("/ajax/Products/FDMSBridge.swf", fdmsLibraryReady);
</script>
<noscript><h1>This page requires JavaScript.
Please enable JavaScript in your browser and reload this page.</h1>
</noscript>
<div id="wrap">
<div id="header">
 <h1>Products</h1>
</div>
<div id="content">
 <table id="products">
<caption>Adobe Software</caption>
<tr>
...
</body>
<script language="javascript">
    /*
    * Once the bridge indicates that it is ready, we can proceed to
    * load the data.
    */
    function fdmsLibraryReady()
    {
        ...
    }
</script>
</html>
```

The `FDMSLibrary.load()` convenience method inserts HTML code that puts a hidden Flash Player on the page. This Flash Player instance uses the specified location (for example, ajax/products/FDMSBridge.swf) to load a compiled SWF file. You can specify any SWF file that includes the FABridge and BlazeDS API classes. Using the FDMSBridge.as file provides the smallest SWF file.

To provide a visible SWF file, you must place the appropriate embed tags in the HTML file, and you must set a `bridgeName` `flashvars`, as the following example shows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html;
            charset=iso-8859-1"/>
<title>Products</title>
<link href="css/accordion.css" rel="stylesheet" type="text/css"/>
<link href="../css/screen.css" rel="stylesheet" type="text/css"/>
<script type="text/javascript" src="include/FABridge.js"></script>
<script type="text/javascript" src="include/FDMSLib.js"></script>
</head>
<body>
<script>
    document.write("<object id='flexApp'
    classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000' \
    codebase=
     'http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=9,0,0,0'
    height='400' width='400'>");
    document.write("<param name='flashvars' value='bridgeName=example'/>");
    document.write("<param name='src' value='app.swf'/>");
    document.write("<embed name='flexApp'
    pluginspage='http://www.macromedia.com/go/getflashplayer' \
    src='app.swf' height='400' width='400'
    flashvars='bridgeName=example'/>");
```

```
    document.write("</object>");
    FABridge.addInitializationCallback("example", exampleBridgeReady);
</script>
<noscript><h1>This page requires JavaScript. Please enable JavaScript
in your browser and reload this page.</h1>
</noscript>
<div id="wrap">
<div id="header">
 <h1>Products</h1>
</div>
<div id="content">
 <table id="products">
<caption>Adobe Software</caption>
<tr>
...
</body>
<script>
    function exampleBridgeReady()
    {
        var exampleBridge = FABridge["example"];
        // Do something with this specific bridge.
    }
</script>
 ...
</html>
```

FDMSLibrary methods

The following list describes important methods you call directly on the FDMSLibary object:

- The `FDMSLibrary.addRef(obj)` method increments the ActionScript reference count for the object passed as an argument. This is a wrapper function that calls the FABridge `addRef()` method. You need to do this in order to keep references to objects (for example, events) valid for JavaScript after the scope of their dispatch function has ended.
- The `FDMSLibrary.destroyObject(object)` method directly destroys an ActionScript object by invalidating its reference across the bridge. After this method is called, any subsequent calls to methods or properties tied to the object fail.
- The `FDMSLibrary.load("path", callback)` method inserts HTML code that puts a hidden Flash Player on the page. This Flash Player instance uses the specified location (for example, ajax/products/FDMSBridge.swf) to load a compiled SWF file.
- `The FDMSLibrary.release(obj)` method decrements the ActionScript reference count for the object passed as an argument. This is a wrapper function that in turn calls the FABridge `release()` method. You call this method to decrease the reference count. If it gets to 0, the garbage collector cleans it up at the next pass after the change.

Limitations of the Ajax client library

The Ajax client library depends on the FABridge library. Additionally, two the Ajax client library JavaScript methods do not return the same thing as their ActionScript counterparts. These methods are the `ArrayCollection.refresh()` and `ArrayCollection.removeItemAt()` methods. These return an undefined type instead of a Boolean and an object.

Ajax client library API reference

The Ajax client library is a set of JavaScript APIs that lets Ajax developers access the messaging capabilities of BlazeDS directly from JavaScript.

AsyncResponder

JavaScript proxy to the ActionScript AsyncResponder class. This class has the same API as the ActionScript 3.0 version.

ArrayCollection

JavaScript proxy to the ActionScript ArrayCollection class. This class has the same API as the ActionScript 3.0 version.

Sort

JavaScript proxy to the ActionScript mx.collections.Sort class. This class has the same API as the ActionScript 3.0 version.

SortField

JavaScript proxy to the ActionScript mx.collections.SortField class. This class has the same API as the ActionScript 3.0 version.

Producer

JavaScript proxy to the ActionScript mx.messaging.Producer class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
function libraryReady()
{
    producer = new Producer();
    producer.setDestination("dashboard_chat");
}

function sendChat(user, msg, color)
{
    var message = new AsyncMessage();
    var body = new Object();
    body.userId = user;
    body.msg = msg;
    body.color = color;
    message.setBody(body);
    producer.send(message);
}
...
</script>
```

Consumer

JavaScript proxy to the ActionScript mx.messaging.Consumer class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
function libraryReady()
{
    consumer = new Consumer();
    consumer.setDestination("dashboard_chat");
    consumer.addEventListener("message", messageHandler);
    consumer.subscribe();
}

function messageHandler(event)
{
    body = event.getMessage().getBody();
    alert(body.userId + ":" + body.msg);
}
...
</script>
```

AsyncMessage

Description

JavaScript proxy to the ActionScript mx.messaging.AsyncMessage class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
function libraryReady()
{
    producer = new Producer();
    producer.setDestination("dashboard_chat");
}

function sendChat(user, msg, color)
{
    var message = new AsyncMessage();
    var body = new Object();
    body.userId = user;
    body.msg = msg;
    body.color = color;
    message.setBody(body);
    producer.send(message);
}
...
</script>
```

ChannelSet

JavaScript proxy to the ActionScript mx.messaging.ChannelSet class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
    function libraryReady()
    {
         alert("Library Ready");

        var cs = new ChannelSet();
        cs.addChannel(new AMFChannel("my-polling-amf",
            "http://servername:8100/app/messagebroker/amfpolling"));

        p = new Producer();
        p.setDestination("MyJMSTopic");
        p.setChannelSet(cs);

        c = new Consumer();
        c.setDestination("MyJMSTopic");
        c.addEventListener("message", messageHandler);
        c.setChannelSet(cs);
        c.subscribe();
    }
...
</script>
```

AMFChannel

JavaScript proxy to the ActionScript mx.messaging.channels.AMFChannel class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
    function libraryReady()
    {
         alert("Library Ready");

        var cs = new ChannelSet();
        cs.addChannel(new AMFChannel("my-polling-amf",
            "http://servername:8100/app/messagebroker/amfpolling"));

        p = new Producer();
        p.setDestination("MyJMSTopic");
        p.setChannelSet(cs);

        c = new Consumer();
        c.setDestination("MyJMSTopic");
        c.addEventListener("message", messageHandler);
        c.setChannelSet(cs);
        c.subscribe();
    }
...
</script>
```

HTTPChannel

JavaScript proxy to the ActionScript mx.messaging.channels.HTTPChannel class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
    function libraryReady()
    {
         alert("Library Ready");

        var cs = new ChannelSet();
        cs.addChannel(new HTTPChannel("my-http",
            "http://servername:8100/app/messagebroker/http"));

        p = new Producer();
        p.setDestination("MyJMSTopic");
        p.setChannelSet(cs);

        c = new Consumer();
        c.setDestination("MyJMSTopic");
        c.addEventListener("message", messageHandler);
        c.setChannelSet(cs);
        c.subscribe();
    }
...
</script>
```

SecureAMFChannel

JavaScript proxy to the ActionScript mx.messaging.channels.SecureAMFChannel class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
    function libraryReady()
    {
         alert("Library Ready");

        var cs = new ChannelSet();
        cs.addChannel(new SecureAMFChannel("my-secure-amf",
            "https://servername:8100/app/messagebroker/secureamf"));

        p = new Producer();
        p.setDestination("MyJMSTopic");
        p.setChannelSet(cs);

        c = new Consumer();
        c.setDestination("MyJMSTopic");
        c.addEventListener("message", messageHandler);
        c.setChannelSet(cs);
        c.subscribe();
    }
...
</script>
```

SecureHTTPChannel

JavaScript proxy to the ActionScript mx.messaging.channels.SecureHTTPChannel class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
    function libraryReady()
    {
         alert("Library Ready");

        var cs = new ChannelSet();
        cs.addChannel(new SecureHTTPChannel("my-secure-http",
            "https://servername:8100/app/messagebroker/securehttp"));
        cs.addChannel(new AMFChannel("my-polling-amf",
            "/app/messagebroker/amfpolling"));

        p = new Producer();
        p.setDestination("MyJMSTopic");
        p.setChannelSet(cs);

        c = new Consumer();
        c.setDestination("MyJMSTopic");
        c.addEventListener("message", messageHandler);
        c.setChannelSet(cs);
        c.subscribe();
    }
...
</script>
```

FDMSLib JavaScript

The bridge provides a gateway between the JavaScript virtual machine and the ActionScript virtual machine for making calls to objects hosted by Flash Player. This bridge is designed specifically to handle various aspects of asynchronous calls that the Messaging APIs require.

Example

```
/**
 *Once the FDMSBridge SWF file is loaded, this method is called.
*/
function initialize_FDMSBridge(typeData)
{
    for (var i=0; i<typeData.length; i++)
    FDMSBridge.addTypeInfo(typeData[i]);

    // setup ArrayCollection etc.
    ArrayCollection.prototype =
        FDMSBridge.getTypeFromName("mx.collections::ArrayCollection");
    ...
    // now callback the page specific code indicating that initialization is
        complete
    FDMSBridge.initialized();
}
```

Chapter 16: Extending applications with factories

BlazeDS provides a factory mechanism that lets you plug in your own component creation and maintenance system to allow it to integrate with systems like EJB and Spring, which store components in their own namespace. You provide a class that implements the flex.messaging.FlexFactory interface. This class is used to create a FactoryInstance that corresponds to a component configured for a specific destination.

Topics

The factory mechanism 188

The factory mechanism

Remoting Service destinations use Java classes that you write to integrate with Flex clients. By default, BlazeDS creates these instances. If they are application-scoped components, they are stored in the ServletContext attribute space using the destination's name as the attribute name. If you use session-scoped components, the components are stored in the FlexSession, also using the destination name as the attribute. If you specify an `attribute-id` element in the destination, you can control which attribute the component is stored in; this lets more than one destination share the same instance.

The following examples shows a destination definition that contains an `attribute-id` element:

```
<destination id="WeatherService">
    <properties>
        <source>weather.WeatherService</source>
        <scope>application</scope>
        <attribute-id>MyWeatherService</attribute-id>
    </properties>
</destination>
```

In this example, BlazeDS creates an instance of the class weather.WeatherService and stores it in the ServletContext object's set of attributes with the name MyWeatherService. If you define a different destination with the same `attribute-id` value and the same Java class, BlazeDS uses the same component instance.

BlazeDS provides a factory mechanism that lets you plug in your own component creation and maintenance system to BlazeDS so it integrates with systems like EJB and Spring, which store components in their own namespace. You provide a class that implements the flex.messaging.FlexFactory interface. You use this class to create a FactoryInstance that corresponds to a component configured for a specific destination. Then the component uses the FactoryInstance to look up the specific component to use for a given request. The FlexFactory implementation can access configuration attributes from a BlazeDS configuration file and also can access FlexSession and ServletContext objects. For more information, see the documentation for the FlexFactory class in the public BlazeDS Javadoc documentation.

After you implement a FlexFactory class, you can configure it to be available to destinations by placing a `factory` element in the `factories` element in the services-config.xml file, as the following example shows. A single FlexFactory instance is created for each BlazeDS web application. This instance can have its own configuration properties, although in this example, there are no required properties to configure the factory.

```
<factories>
    <factory id="spring" class="flex.samples.factories.SpringFactory"/>
</factories>
```

BlazeDS creates one FlexFactory instance for each `factory` element that you specify. BlazeDS uses this one global FlexFactory implementation for each destination that specifies that factory by its ID; the ID is identified by using a `factory` element in the `properties` section of the destination definition. For example, your remoting-config.xml file could have a destination similar to the following one:

```
<destination id="WeatherService">
    <properties>
        <factory>spring</factory>
        <source>weatherBean</source>
    </properties>
</destination>
```

When the `factory` element is encountered at start up, BlazeDS calls the `FlexFactory.createFactoryInstance()` method. That method gets the `source` value and any other attributes it expects in the configuration. Any attributes that you access from the ConfigMap parameter are marked as expected and do not cause a configuration error, so you can extend the default BlazeDS configuration in this manner. When BlazeDS requires an instance of the component for this destination, it calls the `FactoryInstance.lookup()` method to retrieve the individual instance for the destination.

Optionally, factory instances can take additional attributes. There are two places you can do this. When you define the factory initially, you can provide extra attributes as part of the factory definition. When you define an instance of that factory, you can also add your own attributes to the destination definition to be used in creating the factory instance.

The boldface text in the following example shows an attribute added to the factory definition:

```
<factories>
    <factory id="myFactoryId" class="myPackage.MyFlexFactory">
        <properties>
            <myfactoryattributename>
                myfactoryattributevalue
            </myfactoryattributename>
        </properties>
    </factory>
</factories>
```

You could use this type of configuration when you are integrating with the Spring Framework Java application framework to provide the Spring factory with a default path for initializing the Spring context that you use to look up all components for that factory. In the class that implements FlexFactory, you would include a call to retrieve the values of the `myfactoryattributename` from the `configMap` parameter to the `initialize()` method in the FlexFactory interface, as the following example shows:

```
public void initialize(String id, ConfigMap configMap){
    System.out.println("**** MyFactory initialized with: " +
        configMap.getPropertyAsString("myfactoryattributename", "not set"));
}
```

The `initialize()` method in the previous example retrieves a string value where the first parameter is the name of the attribute, and the second parameter is the default value to use if that value is not set. For more information about the various calls to retrieve properties in the config map, see the documentation for the flex.messaging.config.ConfigMap class in the public BlazeDS Javadoc documentation. Each factory instance can add configuration attributes that are used when that factory instance is defined, as the following example shows:

```
<destination id="myDestination">
    <properties>
        <source>mypackage.MyRemoteClass</source>
        <factory>myFactoryId</factory>
        <myfactoryinstanceattribute>
            myfoobar2value
        </myfactoryinstanceattribute>
    </properties>
</destination>
```

In the `createFactoryInstance()` method as part of the FlexFactory implementation, you access the attribute for that instance of the factory, as the following example shows:

```
public FactoryInstance createFactoryInstance(String id, ConfigMap properties) {
    System.out.println("**** MyFactoryInstance instance initialized with
myfactoryinstanceattribute=" +
    properties.getPropertyAsString("myfactoryinstanceattribute", "not set"));
….
}
```

The following example shows the source code of a Spring factory class:

```
package flex.samples.factories;

import org.springframework.context.ApplicationContext;
import org.springframework.web.context.support.WebApplicationContextUtils;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.NoSuchBeanDefinitionException;

import flex.messaging.FactoryInstance;
import flex.messaging.FlexFactory;
import flex.messaging.config.ConfigMap;
import flex.messaging.services.ServiceException;

/**
 * The FactoryFactory interface is implemented by factory components that provide
 * instances to the data services messaging framework.  To configure data services
 * to use this factory, add the following lines to your services-config.xml
 * file (located in the WEB-INF/flex directory of your web application).
 *
<pre>
 *  <factories>
 *     <factory id="spring" class="flex.samples.factories.SpringFactory" />
 *  </factories>
</pre>
 *
 * You also must configure the web application to use spring and must copy the spring.jar
 * file into your WEB-INF/lib directory.  To configure your app server to use Spring,
 * you add the following lines to your WEB-INF/web.xml file:
 *<pre>
 *   <context-param>
 *        <param-name>contextConfigLocation</param-name>
 *        <param-value>/WEB-INF/applicationContext.xml</param-value>
 *   </context-param>
 *
 *   <listener>
 *     <listener-class>
*     org.springframework.web.context.ContextLoaderListener</listener-class>
 *   </listener>
 * </pre>
 * Then you put your Spring bean configuration in WEB-INF/applicationContext.xml (as per the
 * line above).  For example:
```

```
 * <pre>
 *  <?xml version="1.0" encoding="UTF-8"?>
 *  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 * "http://www.springframework.org/dtd/spring-beans.dtd">
 *
 *  <beans>
 *    <bean name="weatherBean" class="dev.weather.WeatherService" singleton="true"/>
 *  </beans>
 *  </pre>
 * Now you are ready to define a Remoting Service destination  that maps to this existing
service.
 * To do this you'd add this to your WEB-INF/flex/remoting-config.xml:
 *
 *<pre>
 *  <destination id="WeatherService">
 *      <properties>
 *          <factory>spring</factory>
 *          <source>weatherBean</source>
 *      </properties>
 *  </destination>
 *</pre>
 */
public class SpringFactory implements FlexFactory
{
    private static final String SOURCE = "source";

    /**
     * This method can be used to initialize the factory itself.
     * It is called with configuration
     * parameters from the factory tag which defines the id of the factory.
     */
    public void initialize(String id, ConfigMap configMap) {}

    /**
     * This method is called when we initialize the definition of an instance
     * which will be looked up by this factory.  It should validate that
     * the properties supplied are valid to define an instance.
     * Any valid properties used for this configuration must be accessed to
     * avoid warnings about unused configuration elements.  If your factory
     * is only used for application scoped components, this method can simply
     * return a factory instance which delegates the creation of the component
     * to the FactoryInstance's lookup method.
     */
    public FactoryInstance createFactoryInstance(String id, ConfigMap properties)
    {
        SpringFactoryInstance instance = new SpringFactoryInstance(this, id, properties);
        instance.setSource(properties.getPropertyAsString(SOURCE, instance.getId()));
        return instance;
    } // end method createFactoryInstance()

    /**
     * Returns the instance specified by the source
     * and properties arguments.  For the factory, this may mean
     * constructing a new instance, optionally registering it in some other
     * name space such as the session or JNDI, and then returning it
     * or it may mean creating a new instance and returning it.
     * This method is called for each request to operate on the
     * given item by the system so it should be relatively efficient.
     * <p>
     * If your factory does not support the scope property, it
     * report an error if scope is supplied in the properties
```

```
     * for this instance.
     * </p>
     */
    public Object lookup(FactoryInstance inst)
    {
        SpringFactoryInstance factoryInstance = (SpringFactoryInstance) inst;
        return factoryInstance.lookup();
    }

    static class SpringFactoryInstance extends FactoryInstance
    {
        SpringFactoryInstance(SpringFactory factory, String id, ConfigMap properties)
        {
            super(factory, id, properties);
        }

        public String toString()
        {
            return "SpringFactory instance for id=" + getId() + " source=" + getSource() +
                " scope=" + getScope();
        }

        public Object lookup()
        {
            ApplicationContext appContext =
WebApplicationContextUtils.getWebApplicationContext(flex.messaging.FlexContext.getServletC
onfig().getServletContext());
            String beanName = getSource();

            try
            {
                return appContext.getBean(beanName);
            }
            catch (NoSuchBeanDefinitionException nexc)
            {
                ServiceException e = new ServiceException();
                String msg = "Spring service named '" + beanName + "' does not exist.";
                e.setMessage(msg);
                e.setRootCause(nexc);
                e.setDetails(msg);
                e.setCode("Server.Processing");
                throw e;
            }
            catch (BeansException bexc)
            {
                ServiceException e = new ServiceException();
                String msg = "Unable to create Spring service named '" + beanName + "' ";
                e.setMessage(msg);
                e.setRootCause(bexc);
                e.setDetails(msg);
                e.setCode("Server.Processing");
                throw e;
            }
        }

    }

}
```

Chapter 17: Message delivery with adaptive polling

The adaptive polling capability lets you write custom logic to control how messages are queued for delivery to Adobe® Flex™ client applications on a per-client basis.

Topics

Adaptive polling 193
Using a custom queue processor. 194

Adaptive polling

The adaptive polling capability provides a per-client outbound message queue API. You can use this API in custom Java code to manage per-client messaging quality of service based on your criteria for determining and driving quality of service. To use this capability, you create a custom queue processor class and register it in a channel definition in the services-config.xml file.

Using a custom queue processor, you can do such things as conflate messages (combine a new message with an existing message in the queue), order messages according to arbitrary priority rules, filter out messages based on arbitrary rules, and manage flushing (sending) messages to the network layer explicitly. You have full control over the delivery rate of messages to clients on a per-client basis, and the ability to define the order and contents of delivered messages on a per-client basis.

Instances of the flex.messaging.client.FlexClient class on the server maintain the state of each client application. You provide adaptive polling for individual client instances by extending the flex.messaging.client.FlexClientOutboundQueueProcessor class. This class provides an API to manage adding messages to an outbound queue and flushing messages in an outbound queue to the network.

When a message arrives at a destination on the server and it matches a specific client subscription, represented by an instance of the flex.messaging.MessageClient class, the message is routed to an instance of FlexClient where it is added to the queue for the channel/endpoint that the MessageClient subscription was created over. An instance of the flex.messaging.MessageClient class represents a specific client subscription. When a message arrives at a destination on the server, and it matches a client subscription, the message is routed to an instance of the FlexClient class. Then the message is added to the queue for the channel/endpoint that the MessageClient subscription was created over.

The default flush behavior depends on the channel/endpoint that you use. If you use polling channels, a flush is attempted when a poll request is received. If you use direct push channels, a flush is attempted after each new message is added to the outbound queue. You can write Java code to return a flex.messaging.FlushResult instance that contains the list of messages to hand off to the network layer, which are sent to the client, and specify an optional wait time to delay the next flush.

For polling channels, a next flush wait time results in the client waiting the specified length of time before issuing its next poll request. For direct push channels, until the wait time is over, the addition of new messages to the outbound queue does not trigger an immediate invocation of flush; when the wait time is up, a delayed flush is invoked automatically. This lets you write code to drive adaptive client polling and adaptive writes over direct connections. You could use this functionality to shed load (for example, to cause clients to poll a loaded server less frequently), or to provide tiered message delivery rates on a per-client basis to optimize bandwidth usage (for example, gold customers could get messages immediately, while bronze customers only receive messages once every 5 minutes).

If an outbound queue processor must adjust the rate of outbound message delivery, it can record its own internal statistics to do so. This could include total number of messages delivered, rate of delivery over time, and so forth. Queue processors that only perform conflation or filtering do not require the overhead of collecting statistics.

The FlexClientOutboundQueueProcessor, FlexClient, MessageClient, and FlushResult classes are documented in the public BlazeDS Javadoc API documentation.

Using a custom queue processor

To use a custom queue processor, you must create a queue processor class, compile it, add it to the class path, and then configure it. The examples in this topic are part of the adaptive polling sample application included in the BlazeDS samples web application.

Creating a custom queue processor

To create a custom queue processor class, you must extend the FlexClientOutboundQueueProcessor class. This class is documented in the public BlazeDS Javadoc API documentation. It provides the methods described in the following table:

Method	Description
`initialize(ConfigMap properties)`	Initializes a new queue processor instance after it is associated with its corresponding FlexClient, but before any messages are enqueued.
`add(List queue, Message message)`	Adds the message to the queue at the desired index, conflates it with an existing message, or ignores it entirely.
`FlushResult flush(List queue)`	Removes messages from the queue to be flushed out over the network. Can contain an optional wait time before the next flush is invoked.
`FlushResult flush(MessageClient messageClient, List queue)`	Removes messages from the queue for a specific MessageClient subscription. Can contain an optional wait time before the next flush is invoked.

The following example shows the source code for a custom queue processor class that sets the delay time between flushes in its `flush(List outboundQueue)` method.

```
package flex.samples.qos;

import java.util.ArrayList;
import java.util.List;

import flex.messaging.client.FlexClient;
import flex.messaging.client.FlexClientOutboundQueueProcessor;
import flex.messaging.client.FlushResult;
import flex.messaging.config.ConfigMap;
import flex.messaging.MessageClient;
```

```
/**
* Per client queue processor that applies custom quality of
* service parameters (in this case: delay).
* Custom quality of services parameters are read from the client FlexClient
* instance.
* In this sample, these parameters are set in the FlexClient instance by
* the client application using the flex.samples.qos.FlexClientConfigService
* remote object class.
* This class is used in the per-client-qos-polling-amf channel definition.
*
*/
public class CustomDelayQueueProcessor extends FlexClientOutboundQueueProcessor
{
/**
* Used to store the last time this queue was flushed.
* Starts off with an initial value of the construct time for the
* instance.
*/
private long lastFlushTime = System.currentTimeMillis();

/**
* Driven by configuration, this is the configurable delay time between
* flushes.
*/
private int delayTimeBetweenFlushes;

public CustomDelayQueueProcessor()
{}

/**
* Sets up the default delay time between flushes. This default is used
* if a client-specific
* value has not been set in the FlexClient instance.
*
* @param properties A ConfigMap containing any custom initialization
* properties.
*/
public void initialize(ConfigMap properties)
{
    delayTimeBetweenFlushes = properties.getPropertyAsInt("flush-delay",-1);
    if (delayTimeBetweenFlushes < 0)
        throw new RuntimeException("Flush delay time forDelayedDeliveryQueueProcessor
            must be a positive value.");
}

/**
* This flush implementation delays flushing messages from the queue
* until 3 seconds have passed since the last flush.
*
* @param outboundQueue The queue of outbound messages.
* @return An object containing the messages that have been removed
* from the outbound queue
* to be written to the network and a wait time for the next flush
* of the outbound queue
* that is the default for the underlying Channel/Endpoint.
*/
public FlushResult flush(List outboundQueue)
{
    int delay = delayTimeBetweenFlushes;
    // Read custom delay from client's FlexClient instance
    System.out.println("***"+getFlexClient());
```

```
    FlexClient flexClient = getFlexClient();
    if (flexClient != null)
    {
        Object obj = flexClient.getAttribute("market-data-delay");
        if (obj != null)
        {
            try {
                delay = Integer.parseInt((String) obj);
            } catch (Exception e) {
            }
        }
    }

    long currentTime = System.currentTimeMillis();
    System.out.println("Flush?" + (currentTime - lastFlushTime) + "<" +delay);
    if ((currentTime - lastFlushTime) < delay)
    {
        // Delaying flush. No messages will be returned at this point
        FlushResult flushResult = new FlushResult();
        // Don't return any messages to flush.
        // And request that the next flush doesn't occur until 3 secondssince the previous.
        flushResult.setNextFlushWaitTimeMillis((int)(delay -
            (currentTime - lastFlushTime)));
        return flushResult;
    }
    else // OK to flush.
    {
        // Flushing. All queued messages will now be returned
        lastFlushTime = currentTime;
        FlushResult flushResult = new FlushResult();
        flushResult.setNextFlushWaitTimeMillis(delay);
        flushResult.setMessages(new ArrayList(outboundQueue));
        outboundQueue.clear();
        return flushResult;
    }
}

public FlushResult flush(MessageClient client, List outboundQueue) {
    return super.flush(client, outboundQueue);
}

}
```

A Flex client application calls the following remote object to set the delay time between flushes on `CustomDelayQueueProcessor`:

```
package flex.samples.qos;

import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;

import flex.messaging.FlexContext;
import flex.messaging.client.FlexClient;

public class FlexClientConfigService
{

    public void setAttribute(String name, Object value)
    {
        FlexClient flexClient = FlexContext.getFlexClient();
        flexClient.setAttribute(name, value);
```

```
    }

    public List getAttributes()
    {
        FlexClient flexClient = FlexContext.getFlexClient();
        List attributes = new ArrayList();
        Enumeration attrNames = flexClient.getAttributeNames();
        while (attrNames.hasMoreElements())
        {
            String attrName = (String) attrNames.nextElement();
            attributes.add(new Attribute(attrName, flexClient.getAttribute(attrName)));
        }

        return attributes;
    }

    public class Attribute {

        private String name;
        private Object value;

        public Attribute(String name, Object value) {
            this.name = name;
            this.value = value;
        }
        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
        public Object getValue() {
            return value;
        }
        public void setValue(Object value) {
            this.value = value;
        }
}
```

Configuring a custom queue processor

You register custom implementations of the FlexClientOutboundQueueProcessor class on a per-channel/endpoint basis. To register a custom implementation, you configure a `flex-client-outbound-queue` property in a channel definition in the services-config.xml file, as the following example shows:

```
<channel-definition id="per-client-qos-polling-amf"
class="mx.messaging.channels.AMFChannel">
    <endpoint url="http://localhost:8400/blazeds/messagebroker/qosamfpolling"
        class="flex.messaging.endpoints.AMFEndpoint"/>
    <properties>
        <polling-enabled>true</polling-enabled>
        <polling-interval-millis>500</polling-interval-millis>
        <flex-client-outbound-queue-processor
            class="flex.samples.qos.CustomDelayQueueProcessor">
            <properties>
                <flush-delay>5000</flush-delay>
            </properties>
        </flex-client-outbound-queue-processor>
    </properties>
</channel-definition>
```

This example shows how you can also specify arbitrary properties to be passed into the `initialize()` method of your queue processor class after it has been constructed and has been associated with its corresponding FlexClient instance, but before any messages are enqueued. In this case, the `flush-delay` value is passed into the `initialize()` method. This is the default value that is used if a client does not specify a flush delay value.

You then specify the channel in your message destination, as the bold text in the following example shows:

```
<destination id="market-data-feed">
    <properties>
        <network>
            <subscription-timeout-minutes>0</subscription-timeout-minutes>
        </network>
        <server>
            <max-cache-size>1000</max-cache-size>
            <message-time-to-live>0</message-time-to-live>
            <durable>true</durable>
            <allow-subtopics>true</allow-subtopics>
            <subtopic-separator>.</subtopic-separator>
        </server>
    </properties>
    <channels>
        <channel ref="per-client-qos-rtmp"/>
    </channels>
</destination>
```

Chapter 18: Measuring message processing performance

As part of preparing your application for final deployment, you can test its performance to look for ways to optimize it. One place to examine performance is in the message processing part of the application. To help you gather this performance information, enable the gathering of message timing and sizing data.

Topics

About measuring message processing performance 199
Measuring message processing performance 204

About measuring message processing performance

The mechanism for measuring message processing performance is disabled by default. When enabled, information regarding message size, server processing time, and network travel time is available to the client that pushed a message to the server, to a client that received a pushed message from the server, or to a client that received an acknowledge message from the server in response a pushed message. A subset of this information is also available for access on the server.

You can use this mechanism across all channel types, including polling and streaming channels, that communicate with the BlazeDS server. However, this mechanism does not work when you make a direct connection to an external server by setting the `useProxy` property to `false` for the HTTPService and WebService tags because it bypasses the BlazeDS Proxy Server.

The MessagePerformanceUtils class defines the available message processing metrics. When a consumer receives a message, or a producer receives an acknowledge message, the consumer or producer extracts the metrics into an instance of the MessagePerformanceUtils class, and then accesses the metrics as properties of that class. For a complete list of the available metrics, see “Available message processing metrics” on page 202.

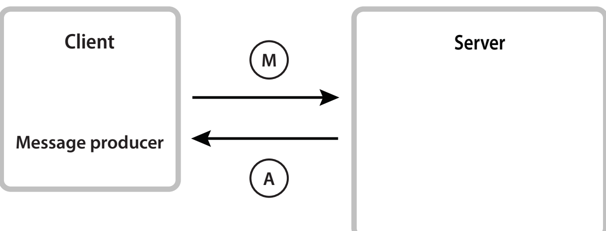
Measuring performance for different channel types

The types of metrics that are available and their calculations, depend on the channel configuration over which a message is sent from or received by the client.

Producer-acknowledge scenario

In the producer-acknowledge scenario, a producer sends a message to a server over a specific channel. The server then sends an acknowledge message back to the producer.

The following image shows the producer-acknowledge scenario:



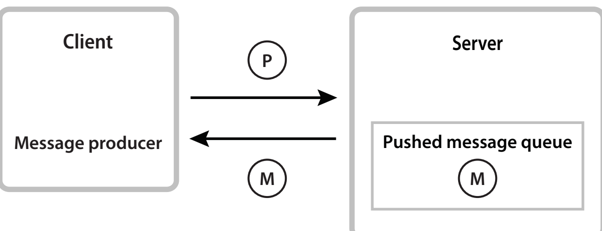
M. Message sent to server. A. Acknowledge message.

If you enable the gathering of message processing metrics, the producer adds information to the message before sending it to the server, such as the send time and message size. The server copies the information from the message to the acknowledge message. Then the server adds additional information to the acknowledge message, such as the response message size and server processing time. When the producer receives the acknowledge message, it uses all of the information in the message to calculate the metrics defined by the MessagePerformanceUtils class.

Message-polling scenario

In a message-polling scenario, a consumer polls a message channel to determine if a message is available on the server. On receiving the polling message, the server pushes any available message to the consumer.

The following image shows the polling scenario:



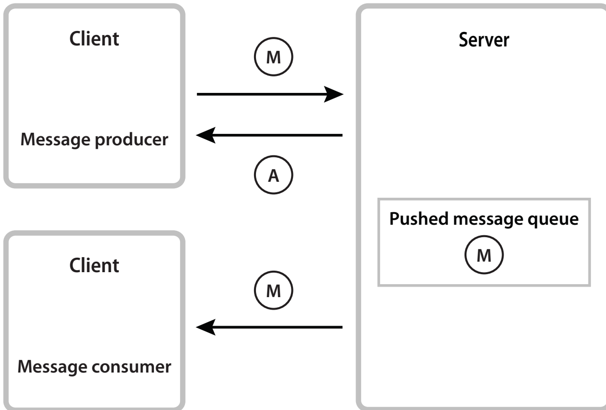
P. Polling message sent. M. Message pushed to client from server.

If you enable the gathering of message processing metrics in this scenario, the consumer obtains performance metrics about the poll-response transaction, such as the response message size and server processing time. The metrics also include information about the message returned by the server. This information lets the consumer determine how long the message was waiting before it was pushed. However, the metric information does not identify the client that originally pushed the message onto the server.

Message-streaming scenario

In the streaming scenario, the server pushes a message to a consumer when a message is available; the consumer itself does not initiate the transaction.

The following image shows this scenario:



M. Message sent to server, and then pushed to client. A. Acknowledge message.

In this scenario, the message producer pushes a message, and then receives an acknowledge message. The producer can obtain metric information as described in "Producer-acknowledge scenario" on page 199.

When the server pushes the message to the consumer, the message contains information from the original message from the producer, and the metric information that the server added. The consumer can then examine the metric data, including the time from when the producer pushed the message until the consumer received it.

Measuring message processing performance for streaming channels

For streaming channels, a pushed message is sent to a consumer without the client first sending a polling message. In this case, the following metrics are not available to the consumer for the pushed message, but instead are set to 0:

- `networkRTT`
- `serverPollDelay`
- `totalTime`

Available message processing metrics

The following table lists the available message processing metrics defined by the MessagePerformanceUtils class:

Property	Description
`clientReceiveTime`	The number of milliseconds since the start of the UNIX epoch, January 1, 1970, 00:00:00 GMT, to when the client received a response message from the server.
`messageSize`	The size of the original client message, in bytes, as measured during deserialization by the server endpoint.
`networkRTT`	The duration, in milliseconds, from when a client sent a message to the server until it received a response, excluding the server processing time. This value is calculated as `totalTime - serverProcessingTime`. If a pushed message is using a streaming channel, the metric is meaningless because the client does not initiate the pushed message; the server sends a message to the client whenever a message is available. Therefore, for a message pushed over a streaming channel, this value is 0. However, for an acknowledge message sent over a streaming channel, the metric contains a valid number.
`originatingMessageSentTime`	The timestamp, in milliseconds since the start of the UNIX epoch on January 1, 1970, 00:00:00 GMT, to when the client that caused a push message sent its message. Only populated for a pushed message, but not for an acknowledge message.
`originatingMessageSize`	Size, in bytes, of the message that originally caused this pushed message. Only populated for a pushed message, but not for an acknowledge message.
`pushedMessageFlag`	Contains `true` if the message was pushed to the client but is not a response to a message that originated on the client. For example, when the client polls the server for a message, `pushedMessageFlag` is `false`. When you are using a streaming channel, `pushedMessageFlag` is `true`. For an acknowledge message, `pushedMessageFlag` is `false`.
`pushOneWayTime`	Time, in milliseconds, from when the server pushed the message until the client received it. **Note:** This value is only relevant if the server and receiving client have synchronized clocks. Only populated for a pushed message, but not for an acknowledge message.
`responseMessageSize`	The size, in bytes, of the response message sent to the client by the server as measured during serialization at the server endpoint.
`serverAdapterExternalTime`	Time, in milliseconds, spent in a module invoked from the adapter associated with the destination for this message, before either the response to the message was ready or the message had been prepared to be pushed to the receiving client. This value corresponds to the message processing time on the server.
`serverAdapterTime`	Processing time, in milliseconds, of the message by the adapter associated with the destination before the response to the message was ready or the message was prepared to be pushed to the receiving client. The processing time corresponds to the time that your code on the server processed the message, not when Blaze DS processed the message.
`serverNonAdapterTime`	Server processing time spent outside the adapter associated with the destination of this message. Calculated as `serverProcessingTime - serverAdapterTime`.
`serverPollDelay`	Time, in milliseconds, that this message sat on the server after it was ready to be pushed to the client but before it was picked up by a poll request. For a streaming channel, this value is always 0.
`serverPrePushTime`	Time, in milliseconds, between the server receiving the client message and the server beginning to push the message out to other clients.

Property	Description
`serverProcessingTime`	Time, in milliseconds, between server receiving the client message and either the time the server responded to the received message or has the pushed message ready to be sent to a receiving client. For example, in the producer-acknowledge scenario, this value is the time from when the server receives the message and sends the acknowledge message back to the producer. In a polling scenario, it is the time between the arrival of the polling message from the consumer and any message returned in response to the poll.
`serverSendTime`	The number of milliseconds since the start of the UNIX epoch, January 1, 1970, 00:00:00 GMT, to when the server sent a response message back to the client.
`totalPushTime`	Time, in milliseconds, from when the originating client sent a message and the time that the receiving client received the pushed message. **Note:** This value is only relevant if the two clients have synchronized clocks. Only populated for a pushed message, but not for an acknowledge message.
`totalTime`	Time, in milliseconds, between this client sending a message and receiving a response from the server. This property contains 0 for a streaming channel.

Considerations when measuring message processing performance

The mechanism that measures message processing performance attempts to minimize the overhead required to collect information so that all timing information is as accurate as possible. However, take into account the following considerations when you use this mechanism.

Synchronize the clocks on different computers

The metrics defined by the MessagePerformanceUtils class include the `totalPushTime`. The `totalPushTime` is a measure of the time from when the originating message producer sent a message until a consumer receives the message. This value is determined from the timestamp added to the message when the producer sends the message, and the timestamp added to the message when the consumer receives the message. However, to calculate a valid value for the `totalPushTime` metric, the clocks on the message-producing computer and on the message-consuming computer must be synchronized.

Another metric, `pushOneWayTime`, contains the time from when the server pushed the message until the consumer received it. This value is determined from the timestamp added to the message when the server sends the message, and the timestamp added to the message when the consumer receives the message. To calculate a valid value for the `pushOneWayTime` metric, the clocks on the message consuming computer and on the server must be synchronized.

One option is to perform your testing in a lab environment where you can ensure that the clocks on all computers are synchronized. For the `totalPushTime` metric, you can ensure that the clocks for the producer and consumer applications are synchronized by running the applications on the same computer. Or, for the `pushOneWayTime` metric, you can run the consumer application and server on the same computer.

Perform different tests for message timing and sizing

The mechanism for measuring message processing performance lets you enable the tracking of timing information, of sizing information, or both. The gathering of timing-only metrics is minimally intrusive to your overall application performance. The gathering of sizing metrics involves more overhead time than gathering timing information.

Therefore, you can run your tests twice: once for gathering timing information and once for gathering sizing information. In this way, the timing-only test can eliminate any delays caused by calculating message size. You can then combine the information from the two tests to determine your final results.

Measuring message processing performance

The mechanism for measuring message processing performance is disabled by default. When you enable it, you can use the MessagePerformanceUtils class to access the metrics from a message received by a client.

Enabling message processing metrics

You use two parameters in a channel definition to enable message processing metrics:

- `<record-message-times>`
- `<record-message-sizes>`

Set these parameters to `true` or `false`; the default value is `false`. You can set the parameters to different values to capture only one type of metric. For example, the following channel definition specifies to capture message timing information, but not message sizing information:

```
<channel-definition id="my-streaming-amf"
    class="mx.messaging.channels.StreamingAMFChannel">
    <endpoint
        url="http://{server.name}:{server.port}/{context.root}/messagebroker/streamingamf"
        class="flex.messaging.endpoints.StreamingAMFEndpoint"/>
    <properties>
        <record-message-times>true</record-message-times>
        <record-message-sizes>false</record-message-sizes>
    </properties>
</channel-definition>
```

Using the MessagePerformanceUtils class

The MessagePerformanceUtils class is a client-side class that you use to access the message processing metrics. You create an instance of the MessagePerformanceUtils class from a message pushed to the client by the server or from an acknowledge message.

The following example shows a message producer that uses the acknowledge message to display in a TextArea control the metrics for a message pushed to the server:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.messaging.messages.AsyncMessage;
            import mx.messaging.messages.IMessage;
            import mx.messaging.events.MessageEvent;
            import mx.messaging.messages.MessagePerformanceUtils;

            // Event handler to send the message to the server.
            private function send():void
            {
                var message:IMessage = new AsyncMessage();
                message.body.chatMessage = msg.text;
```

```
                producer.send(message);
                msg.text = "";
            }

            // Event handler to write metrics to the TextArea control.
            private function ackHandler(event:MessageEvent):void {
                var mpiutil:MessagePerformanceUtils =
                    new MessagePerformanceUtils(event.message);
                myTAAck.text = "totalTime = " + String(mpiutil.totalTime);
                myTAAck.text = myTAAck.text + "\n" + "messageSize= " +
                    String(mpiutil.messageSize);
            }

        ]]>
    </mx:Script>

    <mx:Producer id="producer" destination="chat" acknowledge="ackHandler(event)"/>

    <mx:Label text="Acknowledge metrics"/>
    <mx:TextArea id="myTAAck" width="100%" height="20%"/>

    <mx:Panel title="Chat" width="100%" height="100%">
        <mx:TextArea id="log" width="100%" height="100%"/>
        <mx:ControlBar>
             <mx:TextInput id="msg" width="100%" enter="send()"/>
             <mx:Button label="Send" click="send()"/>
        </mx:ControlBar>
    </mx:Panel>

</mx:Application>
```

In this example, you write an event handler for the `acknowledge` event to display the metrics. The event handler extracts the metric information from the acknowledge message, and then displays the `MessagePerformanceUtils.totalTime` and `MessagePerformanceUtils.messageSize` metrics in a TextArea control.

You can also use the `MessagePerformanceUtils.prettyPrint()` method to display the metrics. The `prettyPrint()` method returns a formatted String that contains nonzero and non-null metrics. The following example modifies the event handler for the previous example to use the `prettyPrint()` method:

```
// Event handler to write metrics to the TextArea control.
private function ackHandler(event:MessageEvent):void {
    var mpiutil:MessagePerformanceUtils = new MessagePerformanceUtils(event.message);
    myTAAck.text = mpiutil.prettyPrint();
}
```

The following example shows the output from the `prettyPrint()` method that appears in the TextArea control:

```
Original message size(B): 509
Response message size(B): 562
Total time (s): 0.016
Network Roundtrip time (s): 0.016
```

A message consumer can write an event handler for the `message` event to display metrics, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="consumer.subscribe();">

    <mx:Script>
        <![CDATA[
```

```
            import mx.messaging.messages.AsyncMessage;
            import mx.messaging.messages.IMessage;
            import mx.messaging.events.MessageEvent;
            import mx.messaging.messages.MessagePerformanceUtils;

            // Event handler to send the message to the server.
            private function send():void
            {
                var message:IMessage = new AsyncMessage();
                message.body.chatMessage = msg.text;
                producer.send(message);
                msg.text = "";
            }

            // Event handler to write metrics to the TextArea control.
            private function ackHandler(event:MessageEvent):void {
                var mpiutil:MessagePerformanceUtils =
                    new MessagePerformanceUtils(event.message);
                myTAAck.text = mpiutil.prettyPrint();
            }

          // Event handler to write metrics to the TextArea control for the message consumer.
            private function messageHandler(event:MessageEvent):void {
                var mpiutil:MessagePerformanceUtils =
                    new MessagePerformanceUtils(event.message);
                myTAMess.text = mpiutil.prettyPrint();

            }
        ]]>
    </mx:Script>

    <mx:Producer id="producer" destination="chat" acknowledge="ackHandler(event)"/>
    <mx:Consumer id="consumer" destination="chat" message="messageHandler(event)"/>

    <mx:Label text="ack metrics"/>
    <mx:TextArea id="myTAAck" width="100%" height="20%" text="ack"/>

    <mx:Label text="receive metrics"/>
    <mx:TextArea id="myTAMess" width="100%" height="20%" text="rec"/>

    <mx:Panel title="Chat" width="100%" height="100%">
        <mx:TextArea id="log" width="100%" height="100%"/>
        <mx:ControlBar>
             <mx:TextInput id="msg" width="100%" enter="send()"/>
             <mx:Button label="Send" click="send()"/>
        </mx:ControlBar>
    </mx:Panel>

</mx:Application>
```

In this example, you use the `prettyPrint()` method to write the metrics for the received message to a TextArea control. The following example shows this output:

```
Response message size(B): 560
PUSHED MESSAGE INFORMATION:
Total push time (s): 0.016
Push one way time (s): 0.016
Originating Message size (B): 509
```

You can gather metrics for HTTPService and WebService tags when they use the Proxy Service, as defined by setting the `useProxy` property to `true` for the HTTPService and WebService tags. The following example gathers metrics for an HTTPService tag:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" backgroundColor="#FFFFFF">

    <mx:Script>
        <![CDATA[

            import mx.messaging.events.MessageEvent;
            import mx.messaging.messages.MessagePerformanceUtils;

          // Event handler to write metrics to the TextArea control for the message consumer.
            private function messageHandler(event:MessageEvent):void {
                var mpiutil:MessagePerformanceUtils =
                    new MessagePerformanceUtils(event.message);
                myTAMess.text = mpiutil.prettyPrint();
            }
        ]]>
    </mx:Script>

    <mx:Label text="Message metrics"/>
    <mx:TextArea id="myTAMess" width="100%" height="20%"/>

    <mx:HTTPService id="srv" destination="catalog"
        useProxy="true"
        result="messageHandler(event);"/>

    <mx:DataGrid dataProvider="{srv.lastResult.catalog.product}"
        width="100%" height="100%"/>

    <mx:Button label="Get Data" click="srv.send()"/>
</mx:Application>
```

Using the server-side classes to gather metrics

For managed endpoints, you can access the total number of bytes serialized and deserialized by using the following methods of the flex.management.runtime.messaging.endpoints.EndpointControlMBean interface:

- `getBytesDeserialized()`

  Returns the total number of bytes deserialized by this endpoint during its lifetime.
- `getBytesSerialized()`

  Returns the total number of bytes serialized by this endpoint during its lifetime.

The flex.management.runtime.messaging.endpoints.EndpointControlMBean class implements these methods.

Writing messaging metrics to the log files

You can write messaging metrics to the client-side log file if you enable the metrics. To enable the metrics, set the `<record-message-times>` or `<record-mssage-sizes>` parameter to `true`, and the client-side log level to `DEBUG`. Messages are written to the log when a client receives an acknowledgment for a pushed message, or a client receives a pushed message from the server. The metric information appears immediately following the debug information for the received message.

The following example initializes logging and sets the log level to DEBUG:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="consumer.subscribe();initLogging();">

    <mx:Script>
        <![CDATA[

            import mx.messaging.messages.AsyncMessage;
            import mx.messaging.messages.IMessage;
            import mx.messaging.events.MessageEvent;
            import mx.messaging.messages.MessagePerformanceUtils;
            import mx.controls.Alert;
            import mx.collections.ArrayCollection;
            import mx.logging.targets.*;
            import mx.logging.*;

            // Event handler to send the message to the server.
            private function send():void
            {
                var message:IMessage = new AsyncMessage();
                message.body.chatMessage = msg.text;
                producer.send(message);
                msg.text = "";
            }

            // Event handler to write metrics to the TextArea control.
            private function ackHandler(event:MessageEvent):void {
                var mpiutil:MessagePerformanceUtils =
                    new MessagePerformanceUtils(event.message);
                myTAAck.text = mpiutil.prettyPrint();
            }

          // Event handler to write metrics to the TextArea control for the message consumer.
            private function messageHandler(event:MessageEvent):void {
                var mpiutil:MessagePerformanceUtils =
                    new MessagePerformanceUtils(event.message);
                myTAMess.text = mpiutil.prettyPrint();
            }

            // Initialize logging and set the log level to DEBUG.
            private function initLogging():void {
                // Create a target.
```

```
                var logTarget:TraceTarget = new TraceTarget();

                // Log all log levels.
                logTarget.level = LogEventLevel.DEBUG;

                // Add date, time, category, and log level to the output.
                logTarget.includeDate = true;
                logTarget.includeTime = true;
                logTarget.includeCategory = true;
                logTarget.includeLevel = true;

                // Begin logging.
                Log.addTarget(logTarget);
            }
        ]]>
    </mx:Script>

    <mx:Producer id="producer" destination="chat" acknowledge="ackHandler(event)"/>
    <mx:Consumer id="consumer" destination="chat" message="messageHandler(event)"/>

    <mx:Label text="ack metrics"/>
    <mx:TextArea id="myTAAck" width="100%" height="20%" text="ack"/>

    <mx:Label text="receive metrics"/>
    <mx:TextArea id="myTAMess" width="100%" height="20%" text="rec"/>

    <mx:Panel title="Chat" width="100%" height="100%">
        <mx:TextArea id="log" width="100%" height="100%"/>
        <mx:ControlBar>
             <mx:TextInput id="msg" width="100%" enter="send()"/>
             <mx:Button label="Send" click="send()"/>
        </mx:ControlBar>
    </mx:Panel>
</mx:Application>
```

For more information on logging, see *Building and Deploying Adobe Flex 3 Applications*.

By default, on Microsoft Windows the log file is written to the file C:\Documents and Settings*USERNAME*\Application Data\Macromedia\Flash Player\Logs\flashlog.txt. The following excerpt is from the log file for the message "My test message":

```
2/14/2008 11:20:18.806 [DEBUG] mx.messaging.Channel 'my-rtmp' channel got connect attempt
status. (Object)#0
  code = "NetConnection.Connect.Success"
  description = "Connection succeeded."
  details = (null)
  DSMessagingVersion = 1
  id = "D46A822C-962B-4651-6F2A-DCB41130C4CF"
  level = "status"
  objectEncoding = 3
2/14/2008 11:20:18.837 [INFO] mx.messaging.Channel 'my-rtmp' channel is connected.
2/14/2008 11:20:18.837 [DEBUG] mx.messaging.Channel 'my-rtmp' channel sending message:
(mx.messaging.messages::CommandMessage)
  body=(Object)#0
  clientId=(null)
  correlationId=""
  destination="chat"
  headers=(Object)#0
  messageId="E2F6B35E-42CD-F088-4B7A-18BF1515F142"
  operation="subscribe"
  timeToLive=0
  timestamp=0
```

```
2/14/2008 11:20:18.868 [INFO] mx.messaging.Consumer 'consumer' consumer connected.
2/14/2008 11:20:18.868 [INFO] mx.messaging.Consumer 'consumer' consumer acknowledge for
subscribe. Client id 'D46A82C3-F419-0EBF-E2C8-330F83036D38' new timestamp 1203006018867
2/14/2008 11:20:18.884 [INFO] mx.messaging.Consumer 'consumer' consumer acknowledge of
'E2F6B35E-42CD-F088-4B7A-18BF1515F142'.
2/14/2008 11:20:18.884 [DEBUG] mx.messaging.Consumer Original message size(B): 626
Response message size(B): 562

2/14/2008 11:20:25.446 [INFO] mx.messaging.Producer 'producer' producer sending message
'CF89F532-A13D-888D-D929-18BF2EE61945'
2/14/2008 11:20:25.462 [INFO] mx.messaging.Producer 'producer' producer connected.
2/14/2008 11:20:25.477 [DEBUG] mx.messaging.Channel 'my-rtmp' channel sending message:
(mx.messaging.messages::AsyncMessage)#0
  body = (Object)#1
    chatMessage = "My test message"
  clientId = (null)
  correlationId = ""
  destination = "chat"
  headers = (Object)#2
  messageId = "CF89F532-A13D-888D-D929-18BF2EE61945"
  timestamp = 0
  timeToLive = 0
2/14/2008 11:20:25.571 [DEBUG] mx.messaging.Channel 'my-rtmp' channel got message
(mx.messaging.messages::AsyncMessageExt)#0
  body = (Object)#1
    chatMessage = "My test message"
  clientId = "D46A82C3-F419-0EBF-E2C8-330F83036D38"
  correlationId = ""
  destination = "chat"
  headers = (Object)#2
    DSMPIO = (mx.messaging.messages::MessagePerformanceInfo)#3
      infoType = "OUT"
      messageSize = 575
      overheadTime = 0
      pushedFlag = true
      receiveTime = 1203006025556
      recordMessageSizes = false
      recordMessageTimes = false
      sendTime = 1203006025556
      serverPostAdapterExternalTime = 0
      serverPostAdapterTime = 0
      serverPreAdapterExternalTime = 0
      serverPreAdapterTime = 0
      serverPrePushTime = 0
    DSMPIP = (mx.messaging.messages::MessagePerformanceInfo)#4
      infoType = (null)
      messageSize = 506
      overheadTime = 0
      pushedFlag = false
      receiveTime = 1203006025556
      recordMessageSizes = true
      recordMessageTimes = true
      sendTime = 1203006025556
      serverPostAdapterExternalTime = 1203006025556
      serverPostAdapterTime = 1203006025556
      serverPreAdapterExternalTime = 0
      serverPreAdapterTime = 1203006025556
      serverPrePushTime = 1203006025556
```

```
  messageId = "CF89F532-A13D-888D-D929-18BF2EE61945"
  timestamp = 1203006025556
  timeToLive = 0

2/14/2008 11:20:25.696 [DEBUG] mx.messaging.Channel Response message size(B): 575
PUSHED MESSAGE INFORMATION:
Originating Message size (B): 506

2/14/2008 11:20:25.712 [INFO] mx.messaging.Producer 'producer' producer acknowledge of
'CF89F532-A13D-888D-D929-18BF2EE61945'.
2/14/2008 11:20:25.712 [DEBUG] mx.messaging.Producer Original message size(B): 506
Response message size(B): 562
Total time (s): 0.156
Network Roundtrip time (s): 0.156
```