ADOBE® BLAZEDS

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, Flash, and Flex, 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.

Windows is either a registered trademark or trademarks of Microsoft Corporation in the United States and/or other countries. Java is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries. All other trademarks are the property of their respective owners.

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 Data Services Applications

Chapter 1: Taking the BlazeDS Test Drive

Getting started with the Test Drive 2

Chapter 2: Understanding BlazeDS

About BlazeDS 4

About RPC services 4

About messaging 6

Chapter 3: The Messaging Framework

About the messaging framework 7

Non-streaming channels 12

Including real-time streaming channels 15

Securing HTTP-based endpoints 19

Working with session data 20

Custom error handling 21

Part 2: RPC Components

Chapter 4: Understanding RPC Components

RPC components 24

Comparing the RPC capability to other technologies 28

Chapter 5: Creating RPC Clients

Declaring an RPC component 30

Configuring a destination 33

Calling a service 36

Setting properties for RemoteObject methods or WebService operations 43

Handling service results 45

Using a service with binding, validation, and event listeners 55

Handling asynchronous calls to services 56

Using capabilities specific to RemoteObject components 58

Using capabilities specific to WebService components 60

Chapter 6: Configuring RPC Services on the Server

Destination configuration 65

Configuring destination properties 67

Configuring the Proxy Service 69

Chapter 7: Serializing Data

Serializing between ActionScript and Java 70

Serializing between ActionScript and web services 79

Chapter 8: Extending Applications with Factories
The factory mechanism 87

Part 3: Messaging

Chapter 9: BlazeDS Message Service
Messaging 93
Messaging architecture 94

Chapter 10: Creating Messaging Clients
Using messaging in a Flex application 96
Working with Producer components 96
Working with Consumer components 100
Using subtopics 103
Using a pair of Producer and Consumer components in an application 106

Chapter 11: Configuring Messaging on the Server
Understanding Message Service configuration 108
Configuring Message Service destinations 110
Creating a custom Message Service adapter 116

Chapter 12: Controlling Message Delivery with Adaptive Polling
Adaptive polling 118
Using a custom queue processor 119

Chapter 13: The Ajax Client Library
About the Ajax client library 124
Using the Ajax client library 125
Ajax client library API reference 128

Chapter 14: Measuring Message Processing Performance
About measuring message processing performance 134
Measuring message processing performance 139

Part 4: Administering BlazeDS Applications

Chapter 15: Configuring BlazeDS
About service configuration files 147
Securing destinations 153
Using software clustering 158
Monitoring and managing services 161
About data services class loading 163
Server-side service logging 164

Chapter 16: Run-Time Configuration

About run-time configuration 167
Configuring components with a bootstrap service 168
Configuring components with a remote object 168
Accessing dynamic components with a Flex client application 170

Part 1: Getting Started with Data Services Applications

Taking the BlazeDS Test Drive 2
Understanding BlazeDS 4
The Messaging Framework 7

Chapter 1: Taking the BlazeDS Test Drive

To learn more about how BlazeDS works and what it can do, you can take the BlazeDS Test Drive. The Test Drive includes some sample applications that demonstrate basic capabilities and best practices for developing applications with BlazeDS.

Topics

Getting started with the Test Drive. 2

Getting started with the Test Drive

Before you begin the BlazeDS Test Drive, you should be familiar with the Adobe Flex programming environment and how to use Flex to build rich Internet applications. For more information, see the Flex Help Resource Center.

You must install BlazeDS to access the Test Drive.

After you start the Integrated BlazeDS server, open a browser and go to http://<*hostname*>:<*port_num*>/blazeds. A link to the Test Drive appears at the top of the page. The Test Drive includes several sample applications: a catalog with mobile phone product information (one with text only, and the other with images), and a chat client.

Sample 1: Accessing data using an HTTP service

You can use the HTTPService component to send and receive HTTP requests by using HTTP GET or POST. The HTTPService component consumes different types of responses, but typically you use it to consume XML. HTTPS-ervice calls are asynchronous. You also use the component with any kind of server-side technology, such as JSP, Servlet, ASP, Ruby on Rails, and PHP.

In the sample, when you click the Get Data button, the data grid is populated with XML data returned by a JSP file. It shows built-in data grid capabilities such as sortable and movable columns. You can click the column head to sort data in ascending or descending order. Drag a column head to move a column of data to another location in the grid. You can also increase or decrease the width of the columns from the header row.

Sample 2: Accessing data using a web service

With the WebService component, you can invoke SOAP-based web services deployed on your application server or the Internet. WebService component calls are also asynchronous.

The sample also uses data grid column definitions. When you click the Get Data button, the grid is populated with data returned by a web service.

Sample 3: Accessing data using Java RPC

Like HTTPService and WebService components, Java RPC calls are also asynchronous. With the RemoteObject component, you can directly invoke methods of Java objects deployed in your application server, and consume the return value. The return values can be a primitive data type, an object, a collection of objects, or an object graph.

In the sample, when you click the Get Data button, the grid is populated with data returned by the `getProducts()` method of the ProductService Java class.

Sample 4: Flex Programming Model 101

Like in any other object-oriented programming language, a Flex application is made of a collection of classes. Using Flex, you can create classes using MXML or ActionScript. You typically create view classes in MXML, and Model and Controller classes in ActionScript. After you define a class, you can use it programmatically or declaratively (as a tag in MXML) without the need for an additional descriptor file. Public properties are automatically available as tag attributes.

In this sample, basic Flex capabilities are displayed. When you click an image in the Catalog panel on the left side of the page, more information appears in the Product Details panel on the right.

Sample 5: Updating data

In this sample, you can update data from one panel of data and see the changes quickly displayed in another panel. The panel on the left shows a data grid with sortable, movable columns. When you click on a row in the Catalog panel on the left, detailed information about that item appears in Details panel on the right. In the Details panel you can modify data in any field, click Update, and the change is passed from the client side to the server side.

Sample 6: Publish-subscribe messaging (data push use case)

Flex supports publish-subscribe messaging through the BlazeDS Message Service. The Message Service manages a set of destinations that Flex clients can publish and subscribe to. Flex provides two components, Producer and Consumer, that you use to publish and subscribe to a destination, respectively.

In the example, a Java component publishes simulated real-time values to a message queue. The Flex client subscribes to that queue and displays the values in real time. Click the Subscribe To 'Feed' destination button. Pushed values appear in the text field. Click the Unsubscribe To 'Feed' destination button to stop the feed.

Sample 7: Publish-subscribe messaging (collaboration use case)

The sample builds on the concepts and APIs introduced in sample 6. The messaging and real-time infrastructure available in Flex lets you build collaboration and data push applications to in a scalable and reliable manner while preserving the lightweight web deployment model.

Open the sample application in two different browser windows. Enter a message in the Send box at the lower edge of the page, and click the Send button. The message appears in the Chat windows in both clients.

Chapter 2: Understanding BlazeDS

Adobe® BlazeDS provides powerful capabilities for using external data in an Adobe® Flex™ application. BlazeDS runs on standard J2EE platforms and servlet containers.

Topics

About BlazeDS. 4
About RPC services 4
About messaging. 6

About BlazeDS

BlazeDS provides a comprehensive set of data-enabling capabilities, which are deployed in a Java web application. BlazeDS also provides a messaging infrastructure for building data-rich Flex applications. This infrastructure underlies the BlazeDS capabilities that are designed for moving data to and from applications: RPC services and the Message Service.

The following table describes the types of services provided in BlazeDS:

Services	Description
RPC services	Provides a call and response model for accessing external data. Lets you create applications that make asynchronous requests to remote services that process the requests, and then return data to your Flex application. For more information, see "About RPC services" on page 4.
Message Service	Provides messaging services for collaborative and real-time applications. Lets you create applications that can send messages to and receive messages from other applications, including Flex applications and Java Message Service (JMS) applications. For more information, see "About messaging" on page 6.

About RPC services

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

A client-side RPC component, which you can create in MXML or ActionScript, calls a remote service, and then stores response data from the service in an ActionScript object from which you can easily obtain the data. The implementation of an RPC service can be an HTTP URL, which uses HTTP POST or GET; a SOAP-compliant web service; or a Java object in a Java web application. The client-side RPC components are the HTTPService, WebService, and RemoteObject components.

The following diagram provides a simplified view of how RPC components interact with RPC services:

Client tier

User interface

HTTPService component - Send and receive data using HTTP GET or POST
WebService component - Interact with web services using SOAP over HTTP
RemoteObject component - Invoke methods on remote Java objects

Request

Result

Data format: AMF, SOAP, or HTTP

RPC service implementations: Java object, web service, other

Resource tier

Server tier

Using RPC components with BlazeDS

You use BlazeDS with RPC components when you 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; using Flex SDK without BlazeDS or ColdFusion does not provide this functionality. When you use BlazeDS, instead of contacting services directly, RPC components contact destinations. *Destinations* are manageable service endpoints that you manage through a server-side XML-based configuration file.

You can use Flex SDK without BlazeDS to create nonenterprise applications that call HTTP services or web services directly, without going through a server-side proxy service. You cannot use RemoteObject components without BlazeDS or ColdFusion.

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 RPC service must either be on the server hosting your application, or the remote server that hosts the RPC 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.

For more information, see “Understanding RPC Components” on page 24.

About messaging

The messaging capability in BlazeDS is built on the same underlying messaging infrastructure as the RPC services. By using messaging components, you create collaborative and real-time messaging applications. These components provide a client-side API for passing text and objects to and from messaging clients, and server-side configuration of message destinations.

You can create messaging components in MXML or ActionScript that connect to server-side message destinations, send messages to those destinations, and receive messages from other messaging clients. Components that send messages are called *producers*, and those that receive messages are called *consumers*. Messages are sent over transport *channels*, which use specific transport protocols, such as the Action Message Format (AMF).

Messaging clients can be Flex applications or other types of applications, such as Java Message Service (JMS) applications. JMS is a Java API that lets applications create, send, receive, and read messages. JMS applications can publish and subscribe to the same message destinations as Flex applications. However, you can create a wide variety of messaging applications using just BlazeDS messaging.

For more information, see "BlazeDS Message Service" on page 93.

Chapter 3: The Messaging Framework

Adobe BlazeDS uses a common messaging framework for all types of data services, including the Message Service, Proxy Service, and Remoting Service. Some aspects of this messaging framework apply to all types of data services.

Topics

About the messaging framework 7
Non-streaming channels 12
Including real-time streaming channels 15
Securing HTTP-based endpoints 19
Working with session data 20
Custom error handling 21

About the messaging framework

The BlazeDS messaging framework sends messages back and forth between server-side data services and clients. A publish-subscribe messaging pattern is at the heart of BlazeDS. A client-side message *producer* creates a message and sends it to a logical destination on the server. A client-side message *consumer* subscribes to the destination and listens for messages that are sent to it.

Message agents

Message producers and consumers, collectively called *message agents*, exchange messages through a common destination. The messaging framework composes a message on the client when an operation is performed. For example, a text message is published to a Message Service destination or a remote method is invoked on an RPC service is modified.

The messaging framework uses a producer to send the message to the appropriate destination. In the case of the Message Service, application code creates and controls producers in application code. In the cases of the RPC services, the messaging framework internally creates producers and composes and sends messages.

To successfully handle a given message that a producer creates, the messaging framework must understand the structure of the message. The destination must be able to correlate messages and, in some cases, persist messages so that they can be reliably routed to consumers. Any time a message is sent, it must be acknowledged for the client to know that it was properly received. The remote service sends an AcknowledgeMessage to the client to satisfy this requirement. The client also dispatches an event for application code when these acknowledgments occur.

Channels and endpoints

To send messages across the network, the client and server use channels that encapsulate message formats, network protocols, and network behaviors and decouple them from services, destinations, and application code. A channel formats and translates messages into a network-specific form and delivers it to an endpoint on the server. The destination and channel information is available to the client as a result of compile-time or runtime access to configuration settings that the server uses to manage services at runtime.

A server-side channel *endpoint* unmarshals messages in a protocol-specific manner and then passes the messages in generic Java form to the *message broker*, which determines where messages should be sent and routes them to the appropriate service.

To accommodate different types of applications, BlazeDS has channels for RPC-style messaging, simple polling, near real-time messaging, and real-time messaging. You can use standard AMF and HTTP channels for RPC services, polling AMF or HTTP channels for near real-time messaging, and streaming AMF and HTTP channels for real-time messaging (data pushed from the server). For more information about channels, see "Channel types" on page 8.

Message broker

The message broker performs authorization checks and routes messages to the appropriate service based on its message type. The first service found that can process the message receives it, locates the destination the message targets, and potentially uses a service adapter if the destination requires one for back-end access. Depending on the the message type, the service and its adapters might perform logic based on the incoming message, such as invoking operations in the case of an RPC service, or managing subscriptions in the case of an adapter-less Message Service destination.

A service or service adapter can request that a message be sent to other connected clients, causing the message to flow back into the message broker and out of the appropriate endpoints to other connected clients. This is a true push operation if the channel is capable of real-time push, or it may be a polling operation if the channel relies upon a protocol that does not support real-time messaging. It could be a mix of the approaches, with some consumers polling and some accessing the data in real time, depending on how the destination's channels are configured. The service and destinations are decoupled from the actual network operation and protocol-specific message formats, as is the application code. The messaging framework sends special command messages along the same path. This type of message represents infrastructure operations, such as the act of subscribing, rather than the publishing of application message data.

Channel types

BlazeDS includes several types of channels. There are standard (unsecured) and secure Action Message Format (AMF) channels, and HTTP channels. There are AMF and HTTP channels for simple non-polling request-response patterns and for client polling patterns to simulate real-time messaging. The streaming AMF and HTTP channels are exclusively true data streaming for real-time messaging (data pushed from the server). If you require data security, use one of the secure channels.

The AMF channels enable binary representations of ActionScript object graphs in the body of an HTTP (or SSL-wrapped HTTPS) POST message. Because AMF is binary, AMF-based channels provide significant performance and band width advantages over standard HTTP channels, which send non-binary XML and text in the body on an HTTP POST message.

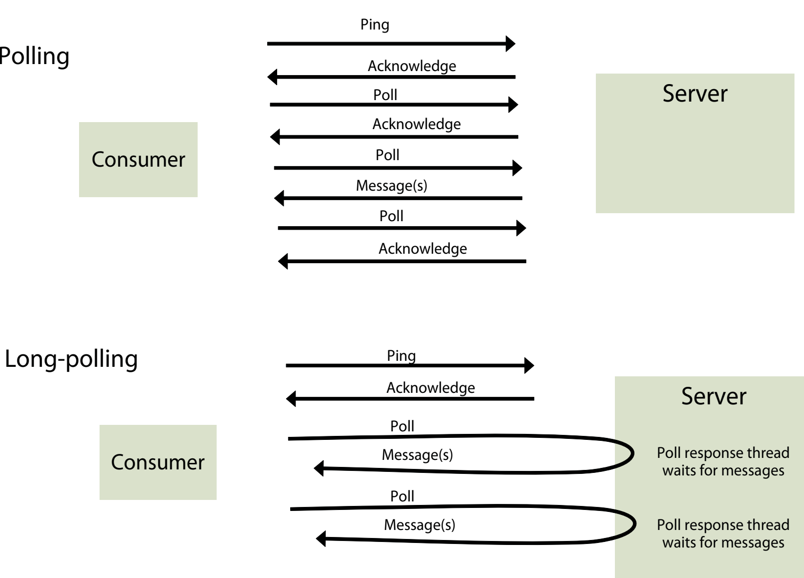
You can use any channel with any service destination, although some combinations make more sense than others. However, if a non-polling channel is configured for a publish-subscribe Message Service destination, only the destination's producers use that channel, and the framework searches for a different channel configured in the destination to provide messages to consumers. It doesn't make sense for a consumer to use a non-polling non-real-time channel, because it would never be able to receive its messages without a server push or a periodic client poll.

You create channel definitions in the services-config.xml file or at run time. The endpoint URL of each channel definition must be unique. For information about creating channels at run time, see "Run-Time Configuration" on page 167.

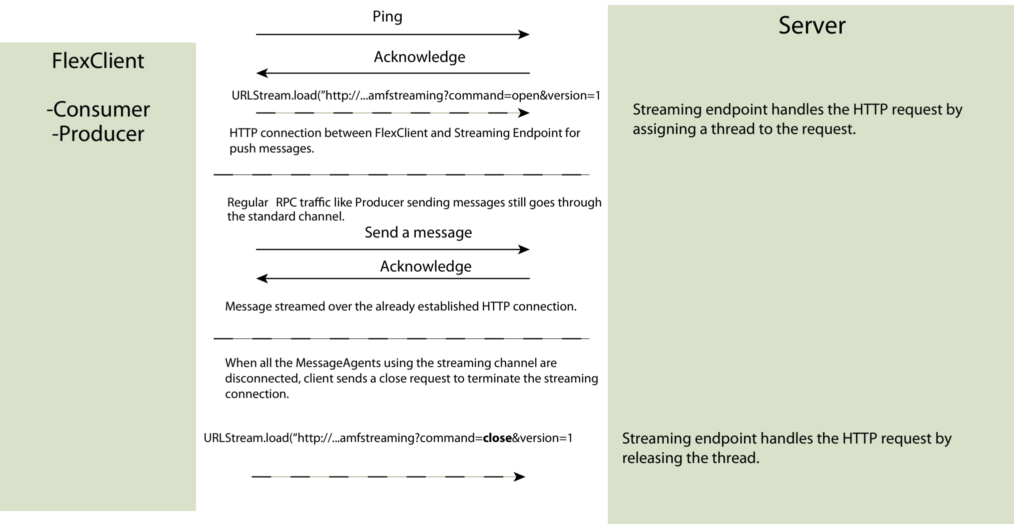
For real-time messaging or to approximate real-time messaging, you can choose true streaming channels, long polling channels, or standard polling channels. Each of these channels types uses a different network traffic pattern.

Note: *LiveCycle Data Services ES now includes an NIO-based HTTP server, which provides asynchronous, non-blocking NIO-based AMF and HTTP endpoints that support highly scalable long-polling and streaming client HTTP connections. This server provides functionality similar to that of the RTMP server that you use with RTMP-based endpoints, but for standard HTTP-based endpoints. For more information, see* *"Using NIO AMF and NIO HTTP Endpoints" on page 35*.

The following illustration shows the network traffic pattern for standard polling and long polling channels. In the standard polling scenario, the client constantly polls the server for new messages even when no messages are yet available. In contrast, in the long polling scenario the poll response thread waits for messages to be available and then returns the messages to the client.



The following illustration shows the network traffic pattern for the streaming AMF and HTTP channels. In this scenario, an HTTP connection thread is established between the FlexClient object on the client and the streaming endpoint on the server. Messages are streamed over this open connection. When all of the producers and consumers using the streaming channel are disconnected, the HTTP connection thread is closed.



Channel configuration

You can configure message channels in the services-config.xml file or at run time; for more information on run time configuration, see "Run-Time Configuration" on page 167. The following example shows an AMF channel definition in the services-config.xml file. The channel configuration is preceded by a destination that references it.

```
...
    <destination id="MyTopic">
...
        <channels>
            <channel ref="samples-amf"/>
        </channels>
...
    </destination>
...

    <channel-definition id="samples-amf"
        type="mx.messaging.channels.AMFChannel">
        <endpoint url="/myapp/messagebroker/amf" port="8100"
            type="flex.messaging.endpoints.AmfEndpoint"/>
    </channel-definition>
</channels>
...
```

By default, endpoints that use the AMF protocol use an optimization mechanism that reduces the size of messages transported across the network between clients and server-based messaging endpoints. If necessary, you can disable this optimization by settting the channel definition's `enable-small-messages` serialization property to `false`, as the following example shows:

```
<channel-definition ... >
    <endpoint ... />
    <properties>
        <serialization>
            <enable-small-messages>false</enable-small-messages>
        </serialization>
    </properties>
</channel-definition>
```

You can also assign channels on the client at run time by creating a ChannelSet object that contains one or more Channel objects, and assigning the ChannelSet to a service component, as the following MXML example shows:

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

The following example shows ActionScript code creating a ChannelSet object:

```
...
// Create a ChannelSet.
var cs:ChannelSet = new ChannelSet();
// Create a Channel.
var customChannel:Channel = new AMFChannel("my-polling-amf", endpointUrl);
// Add the Channel to the ChannelSet.
cs.addChannel(customChannel);
// Assign the ChannelSet to a RemoteObject instance.
myRemoteObject.channelSet = cs;
...
```

When you configure channels at run time, you can dynamically control the endpoint URL. You can add any number of Channel objects to a ChannelSet; the ChannelSet object searches through the set to find a channel to which it can connect. It is a best practice to specify an `id` property that the destination recognizes, but you can pass null to skip `id` property checking.

In addition to the searching behavior provided by the ChannelSet class, the Channel class defines a `failoverURIs` property. This property lets you configure a Channel object 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 this fails, and the channel defines failover URIs, each is attempted before the channel gives up and the ChannelSet object searches for the next available channel. If no channel in the set can connect, any pending unsent messages generate faults on the client.

Non-streaming channels

You generally use non-streaming message channels without client polling for RPC service destinations, which require simple call and response messaging. When working with a Message Service destination, you can use these channels with standard client polling to constantly poll for new messages from the server, or with long client polling to provide near real-time messaging when using a true streaming channel is not an option in your network environment.

Non-streaming AMF channels

You use AMF-based channels to send AMF-encoded messages to the AMF endpoint, which supports AMF requests and responses sent over HTTP. You can use the AMF channel, the polling AMF channel, or the secure AMF channel.

You can configure these channels for near–real-time messaging by setting specific endpoint properties; for more information, see “Near-real-time messaging on HTTP and AMF channels” on page 13.

AMF channel

The following example shows a non-polling AMF channel definition:

```
<channel-definition id="samples-amf"
    type="mx.messaging.channels.AMFChannel">
    <endpoint url="http://{server.name}:8100/myapp/messagebroker/amf"
        type="flex.messaging.endpoints.AmfEndpoint"/>
</channel-definition>
```

Secure AMF channel

The secure AMF channel is similar to the AMF channel, but uses HTTPS instead of HTTP. It uses a different endpoint and class than the AMF channel. The following example shows a secure AMF channel definition:

```
<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>
```

Polling AMF channel

You can use an AMF polling channel to repeatedly poll the AMF 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 `polling()` method of a channel for which polling is enabled; for example, you might want set the polling interval to a high number so that the channel does not automatically poll, and call the `polling()` method to poll manually based on an event, such as a button click.

When you use the polling AMF channel for publish-subscribe messaging, you set the `polling` property to `true` in the channel definition. You can also configure the polling interval in the channel definition. The following example shows an AMF polling channel definition with polling enabled:

```
<channel-definition id="samples-polling-amf"
    type="mx.messaging.channels.AMFChannel">
    <endpoint url="http://{server.name}:8100/dev/messagebroker/amfpolling"
        type="flex.messaging.endpoints.AmfEndpoint"/>
    <properties>
            <polling-enabled>true</polling-enabled>
            <polling-interval-seconds>8</polling-interval-seconds>
    </properties>
</channel-definition>
```

Non-streaming HTTP channels

You use the HTTP and secure HTTP channels to connect to the HTTP endpoint, which supports HTTP requests and responses. These channels use a text-based (XML) message format.

You can configure these channels for near–real-time messaging by setting specific endpoint properties; for more information, see "Near-real-time messaging on HTTP and AMF channels" on page 13.

HTTP channel

The following example shows a non-polling HTTP channel definition:

```
<channel-definition id="my-http"
    class="mx.messaging.channels.HTTPChannel">
    <endpoint url="http://{server.name}:8100/dev/messagebroker/http"
        class="flex.messaging.endpoints.HTTPEndpoint"/>
</channel-definition>
```

You should not use types that implement the IExternalizable interface (used for custom serialization) with the HTTP channel 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. For more information about the IExternalizable interface, see "Serializing Data" on page 70.

You can specify a URL in the `redirectURL` property in the `properties` section of a channel definition when using HTTP-based channel endpoints. This is a legacy capability that lets you redirect a request to another URL if the MIME type of the request wasn't correct.

Secure HTTP channel

The secure HTTP channel is similar to the HTTP channel, but you use HTTPS instead of HTTP. Uses a different channel class than the HTTP channel.

The following example shows a secure HTTP channel definition:

```
<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>
```

Near-real-time messaging on HTTP and AMF channels

The HTTP channel uses XML over an HTTP connection. The AMF channel uses the AMF binary format over the HTTP protocol. Each of these channels supports simple polling mechanisms that can be used by clients to request messages from the server. You can use these channels to get pushed messages to the client when the other more efficient and real-time mechanisms are not suitable. The polling interval is configurable or polling can be controlled more flexibly from ActionScript code on the client. This mechanism uses the application server's normal HTTP request processing logic and works with typical J2EE deployment architectures. When you use the HTTP channel or AMF channel, the FlexSession maps directly to the underlying application server's HTTP session so the application server's session failover can be used to provide fault tolerant session storage. You typically should ensure your application server is configured to use sticky sessions so one clients requests go to the same application server process instance.

You can establish near–real-time messaging for any channel that uses an HTTP or AMF endpoint by setting the `polling-enabled`, `polling-interval-millis`, and `wait-interval-millis` properties in a channel definition. This applies to the HTTP, AMF, and polling AMF channels, and the secure HTTP and secure AMF channels.

In the default configuration for a polling HTTP or AMF channel, the channel does not wait for messages on the server. When the poll request is received, it checks to see if there are any messages queued up for the polling client and if so, those messages are delivered in the response to the HTTP request. To achieve near–real-time messaging, set the following properties in the `properties` section of a channel definition in the services-config.xml file:

- `polling-enabled` property to `true` to enable polling.
- `polling-interval-millis` property to `0` (zero). This is the number of milliseconds between client poll request.
- `wait-interval-millis` property to `-1`. This is the number of milliseconds that the server thread responding to a client poll request waits for messages from that client.
- `max-waiting-poll-requests` property to a value that is at least 10 less than the number of application server request-handling threads. How you set or determine the maximum number of threads for your application server depends on application server that you use; for more information, see your application server documentation.

When you use these settings, the client polls once and because the wait-interval is set to -1, the server waits to respond if there are no messages available and responds as soon as it has messages for the client.

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 non-zero 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 needs to process, but you are limited by the number of threads on the server.

You configure a wait-interval-millis on the channel to indicates the amount of time the server thread should wait for pushed messages. If that interval is set to -1, it means to wait indefinitely - as long as the client can maintain that connection to the server. As soon as any messages are received for that client, we push them in the response to the request. The polling-interval-millis on the client then is used to determine when next to poll the server. If you set poll-interval-millis to 0 and wait-interval-millis to -1, you get real time behavior from this channel.

The following example shows an AMF channel definition configured for near–real-time messaging:

```
<channel-definition id="my-amf" class="mx.messaging.channels.AMFChannel">
    <endpoint url="http://servername:8100/contextroot/messagebroker/amf"
        class="flex.messaging.endpoints.AMFEndpoint"/>
    <properties>
        <polling-enabled>true</polling-enabled>
        <polling-interval-millis>0</polling-interval-millis>
        <!-- Optional. Default is 0. This parameter specifies the number
            of milliseconds the server poll response thread will wait
            for new messages to arrive when the server has no messages
            for the client at the time of poll request handling. Zero means
            that server does not wait for new messages for the client
            and returns an empty acknowledgment as usual. -1 means that
            server waits indefinitely until new messages arrive for the
            client before responding to the client poll request. -->
            <wait-interval-millis>-1</wait-interval-millis>
        <!-- Optional. Default is 0. This parameter specifies the maximum
            number of server poll response threads that can be in wait
            state. When this limit is reached, the subsequent poll requests
            will be treated as having zero wait-interval-millis.-->
        <max-waiting-poll-requests>0</max-waiting-poll-requests>
    </properties>
</channel-definition>
```

The caveat for using the poll-wait-interval with BlazeDS is the utilization of available application server threads. Because this channel ties up one application server's request handling thread for each waiting client connection, this mechanism can have an impact on server resources and performance. Modern JVMs can typically support a couple of hundred threads comfortably if given enough heap space. You should check the maximum thread stack size (often 1 or 2M per thread) and make sure you have enough memory and heap space for the number of application server threads you configure.

To ensure Flex clients using channels with wait-interval-millis do not lock up your application server, BlazeDS requires that you set an additional configuration attribute specifying the maximum number of waiting connections that BlazeDS should manage. This is configured through the max-waiting-poll-requests property. This number must be set to a number smaller than the number of HTTP request threads your app server is configured to use. For example, you might configure the application server to have at most 200 threads and allow at most 170 waiting poll requests. This would ensure 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 which are I/O heavy may need a large number of threads to ensure all I/O channels are utilized completely. Examples where you need application server threads include: # of threads simultaneously writing responses to clients behind slow network connections, # of threads used to execute database queries or updates, and # of threads needed to perform computation on behalf of user requests.

The other consideration for using wait-interval-millis is that BlazeDS must avoid monopolizing the available connections the browser will allocate for communicating with a server. The HTTP 1.1 spec mandates that browsers allocate at most 2 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 app server session at a time. If a new polling request comes in over the same HTTP session as an existing waiting request, the waiting request returns and the new request starts waiting in its place.

If you have an application that has more than one player instance on the same page, beware that only one of those instances can have a real time HTTP channel connection open to the server at the same time.

Including real-time streaming channels

BlazeDS supports a variety of deployment options that offer real-time messaging from Flex clients to Java 2 Enterprise Edition (J2EE) servers. Although efficient real-time messaging is difficult when client and application server are both behind firewalls, BlazeDS attempts to maximize the real time experience while keeping load on the server resources manageable.

The following issues prevent a simple socket from being used to create a real-time connection between client and server in a typical deployment of a rich Internet application:

- The client's only access to the Internet is through a proxy server.
- The application server on which BlazeDS is installed can only be accessed from behind the web tier in the IT infrastructure.

BlazeDS offers a few approaches for creating efficient real-time connections in these environments, but in the worse case, the server fall's back to simple polling. The main disadvantages of polling are increased overhead on both client and server machines and reduced response times for pushed messages.

One design goal for BlazeDS is to create an environment where J2EE code can be run in a channel-independent manner. Your backend server code can either be used in a traditional J2EE context or with protocols that offer scalable real time connectivity currently not available in the J2EE platform. For example, BlazeDS provides the FlexSession, which mirrors the functionality available with the HttpSession for the standard HTTP protocol that J2EE supports.

You can configure clients using a list of channels that they can use to connect to the server. A client attempts to use the first channel in the list, and if that fails, it tries the next channels until it finds one it can connect to. You also could use this mechanism to provide redundant server infrastructures to provide high reliability.

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.

Using HTTP-based streaming is similar to setting a long polling interval on a standard AMF or HTTP channel, 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. See “Channel types” on page 8for illustrations of the network traffic patterns for long polling and HTTP-based streaming. Unlike polling channels, because streaming channels keep a constant connection open, they can be adversely affected by proxies and other code in the middle layer between the Flex client and BlazeDS server.

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

Property	Description
idle-timeout-minutes	Optional. Default value is 0. Specifies the number of minutes that a streaming channel is allowed to remain idle before it is closed. Setting idle-timeout-minutes to 0 disables the timeout completely, but this is a potential security concern.
max-streaming-clients	Optional. Default value is 10. Limits the number of FlexClient objects on the client-side using the endpoint. You should determine an appropriate value by considering 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. The value should be a number that is lower than the maximum number of threads available on the application server. Note that this value is for the number of FlexClient objects, which can each contain one or more message agent (Producer or Consumer object).

Property	Description
server-to-client-heartbeat-millis	Optional. Default value is 5000. Number of milliseconds that the server waits before writing a single null byte to the streaming connection to make sure the client is still available. This is important to determine when a client is no longer available so that its associated thread on the server can be cleaned up. Note that this functionality keeps the session alive. A non-positive value disables this functionality.
<user-agent-settings> <user-agent match-on="MSIE" kickstart-bytes= "2048" max-streaming-connections-per-session="1"/> <user-agent match-on="Firefox" kickstart-bytes= "2048" max-streaming-connections-per-session="1"/> </user-agent-settings>	Optional. Default values are as shown. User agents are used to customize the streaming endpoint for specific web browsers. This is needed for following reasons: • A certain number of bytes must written before the endpoint can reliably use a streaming connection. This is specified by the kickstart-bytes attribute. • There is a browser-specific limit to the number of connections per session. In Firefox, the limit is eight connections per session. In Internet Explorer, the limit is four connections per session. Therefore, BlazeDS must limit the number of streaming connections per session. By default, BlazeDS uses 1 as the value for max-streaming-connections-per-session, but technically, it can support one less than the connections per session enforced by the browser. You can change the values for Internet Explorer and Firefox, and you can add other browser-specific limits by specifying a new user-agent element with a match-on value for a specific browser.

Streaming AMF and HTTP use the following standard and secure channel classes on the Flex client:

- StreamingAMFChannel
- StreamingHTTPChannel
- SecureStreamingAMFChannel
- SecureStreamingHTTPChannel

On the BlazeDS server, the following channel endpoints handle the requests for these channels:

- StreamingAMFEndpoint
- StreamingHTTPEndpoint
- SecureStreamingAMFEndpoint
- SecureStreamingHTTPEndpoint

The streaming channel endpoints are manageable using the StreamingAMFEndpointControl and StreamingHTTPEndpointControl MBean implementation classes, respectively. These MBeans use the following properties to record data. For more information about MBeans, see “Monitoring and managing services” on page 161.

Property	Description
MaxStreamingClients	Maximum number of streaming clients the endpoint supports.
StreamingClientsCount	Number of streaming client the endpoint currently has.

Streaming AMF channel configuration

You configure the streaming AMF channel similarly to the other HTTP-based channels, but instead of polling-related configuration options (polling-enabled, polling-interval, wait-interval, max-waiting-poll-requests), there are streaming-related configuration options.The following example shows a streaming AMF channel definition:

```
        <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>
                <idle-timeout-minutes>0</idle-timeout-minutes>
                <max-streaming-clients>10</max-streaming-clients>
                <server-to-client-heartbeat-millis>5000</server-to-client-heartbeat-
millis>
                <user-agent-settings>
                    <user-agent match-on="MSIE" kickstart-bytes="2048" max-streaming-
connections-per-session="1"/>
                    <user-agent match-on="Firefox" kickstart-bytes="2048" max-streaming-
connections-per-session="1"/>
                </user-agent-settings>
            </properties>
        </channel-definition>
```

Secure streaming AMF channel configuration

A secure streaming AMF channel is similar to a standard streaming AMF channel but uses different values in the channel-definition and endpoint elements as the following example shows:

```
        <channel-definition id="my-secure-streaming-amf"
class="mx.messaging.channels.SecureStreamingAMFChannel">
            <endpoint
url="http://{server.name}:{server.port}/{context.root}/messagebroker/streamingamfsecure"
class="flex.messaging.endpoints.SecureStreamingAMFEndpoint"/>
            <properties>
                <idle-timeout-minutes>0</idle-timeout-minutes>
                <max-streaming-clients>10</max-streaming-clients>
                <server-to-client-heartbeat-millis>5000</server-to-client-heartbeat-
millis>
                <user-agent-settings>
                    <user-agent match-on="MSIE" kickstart-bytes="2048" max-streaming-
connections-per-session="1"/>
                    <user-agent match-on="Firefox" kickstart-bytes="2048" max-streaming-
connections-per-session="1"/>
                </user-agent-settings>
            </properties>
        </channel-definition>
```

Streaming HTTP channel configuration

You configure the streaming HTTP channel similarly to the other HTTP-based channels, but instead of polling-related configuration options (polling-enabled, polling-interval, wait-interval, max-waiting-poll-requests), there are streaming-related configuration options. The following example shows a streaming HTTP channel definition:

```
        <channel-definition id="my-streaming-http"
class="mx.messaging.channels.StreamingHTTPChannel">
            <endpoint
url="http://{server.name}:{server.port}/{context.root}/messagebroker/streaminghttp"
class="flex.messaging.endpoints.StreamingHTTPEndpoint"/>
            <properties>
                <idle-timeout-minutes>0</idle-timeout-minutes>
                <max-streaming-clients>10</max-streaming-clients>
                <server-to-client-heartbeat-millis>5000</server-to-client-heartbeat-
millis>
                <user-agent-settings>
                    <user-agent match-on="MSIE" kickstart-bytes="2048" max-streaming-
connections-per-session="1"/>
```

```
                    <user-agent match-on="Firefox" kickstart-bytes="2048" max-streaming-
connections-per-session="1"/>
                </user-agent-settings>
            </properties>
        </channel-definition>
```

Secure streaming HTTP channel configuration

A secure streaming HTTP channel is similar to a standard streaming HTTP channel but uses different values in the channel-definition and endpoint elements as the following example shows:

```
        <channel-definition id="my-secure-streaming-http"
class="mx.messaging.channels.SecureStreamingHTTPChannel">
            <endpoint
url="http://{server.name}:{server.port}/{context.root}/messagebroker/streaminghttpsecure"
class="flex.messaging.endpoints.SecureStreamingHTTPEndpoint"/>
            <properties>
                <idle-timeout-minutes>0</idle-timeout-minutes>
                <max-streaming-clients>10</max-streaming-clients>
                <server-to-client-heartbeat-millis>5000</server-to-client-heartbeat-
millis>
                <user-agent-settings>
                    <user-agent match-on="MSIE" kickstart-bytes="2048" max-streaming-
connections-per-session="1"/>
                    <user-agent match-on="Firefox" kickstart-bytes="2048" max-streaming-
connections-per-session="1"/>
                </user-agent-settings>
            </properties>
        </channel-definition>
```

Securing HTTP-based endpoints

You can protect HTTP-based channel endpoints by using whitelists and blacklists that list specific firewall, router, or web server IP addresses. A whitelist contains client IP addresses that are permitted to access endpoints. A blacklist contains client IP addresses that are restricted from accessing endpoints.

The blacklist takes precedence over the whitelist in the event that the client IP address is a member of both the whitelist and blacklist.

The `whitelist` and `blacklist` elements can contain 0-N `ip-address` and/or `ip-address-pattern` elements. The `ip-address` element supports simple IP address matching, allowing individual bytes in the address to be designated as a wildcard by using an asterisk character (*).

The `ip-address-pattern` element supports regular expression pattern matching of IP addresses. This allows for powerful range-based IP address filtering.

The following example shows a whitelist and a blacklist:

```
<whitelist>
    <ip-address-pattern>237.*</ip-address-pattern>
    <ip-address>10.132.64.63</ip-address>
</whitelist>

<blacklist>
    <ip-address>10.60.147.*</ip-address>
    <ip-address-pattern>10\\.132\\.17\\.5[0-9]{1,2}</ip-address-pattern>
</blacklist>
```

Working with session data

BlazeDS provides the following classes for working with session data. These classes are included in the public BlazeDS Javadoc documentation.

- flex.messaging.FlexContext
- flex.messaging.FlexSession
- flex.messaging.FlexSessionListener
- flex.messaging.FlexSessionAttributeListener
- flex.messaging.FlexSessionBindingEvent
- flex.messaging.FlexSessionBindingListener

The FlexContext class is useful for getting access to the session and the HTTP pieces of the session, such as the HTTP servlet request and response. This lets you access HTTP data when you use a Flex application in the context of a larger web application where other classes, such as JSPs or Struts actions, might have stored information.

The FlexSession class provides access to an ID and also provides `setAttribute` and `getAttribute` functionality. This is useful for storing data on the server that doesn't have to go back to the client. However, FlexSession is not cluster-aware; if a client connects to a different server in the cluster, the client receives a new FlexSession. Nothing stored in the FlexSession attributes is persisted for clustering purposes. The FlexSessionListener class is useful for monitoring who is connected. You add a listener by using the static method to track new connections being made. You receive a reference to the session that was added. Each session can then report when it is destroyed to those same listeners. You use this for monitoring connections that close, and also to clean up resources.

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. By exposing this class as a remote object, you can make it accessible to a Flex client application; you 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 examples 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();
...
```

Custom error handling

For use cases where you return additional information to the client as part of a message exception, you can use the `extendedData` property of the flex.messaging.MessageException class. This property is a HashMap, and provides a flexible way to return additional data to the client when a failure occurs. The public BlazeDS Javadoc documentation includes documentation for the flex.messaging.MessageException class.

Note: *BlazeDS serialization provides bean serialization of any Throwable type. This gives you the option of throwing your own Throwable exceptions with getters for the properties that you send to the client.*

The following example shows a Java test class that adds extra data to an exception:

```
package errorhandling;

import java.util.HashMap;
import java.util.Map;
import flex.messaging.MessageException;

public class TestException {
```

```
public String generateMessageExceptionWithExtendedData(String extraData)
{
    MessageException me = new MessageException("Testing extendedData.");
    Map extendedData = new HashMap();
        // The method that invokes this expects an "extraData" slot in // this map.
        extendedData.put("extraData", extraData);
        me.setExtendedData(extendedData);
        me.setCode("999");
        me.setDetails("Some custom details.");
        throw me;
        }

}
```

The following example shows an ActionScript method that generates an exception with extra data:

```
<?xml version="1.0"?>
<mx:Application
        xmlns:mx="http://www.adobe.com/2006/mxml"
        creationComplete="run()">

    <mx:RemoteObject
        destination="myDest"
        id="myException"
        fault="onServiceFault(event)"
/>
    <mx:Script>
    <![CDATA[
        import mx.rpc.events.*;
        import mx.messaging.messages.*;

        public var mydata : String = "Extra data.";
        public var actualData : String;

        public function onServiceFault(event:FaultEvent):void {
            var errorMessage:ErrorMessage = ErrorMessage(event.message);
            if (errorMessage.extendedData != null)
                actualData = ErrorMessage(event.message).extendedData.extraData;
        // else a fault was generated that may not have originated in a server
        // error or the server error did not contain additional information.
        }

        public function run_exception():void {
            var call : Object =
                myException.generateMessageExceptionWithExtendedData(mydata);
        }

        public function run():void {
            run_exception();
        }
    ]]>
    </mx:Script>
</mx:Application>
```

Part 2: RPC Components

Understanding RPC Components 24
Creating RPC Clients 30
Configuring RPC Services on the Server. 65
Extending Applications with Factories 87

Chapter 4: Understanding RPC Components

Adobe® Flex™ remote procedure call (RPC) component are based on a service-oriented architecture (SOA). RPC components let you interact with server-side RPC services to provide data to your applications.

You can access data through HTTP (HTTP services), SOAP (web services), or Java objects (remote object services). Another common name for an HTTP 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.

In a typical Flex application, an RPC component sends data as input to one or more RPC services. When an RPC service executes, it returns its results data to the RPC component that made the request.

Topics

RPC components. 24
Comparing the RPC capability to other technologies 28

RPC components

You can use Flex RPC components in MXML or ActionScript to work with three types of RPC services: remote object services, web services, and HTTP services. When you use BlazeDS, you also contact *destinations*, which are RPC services that have corresponding server-side configurations that provide server-proxied access.

The following diagram provides a simplified view of how RPC components interact with RPC services:

Client tier
User interface
HTTPService component - Send and receive data using HTTP GET or POST
WebService component - Interact with web services
RemoteObject component - Interact with Java objects
Request
Result
RPC service implementations: Java object, web service, other
Resource tier
Server tier

The following list describes some key considerations when you are creating an application that uses BlazeDS to call an RPC service:

1 What is the best type of service to use? For more information, see "RemoteObject components" on page 25, "WebService components" on page 26, and "HTTPService components" on page 26.

2 What is the best way to pass data to a service? For more information, see "Calling a service" on page 36.

3 How do you want to handle data results from a service? For more information, see "Handling service results" on page 45.

4 How can you debug your code that uses RPC components? For more information, see "Server-side service logging" on page 372.

5 What security measures can or should you implement? For more information, see "Securing destinations" on page 153.

RemoteObject components

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 7.0.2 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.

You can 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 source path.

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 "Creating RPC Clients" on page 30.

WebService components

WebService components let you access *web services*, which are software modules with methods, 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's 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 to the Flex web application's source path.

For more information about using WebService components, see "Creating RPC Clients" on page 30.

HTTPService components

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 POSTs.

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.

You can 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.

Example: An RPC component

The following example shows MXML code for a RemoteObject component that connects to a Remoting Service 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. The value in the curly braces ({ }) in the TextArea control is a binding expression that copies service results data into the `text` property of the TextArea control:

```
<?xml version="1.0"?>
<!-- fds\rpc\RPCIntroExample1.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Connect to a service destination.-->
    <mx:RemoteObject id="remoteService" destination="myRO"/>

    <!-- 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 text="{remoteService.getData.lastResult.prop1}"/>
</mx:Application>
```

The following example shows the corresponding Remoting Service destination that is defined in the remoting-config.xml file on the server. The `source` value is a Java class that is in the web application classpath.

```
<destination id="census">
    <properties>
        <source>flex.samples.myRO</source>
    </properties>
</destination>
```

For more information, see “Creating RPC Clients” on page 30.

Handling results of RPC operations

The Example: An RPC component topic shows how you can use data binding with the result of an asynchronous operation to display the result automatically when it is received from the server. This approach prevents you from having to write code to handle the result. Instead, when the `remoteService.getData()` operation updates its `lastResult` property, the TextArea component is notified through a PropertyChangeEvent. It then updates its display automatically. However, in some cases, you do need to handle asynchronous results. For these cases, note that calling `remoteService.getData(inputText.text)` returns an AsyncToken object that you can use to be notified of the returned result, as the following ActionScript code snippet shows:

```
import mx.rpc.AsyncToken;
import mx.rpc.AsyncResponder;
….

var token:AsyncToken = remoteService.getData(inputText.text);

// you can set additional properties on the token
// to pass data to your handlers …
token.info =

token.addResponder(new AsyncResponder(myResultHandler, myFaultHandler, token));

…..
private function myResultHandler(event:Object, token:AsyncToken):void{
var resultEvent:ResultEvent = ResultEvent(event);
//… Include result handler code here
}
private function myFaultHandler(event:Object, token:AsyncToken):void{
var faultEvent:FaultEvent = FaultEvent(event);
// Include fault handler code here.
}
...
```

When the remote service returns a valid result, the `myResultHandler()` method is called with the ResultEvent event passed in. If the remote service fails, the `myFaultHandler()` method is called. Note 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 might come back before the token is assigned. This is not a problem in ActionScript because the remote call cannot be initiated until the currently executing code finishes.

Alternatively, you can listen for ResultEvent and FaultEvent events at the service level. This approach is similar to using the token, but all remote calls made with the same service use the same result and fault handlers.

Comparing the RPC capability to 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"?>
<!-- fds\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 user's web 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, you use an HTTPService, a WebService, or a RemoteObject component to call a server-side object that returns results from a data source.

Chapter 5: Creating RPC Clients

Adobe® Flex™ supports a service-oriented architecture in which a Flex application interacts with remote data sources by calling several types of services and receiving responses from the services. In a typical Flex application, client-side RPC components send asynchronous requests to remote services, which return result data to the client-side components.

For introductory information about RPC components, see "Understanding RPC Components" on page 24.

Topics

Declaring an RPC component 30
Configuring a destination 33
Calling a service 36
Setting properties for RemoteObject methods or WebService operations 43
Handling service results 45
Using a service with binding, validation, and event listeners 55
Handling asynchronous calls to services 56
Using capabilities specific to RemoteObject components 58
Using capabilities specific to WebService components 60

Declaring an RPC component

You can declare RPC components in MXML or ActionScript to connect to RPC services. If you are not using Adobe BlazeDS, you can use a WebService or HTTPService component to contact an RPC service directly.

You can use a RemoteObject component to call methods on a ColdFusion component or Java class. 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/

If you are using BlazeDS, you have the option of connecting to RPC services directly by specifying URLs in ActionScript or connecting to destinations defined in the services-config.xml file or a file that it includes by reference. 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. A destination for a RemoteObject component is a Java object defined as a Remoting Service destination. A destination for an HTTPService component is a JSP page or another resource accessed over HTTP. A destination for a WebService component is a SOAP-compliant web service. You configure HTTP services and web services as Proxy Service destinations. The following example shows a Proxy Service destination definition for an HTTP service in the proxy-config.xml file:

```
<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>
```

One major difference between Remoting Service destinations and Proxy Service destinations is that you always host the remote Java object in your BlazeDS web application, while you contact a Proxy Service destination through a URL that could be remote or local to the BlazeDS web application. A BlazeDS developer is responsible for writing and compiling the remote object Java class and adding it to the web application classpath by placing in the classes or lib directory. You can use any plain old Java object (POJO) that is available in the web application classpath as a Remoting Service destination. This technique requires that your class have a zero-argument constructor so that BlazeDS can constuct an instance. You can also use a factory to integrate BlazeDS with a component namespace of components that already exist, or configure BlazeDS to invoke components found in the ServletContext or HttpSession.

The following example shows a Remoting Service destination definition the remoting-config.xml file. The source element specifies the fully qualified name of a class in the web application's classpath.

```
<destination id="census">
    <properties>
        <source>flex.samples.census.CensusService</source>
    </properties>
```

</destination>

The following examples 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) {
            }
        }

        System.out.println("Service execution time: " + (System.currentTimeMillis() -
startTime));

        return list;
    }
}
```

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.

Optionally, you can use an HTTPService component or a WebService component to connect to a default destination.

This section discusses the basics of declaring a service connection in MXML and configuring a destination. For information about calling a service in MXML and ActionScript, see “Calling a service” on page 36. For information about configuring destinations, see “Configuring RPC Services on the Server” on page 65.

Using an RPC component with a server-side destination

An RPC component's `destination` property references a BlazeDS destination configured in the services-config.xml file or a file that it includes by reference. 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. For more information about configuring destinations, see “Configuring RPC Services on the Server” on page 65.

To declare a connection to a destination in an MXML tag, you set an `id` property and a `destination` property in one of the three RPC service tags. The `id` property is required for calling the services and handling service results.

Note: *When you compile your Flex application, and you are using server-side destinations defined in a configuration file, you must specify the location of the services-config.xml compile to the compiler or Flex Builder. The channel URLs from that file are compiled into the SWF file. You can view this information in the* `xml` *property of the mx.messaging.config.ServerConfig class. If you make changes to services-config.xml or configuration files that it includes by reference, you must recompile your application for these to take effect.*

The following examples show simple RemoteObject, HTTPService, and WebService component declarations in MXML tags:

```
<?xml version="1.0"?>
<!-- fds\rpc\RPCServerMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <!-- Declare a RemoteObject component. -->
    <mx:RemoteObject
```

```
        id="employeeRO"
        destination="SalaryManager"/>

    <!-- Declare an HTTPService component. -->
    <mx:HTTPService
        id="employeeHTTP"
        destination="SalaryManager"
        useProxy="true"/>

    <!-- Declare a WebService component. -->
    <mx:WebService
        id="employeeWS"
        destination="SalaryManager"
        useProxy="true"/>
</mx:Application>
```

The following examples show simple RemoteObject, HTTPService, and WebService component declarations in ActionScript. You place the code in these examples inside ActionScript methods:

```
<?xml version="1.0"?>
<!-- fds\rpc\RPCNoServerAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            // Import required packages.
            import mx.rpc.remoting.RemoteObject;
            import mx.rpc.http.HTTPService;
            import mx.rpc.soap.WebService;
            private var employeeRO:RemoteObject;
            private var employeeHTTP:HTTPService;
            private var employeeWS:WebService;

            public function declareServices():void{
                // Create a RemoteObject component.
                employeeRO = new RemoteObject();
                employeeRO.destination = "SalaryManager";
                // Create an HTTPService component.
                employeeHTTP = new HTTPService();
                employeeHTTP.destination = "SalaryManager";
                // Create a WebService component.
                employeeWS = new WebService();
                employeeWS.destination = "SalaryManager";
            }
        ]]>
    </mx:Script>

</mx:Application>
```

Configuring a destination

You configure a destination in a service definition in the services-config.xml file or a file that it includes by reference. A destination specifies the RPC service class or URL, the transport channel to use, the adapter with which to access the service, and settings for securing the destination.

You configure RemoteObject destinations in the Remoting Service definition in the services-config.xml file or a file that it includes by reference, such as the remoting-config.xml file. You configure HTTP service and web service destinations in the Proxy Service definition in the services-config.xml file or a file that it includes by reference, such as the proxy-config.xml file.

The Proxy Service and its adapters provide functionality that lets applications access HTTP services and web services on different domains. Additionally, the Proxy Service lets you limit access to specific URLs and URL patterns, and provide security. When you do not have BlazeDS or do not require the functionality provided by the Proxy Service, you can bypass it. You bypass the proxy by setting an HTTPService or WebService component's `useProxy` property to `false`, which is the default value. For more information on the Proxy Service, see “Configuring RPC Services on the Server” on page 65.

Basic authentication relies on standard J2EE basic authentication from the web application container, and you configure aspects of it in the services-config.xml file and the web.xml file of your Flex web application. You configure custom authentication entirely in services-config.xml file or a file that it includes by reference. For more information about securing service destinations, see “Configuring RPC Services on the Server” on page 65.

The following example shows a basic server-side configuration for a Remoting Service in the services-config.xml file or a file that it includes by reference, such as the remoting-config.xml file. RemoteObject components connect to Remoting Service destinations. The configuration of HTTP services and web services is very similar to this example.

```
<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>
```

This Remoting Service destination uses 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 about message channels, see “Securing destinations” on page 153.

Using a service without server-side configuration

You can use RPC components to connect to HTTP services and web services without configuring BlazeDS destinations. To do so, you set an HTTPService component's `url` property or a WebService component's `wsdl` property instead of setting the component's `destination` property.

When an RPC component's `useProxy` property is set to `false`, the default value, the component communicates directly with the RPC service based on the `url` or `wsdl` property value. If you are not using BlazeDS, the `useProxy` property value must be `false`. Connecting to a service in this manner requires that at least one of the following is true:

- The RPC service is in the same domain as your Flex application.
- A crossdomain.xml (cross-domain policy) file that allows access from your application's domain is installed on the web server hosting the RPC service.

When you set an HTTPService or WebService component's `useProxy` property to `true` and set the `url` or `wsdl` property and do not set the `destination` property, the component uses a default destination that is configured in the `proxy-service` section of the services-config.xml file or a file that it includes by reference, such as the proxy-config.xml file. This requires BlazeDS, and the `defaultHTTP` or `defaultHTTPS` destination in the configuration file must contain a URL pattern that matches the value of your component's `url` or `wsdl` property, respectively. The defaultHTTPS destination is used when your `url` or `wsdl` property specifies an HTTPS URL.

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; the components use the default `useProxy` value of `false` and contact the services directly, as is required when you are not using BlazeDS.

```
<?xml version="1.0"?>
<!-- fds\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>
```

The following examples show ActionScript code for declaring HTTPService and WebService components with no corresponding server-side configuration. In these examples, the `useProxy` property is not set explicitly; the components use the default `useProxy` value of `false` and contact the services directly, as is required when you are not using BlazeDS.

```
<?xml version="1.0"?>
<!-- fds\rpc\RPCNoServerAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            // Import required packages.
            import mx.rpc.http.HTTPService;
            import mx.rpc.soap.WebService;

            private var yahoo_web_search:mx.rpc.http.HTTPService;

            private var macr_news:mx.rpc.soap.WebService;

            public function declareServices():void {
            // Create an HTTPService component.
            yahoo_web_search = new HTTPService();
            yahoo_web_search.url =
                "http://api.search.yahoo.com/WebSearchService/V1/webSearch";
            // Create a WebService component.
            macr_news = new WebService();
            macr_news.wsdl =
                "http://ws.invesbot.com/companysearch.asmx?wsdl";
            }
        ]]>
    </mx:Script>

</mx:Application>
```

Calling a service

You can declare an RPC component and call the service in MXML or ActionScript. Regardless of the source of input data, calls to a service are asynchronous and require an ActionScript method, called an event listener, which is called after the RPC call is completed.

The two general categories of events are user events and system events. *User events* occur as a result of user interaction with the application; for example, when a user clicks a Button control, a user event occurs. *System events* occur as a result of systematic code execution. For information about events, see the Flex Help Resource Center.

Flex provides two ways to call a service: *explicit parameter passing* and *parameter binding*. You can use explicit parameter passing when you declare an RPC component in MXML or ActionScript. Parameter binding is only available when you declare an RPC component in MXML.

You can use an RPC component's `requestTimeout` property to define the number of seconds to allow a request to remain outstanding before timing it out. The `requestTimeout` property is available on RemoteObject, HTTPService, and WebService components.

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. You cannot use Flex data validators automatically in combination with explicit parameter passing.

Explicit parameter passing with RemoteObject and WebService components

The way you use explicit parameter passing with RemoteObject and WebService components is very similar. 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. Simple event listeners handle the service-level result and fault events. This example show sa `destination` property, but the way you call a service is the same whether you connect to the service directly or go through a destination.

```
<?xml version="1.0"?>
<!-- fds\rpc\RPCParamPassing.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    verticalGap="10">
    <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>
```

Explicit parameter passing with HTTPService tags

Explicit parameter passing with HTTPService components is different than it is with RemoteObject and WebService components. You always use an HTTPService component's `send()` method to call a service. This is different from RemoteObject and WebService components, on which you call methods that are client-side versions of the methods or operations of the RPC service.

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

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 examples show two ways to call an HTTP service using the `send()` method with a parameter. The second example also shows how to call the `cancel()` method to cancel an HTTP service call.

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

    <mx:Script>
        <![CDATA[
            public function callService():void {
                // Cancel all previous pending calls.
                myService.cancel();

                var params:Object = new Object();
                params.param1 = 'val1';
                myService.send(params);
            }
        ]]>
    </mx:Script>

    <mx:HTTPService
        id="myService"
        destination="Dest"
        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>
```

Using parameter binding

Parameter binding lets you copy data from user-interface controls or models to request parameters. Parameter binding is only available for RPC components that you declare in MXML. You can apply validators to parameter values before submitting requests to services. For more information about data binding, data models, and data validation, see the Flex Help Resource Center.

When you use parameter binding, you declare one of the following depending on what type of RPC service component you are using:

- RemoteObject method parameter tags nested in an `<mx:arguments>` tag under an `<mx:method>` tag
- HTTPService parameter tags nested in an `<mx:request>` tag
- WebService operation parameter tags nested in an `<mx:request>` tag under an `<mx:operation>` tag. You use the `send()` method to send the request

Parameter binding with RemoteObject components

When you use parameter binding with RemoteObject components, you always declare methods in a RemoteObject component's `<mx:method>` tag.

An `<mx:method>` tag can contain an `<mx:arguments>` tag that contains child tags for the method parameters. The `name` property of an `<mx:method>` tag must match one of the service's method names. The order of the argument tags must match the order of the service's method parameters. You can name argument tags to match the actual names of the corresponding method parameters as closely as possible, but this is not necessary.

Note: *If argument tags inside an* `<mx:arguments>` *tag have the same name, service calls fail if the remote method is not expecting an Array as the only input source. There is no warning about this when the application is compiled.*

You can bind data to a RemoteObject component's method parameters. You reference the tag names of the parameters for data binding and validation.

The following example shows a method with two parameters bound to the text properties of TextInput controls. A PhoneNumberValidator validator is assigned to `arg1`, which is the name of the first argument tag. This example shows a `destination` property on the RemoteObject component, but the way you call a service is the same whether you connect to the service directly or go through a destination.

```
<?xml version="1.0"?>
<!-- fds\rpc\ROParamBind1.mxml. Compiles -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:RemoteObject
        id="ro"
        destination="roDest" >

        <mx:method name="setData">
            <mx:arguments>
                <arg1>{text1.text}</arg1>
                <arg2>{text2.text}</arg2>
            </mx:arguments>
        </mx:method>
    </mx:RemoteObject>
    <mx:TextInput id="text1"/>
    <mx:TextInput id="text2"/>

    <mx:PhoneNumberValidator source="{ro.setData.arguments}" property="arg1"/>
</mx:Application>
```

Flex sends the argument tag values to the method in the order that the MXML tags specify.

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"?>
<!-- fds\rpc\ROParamBind2.mxml -->
<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: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:ArrayCollection id="employeeAC"
        source="{ArrayUtil.toArray(employeeRO.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="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, you can 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.

Parameter binding with HTTPService components

When an HTTP service takes query parameters, you can declare them as child tags of an `<mx:request>` tag. The names of the tags must match the names of the query parameters that the service expects.

The following example uses parameter binding in an HTTPService component's `<mx:request>` tag to bind the data of a selected ComboBox item to the `employeeSrv` request when the user clicks a Button control. When you use parameter binding, you call a service by using the `send()` method with no parameters. This example shows a `url` property on the HTTPService component, but the way you call a service is the same whether you connect to the service directly or go through a destination.

```
<?xml version="1.0"?>
<!-- fds\rpc\HttpServiceParamBind.mxml. Compiles -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" verticalGap="20">
    <mx:Script>
        <![CDATA[
            import mx.utils.ArrayUtil;
        ]]>
    </mx:Script>

    <mx:HTTPService
        id="employeeSrv"
        url="employees.jsp">
        <mx:request>
            <deptId>{dept.selectedItem.data}</deptId>
        </mx:request>
    </mx:HTTPService>
    <mx:ArrayCollection
        id="employeeAC"
        source=
            "{ArrayUtil.toArray(employeeSrv.lastResult.employees.employee)}"/>
    <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="{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, you can use the `toArray()` method of the mx.utils.ArrayUtil class to convert it to an Array, as the previous example shows. 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.

Parameter binding with WebService components

When you use parameter binding with a WebService component, you always declare operations in the WebService component's `<mx:operation>` tags. An `<mx:operation>` tag can contain an `<mx:request>` tag that contains the XML nodes that the operation expects. The `name` property of an `<mx:operation>` tag must match one of the web service operation names.

You can bind data to parameters of web service operations. You reference the tag names of the parameters for data binding and validation.

The following example uses parameter binding in a WebService component's `<mx:operation>` tag to bind the data of a selected ComboBox item to the `employeeWS.getList` operation when the user clicks a Button control. The `<deptId>` tag corresponds directly to the `getList` operation's `deptId` parameter. When you use parameter binding, you call a service by using the `send()` method with no parameters. This example shows a `destination` property on the WebService component, but the way you call a service is the same whether you connect to the service directly or go through a destination.

```
<?xml version="1.0"?>
<!-- fds\rpc\WebServiceParamBind.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" verticalGap="10">
    <mx:Script>
        <![CDATA[
            import mx.utils.ArrayUtil;
            import mx.controls.Alert;
        ]]>
    </mx:Script>

    <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:ArrayCollection
        id="employeeAC"
        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="{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 also manually specify an entire SOAP request body in XML with all of the correct namespace information defined in an `<mx:request>` tag. To do this, you must set the value of the `format` attribute of the `<mx:request>` tag to `xml`, as the following example shows:

```
<?xml version="1.0"?>
<!-- fds\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>
```

Setting properties for RemoteObject methods or WebService operations

You can set the following properties on a RemoteObject component's `<mx:method>` tag or a WebService component's `<mx:operation>` tag. The `name` property is the only property that is required. The following table describes the properties:

Property	Description
`concurrency`	Value that 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 are permitted: • `multiple` Existing requests are not cancelled, and the developer is responsible for ensuring the consistency of returned data by carefully managing the event stream. This is the default value. • `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, or the pendingCall object. HTTP requests are sent to the server and get processed on the server side. However, the result is ignored when a request is cancelled (requests are cancelled when you use the `single` or `last` value). The `last` request is not necessarily the last one that the server receives over HTTP.
`fault`	ActionScript code that runs when an error occurs. The fault is passed as an event parameter.
`name`	(Required) Name of the operation or method to call.
`result`	ActionScript code that runs when a lastResult object is available. The result object is passed as an event parameter.

Calling services in ActionScript

You always use explicit parameter passing when you call a service in ActionScript. You can call RemoteObject, WebService, and HTTPService components in ActionScript.

Calling RemoteObject components in ActionScript

The following ActionScript example is equivalent to the MXML example in "Explicit parameter passing with RemoteObject and WebService components" on page 36. Calling the `useRemoteObject()` method declares the service, sets the destination, sets up result and fault event listeners, and calls the service's `getList()` method.

```
<?xml version="1.0"?>
<!-- fds\rpc\ROInAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.rpc.remoting.RemoteObject;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;

            [Bindable]
            public var empList:Object;
            public var employeeRO:RemoteObject;

            public function useRemoteObject(intArg:int, strArg:String):void {
                employeeRO = new RemoteObject();
                employeeRO.destination = "SalaryManager";
                employeeRO.getList.addEventListener("result", getListResultHandler);
                employeeRO.addEventListener("fault", faultHandler);
```

```
                employeeRO.getList(deptComboBox.selectedItem.data);
            }

            public function getListResultHandler(event:ResultEvent):void {
                 // Do something
                empList=event.result;
            }

            public function faultHandler (event:FaultEvent):void {
             // Deal with event.fault.faultString, etc.
                Alert.show(event.fault.faultString, 'Error');
            }
        ]]>
    </mx:Script>
    <mx:ComboBox id="deptComboBox"/>
</mx:Application>
```

Calling web services in ActionScript

The following example shows a web service call in an ActionScript script block. Calling the `useWebService()` method declares the service, sets the destination, fetches the WSDL document, and calls the service's `echoArgs()` method. Notice that you must call the `WebService.loadWSDL()` method when you declare a WebService component in ActionScript.

```
<?xml version="1.0"?>
<!-- fds\rpc\WebServiceInAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
        import mx.rpc.soap.WebService;
        import mx.rpc.events.ResultEvent;
        import mx.rpc.events.FaultEvent;
        private var ws:WebService;
        public function useWebService(intArg:int, strArg:String):void {
            ws = new WebService();
            ws.destination = "echoArgService";
            ws.echoArgs.addEventListener("result", echoResultHandler);
            ws.addEventListener("fault", faultHandler);
            ws.loadWSDL();
            ws.echoArgs(intArg, strArg);
        }

        public function echoResultHandler(event:ResultEvent):void {
            var retStr:String = event.result.echoStr;
            var retInt:int = event.result.echoInt;
         //Do something.
        }

        public function faultHandler(event:FaultEvent):void {
      //deal with event.fault.faultString, etc
        }
        ]]>
    </mx:Script>
</mx:Application>
```

Calling HTTP services in ActionScript

The following example shows an HTTP service call in an ActionScript script block. Calling the `useHTTPService()` method declares the service, sets the destination, sets up result and fault event listeners, and calls the service's `send()` method.

```
<?xml version="1.0"?>
<!-- fds\rpc\HttpServiceInAS.mxml. Compiles -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" verticalGap="10">
    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.rpc.http.HTTPService;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;

            private var service:HTTPService

            public function useHttpService(parameters:Object):void {
                service = new HTTPService();
                service.destination = "sampleDestination";
                service.method = "POST";
                service.addEventListener("result", httpResult);
                service.addEventListener("fault", httpFault);
                service.send(parameters);
            }

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

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

Handling service results

After an RPC component calls a service, the data that the service returns is placed in a lastResult object. By default, the `resultFormat` property value of HTTPService components and WebService component operations 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, you must 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 case insensitive, so it internally uppercases all of its data. Keep this in mind when consuming a ColdFusion web service.*

Handling results as XML with the e4x result format

You can set the `resultFormat` property value of HTTPService components and WebService operations to `e4x` to create a lastResult object of type XML. You can access the lastResult object by using ECMAScript for XML (E4X) expressions. Using a `resultFormat` of `e4x` is the preferred way to work with XML, but you can also set the `resultFormat` property to `xml` to create a lastResult object 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 Flex Language 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 51.

***Note:** If you want to use E4X syntax on service results, you must 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 may 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 gets 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>
...
</soap:Body>
...
```

Because this soap:Body contains namespace information, if you set the `resultFormat` property of the WebService operation to `e4x`, you must create a namespace object for the http://ws.invesbot.com/ namespace. The following example shows an application that does that:

```
<?xml version="1.0"?>
<!-- fds\rpc\WebServiceE4XResult1.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*"
    pageTitle="Test" >
    <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 var for a namespace and access it in a binding to the service result, as the following example shows:

```
<?xml version="1.0"?>
<!-- fds\rpc\WebServiceE4XResult2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*"
    pageTitle="Test" >
    <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

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

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

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>
    </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 an 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>
```

Binding a service result to other objects

You can bind properties of an RPC component's lastResult object to the properties of other objects, including user-interface components and data models. The lastResult is the result object from the last invocation. Whenever a service request executes, the lastResult is updated and any associated bindings are also updated.

In the following example, two properties of the lastResult object, `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}"/>
```

You can bind a lastResult object to the `source` property of an ArrayCollection object. When you use an HTTPS-ervice or WebService component, you can bind a lastResult to an XMLListCollection object when the HTTPService component's or WebService operation's `resultFormat` property is set to `e4x`. There is no `resultFormat` property on RemoteObject methods. You can then bind the ArrayCollection object or XMLListCollection object to a complex property of a user-interface component, such as a List, ComboBox, or DataGrid control.

You can use the ArrayCollection or XMLListCollection API to work with the data. When using an XMLListCollection object, you can use ECMAScript for XML (E4X) expressions to work with the data.

Binding a result to an ArrayCollection object

In the following example, a service lastResult object, `employeeWS.getList.lastResult`, is bound to the `source` property of an ArrayCollection object. The ArrayCollection object is bound to the `dataProvider` property of a DataGrid control that displays employees' names, phone numbers, and e-mail addresses.

```
<?xml version="1.0"?>
<!-- fds\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, as the previous example shows. 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

In the following example, a service lastResult object, `employeeWS.getList.lastResult`, is bound to the `source` property of an XMLListCollection object. The XMLListCollection object is bound to the `dataProvider` property of a DataGrid control that displays employees' names, phone numbers, and e-mail addresses.

Note: *To bind service results to an XMLListCollection, you must set the `resultFormat` property of your HTTPService or WebService component to `e4x`. The default value of this property is `object`.*

```
<?xml version="1.0"?>
<!-- fds\rpc\BindResultXMLListCollection.mxml. Warnings on mx:Object -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" verticalGap="10">
    <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 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 very 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, which specifies how the rest of the message should be processed. To best handle results from this type of web service, you set the `resultFormat` property of a Flex WebService operation to `object`. You can optionally set the WebService operation's `resultFormat` property to `e4x`, but 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, for example, to a DataGrid control.

When you set the `resultFormat` property of a Flex WebService operation to `object`, 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 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 backend 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;
    }
}
```

Handling result and fault events

When a service call is completed, the RemoteObject method, WebService operation, or HTTPService component dispatches a result event or a fault event. A *result event* indicates that the result is available. A *fault event* indicates that an error occurred. The result event acts as a trigger to update properties that are bound to the lastResult. You can handle fault and result events explicitly by adding event listeners to RemoteObject methods or WebService operations. For an HTTPService component, you specify result and fault event listeners on the component itself because an HTTPService component does not have multiple operations or methods.

When you do not specify event listeners for result or fault events on a RemoteObject method or a WebService operation, the events are passed to the component level; you can specify component-level result and fault event listeners.

In the following MXML example, the `result` and `fault` events of a WebService operation specify event listeners; the `fault` event of the WebService component also specifies an event listener:

```
<?xml version="1.0"?>
<!-- fds\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 showErrorDialog(event:FaultEvent):void {
                // Handle operation fault.
                Alert.show(event.fault.faultString, "Error");
            }
            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");
            }
            public function log(event:ResultEvent):void {
                // Handle result.
            }
        ]]>
    </mx:Script>
    <mx:WebService id="WeatherService" destination="wsDest"
        fault="defaultFault(event)">
        <mx:operation name="GetWeather"
            fault="showErrorDialog(event)"
            result="log(event)">
            <mx:request>
                <ZipCode>{myZip.text}</ZipCode>
            </mx:request>
        </mx:operation>
    </mx:WebService>
    <mx:TextInput id="myZip"/>
</mx:Application>
```

In the following ActionScript example, a result event listener is added to a WebService operation; a fault event listener is added to the WebService component:

```
<?xml version="1.0"?>
<!-- fds\rpc\RPCResultFaultAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.rpc.soap.WebService;
            import mx.rpc.soap.SOAPFault;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;

            private var ws:WebService;

            public function useWebService(intArg:int, strArg:String):void {
                ws = new WebService();
                ws.destination = "wsDest";
                ws.echoArgs.addEventListener("result", echoResultHandler);
                ws.addEventListener("fault", faultHandler);
```

```
                ws.loadWSDL();
                ws.echoArgs(intArg, strArg);
            }

            public function echoResultHandler(event:ResultEvent):void {
                var retStr:String = event.result.echoStr;
                var retInt:int = event.result.echoInt;
                //do something
        }

            public function faultHandler(event:FaultEvent):void {
            //deal with event.fault.faultString, etc.
                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.
                     ...
                }
            }
        ]]>
    </mx:Script>
</mx:Application>
```

You can also use the mx.rpc.events.InvokeEvent event to indicate when an RPC component request has been invoked. This is useful if operations are queued and invoked at a later time.

Using a service with binding, validation, and event listeners

You can validate data before passing it to a service, and dispatch an event when the service returns a result or a fault. The following example shows an application that validates service request data and assigns an event listener to result and fault events.

This two-tier application does the following:

1. Declares a web service.
2. Binds user-interface data to a web service request.
3. Validates a ZIP code.
4. Binds data from a web service result to a user-interface control.
5. Specifies result and fault event listeners for a WebService operation.

You can also create multitier applications that use an additional data-model layer between the user interface and the web service. For more information about data models, data binding, and validation, see the Flex Help Resource Center.

```
<?xml version="1.0"?>
<!-- fds\rpc\RPCBindValidateEvents.mxml. Compiles w destination error -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="600" height="400">

    <!-- WebService component handles web service requests and results. -->
    <mx:WebService id="WeatherService" destination="wsDest">
        <mx:operation name="GetWeather"
            fault="showErrorDialog(event.fault.faultString)" result="log();">
```

```
    <!-- The mx:request data model stores web service request data. -->
            <mx:request>
                <ZipCode>{myZipField.text}</ZipCode>
            </mx:request>
        </mx:operation>
    </mx:WebService>

    <!-- Validator validates ZIP code using the standard Zip Code validator. -->
    <mx:ZipCodeValidator
        source="{WeatherService.GetWeather.request}" property="ZipCode"
        trigger="{mybutton}" triggerEvent="click"/>
    <mx:VBox>
    <mx:TextInput id="myZipField" text="enter zip" width="80"/>

    <!-- Button triggers service request. -->
    <mx:Button id="mybutton" label="Get Weather"
        click="WeatherService.GetWeather.send();"/>

    <!-- TextArea control displays the results that the service returns. -->
        <mx:TextArea id="temp" text="The current temperature in
        {WeatherService.GetWeather.lastResult.CityShortName} is
        {WeatherService.GetWeather.lastResult.CurrentTemp}."
        height="30" width="200"/>
    </mx:VBox>
    <mx:Script>
        <![CDATA[
            public function log():void {
                // function implementation
            }

            public function showErrorDialog(error:String):void {
                // function implementation
            }
        ]]>
    </mx:Script>
</mx:Application>
```

Handling asynchronous calls to services

Because ActionScript code executes asynchronously, if you allow concurrent calls to a service, you must ensure that your code handles the results appropriately, based on the context in which the service is called. 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 very simple application, user-interface components might call services directly. In applications that are slightly larger, business objects might call services. In still larger applications, business objects might 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.

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 service call. When you use the default `concurrency` value, `multiple`, you can use the call object that is 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 this call object when it is returned, and then in a result event listener you can pass back the call object as event.token. This is an implementation of the ACT design pattern that uses the call object 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 might attach simple identifiers to individual calls, 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 call object.

```
...
<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 call object 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. You must make the second service call in the result event listener of the first, as the following example shows:

```
...
<mx:WebService id="ws" destination="wsDest"...>
<mx:operation name="getCurrentSales" result="resultHandler(event.result)"/>
<mx:operation name="setForecastWithSalesInput"/>
</mx:WebService>
<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>
...
```

Using capabilities specific to RemoteObject components

You can use a RemoteObject component to call methods on a Remoting Services destination, which specifies a Java object that resides on the application server on which BlazeDS is running and is in the web application's source path.

Accessing Java objects in the source path

The RemoteObject component lets you access stateless and stateful objects that are in the BlazeDS web application's source path. You can place stand-alone class files in the web application's WEB-INF/classes directory to add them to the source path. You can place classes contained in Java Archive (JAR) files in the web application's WEB-INF/lib directory to add them to the source path. You must specify the fully qualified class name in the `source` property of a Remoting Service destination in the services-config.xml file or a file that it includes by reference, such as the remoting-config.xml file. The class also must have a no-args constructor. For information about configuring Remoting Service destinations, see "Configuring RPC Services on the Server" on page 65.

When you configure a Remoting Service destination to access stateless objects (the request scope), Flex creates a new object for each method call instead of calling methods on the same object. You can set the scope of an object to the request scope (default value), the application scope, or the session scope. Objects in the application scope are available to the web application that contains the object. Objects in the session scope are available to the entire client session.

When you configure remote object destination to access stateful objects, Flex creates the object once on the server and maintains state between method calls. You should use the request scope if storing the object in the application or session scope causes memory problems.

Accessing EJBs and other objects in JNDI

You can 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 EJB's business methods.

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 EJB's `sayHello()` method

```
...
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);
    }
...
```

Reserved method names

The Flex remoting library uses the following method names; do not use these as your own method names:

```
addHeader()
addProperty()
deleteHeader()
hasOwnProperty()
isPropertyEnumerable()
isPrototypeOf()
registerClass()
toLocaleString()
toString()
unwatch()
valueOf()
watch()
```

You also should not begin method names with an underscore (_) character.

RemoteObject and WebService Operations (methods) are usually accessible by simply naming them after the service variable. However, if your Operation name happens to match a defined method on the service, you can use the following method in ActionScript on a RemoteObject or WebService component to return the Operation of the given name:

```
public function getOperation(name:String):Operation
```

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), WSDL 1.1 rpc-encoded, 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: choice, union, default, list, or group. Flex also does not support the following data types: duration, gMonth, gYear, gYearMonth, gDay, gMonthDay, Name, Qname, NCName, anyURI, or language. 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 applications can access over HTTP. Flash applications are only allowed HTTP access to resources on the same domain and by the same protocol from which they were served. This 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 cross-domain.xml (cross-domain policy) file that allows access from your application's domain must be installed on the web server hosting the RPC service.

Reading WSDL documents

You can 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.

A WSDL document contains the tags described in the following table:

Tag	Description
`<binding>`	Specifies the protocol that clients, such as Flex applications, use to communicate with a web service. Bindings exist for SOAP, HTTP GET, HTTP POST, and MIME. Flex supports the SOAP binding only.
`<fault>`	Specifies an error value that is returned as a result of a problem processing a message.
`<input>`	Specifies a message that a client, such as a Flex application, sends to a web service.
`<message>`	Defines the data that a web service operation transfers.

Tag	Description
<operation>	Defines a combination of <input>, <output>, and <fault> tags.
<output>	Specifies a message that the web service sends to a web service client, such as a Flex application.
<port>	Specifies a web service endpoint, which specifies an association between a binding and a network address.
<portType>	Defines one or more operations that a web service provides.
<service>	Defines a collection of <port> tags. Each service maps to one <portType> tag and specifies different ways to access the operations in that <portType> tag.
<types>	Defines data types that a web service's messages use.

RPC-oriented operations and document-oriented operations

A WSDL file can specify either remote procedure call (RPC) oriented 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

Flex uses Java server sessions to maintain the state of web service endpoints that use cookies to store session information. This capability acts as an intermediary between Flex applications and web services. It adds an endpoint's identity 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; it is not supported for destinations that use the RTMP channel when using the Proxy Service.

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. This section describes how to add SOAP headers to web service requests, clear SOAP headers, and get SOAP headers that are contained in web service results.

Adding SOAP headers to web service requests

Some web services require that you pass along a SOAP header when you call an operation.

You can add a SOAP header to all web service operations or individual operations by calling a WebService or Operation object's `addHeader()` method or `addSimpleHeader()` method in an event listener function.

When you use the `addHeader()` method, you first must 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)
```

To create the QName object in the first parameter of the `SOAPHeader()` method, you use the following constructor:

```
QName(uri:String, localName:String)
```

The `content` parameter of the `SOAPHeader()` constructor is a set of name-value pairs based on the following format:

```
{name1:value1, name2:value2}
```

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 can be a String if it is a simple value, an Object that will undergo basic XML encoding, or XML if you want to specify the header XML yourself.

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 an event listener function called `headers`, and the event listener is assigned in the `load` property of an `<mx:WebService>` tag:

```
<?xml version="1.0"?>
<!-- fds\rpc\WebServiceAddHeader.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" height="600">
    <mx:WebService id="ws" destination="wsDest" load="headers();"/>
    <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:Application>
```

Clearing SOAP headers

You use a WebService or Operation object's `clearHeaders()` method to remove SOAP headers that you added to the object, as the following example shows for a WebService object. You must call `clearHeaders()` at the level (WebService or Operation) where the header was added.

```
<?xml version="1.0"?>
<!-- fds\rpc\WebServiceClearHeader.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" height="600" >

    <!-- The value of the destination property is for demonstration only and is not a real
destination. -->

    <mx:WebService id="ws" destination="wsDest" load="headers();"/>

    <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: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 service's business operations. Before calling the business operations, you must 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 passes that variable into a function to change 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, might 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 61.

Chapter 6: Configuring RPC Services on the Server

You connect to a remote procedure call (RPC) service from an Adobe® Flex™ client application by declaring an RPC component in MXML or ActionScript. The RPC component can specify the actual URL or WSDL URL of an HTTP service or web service, or the name of an BlazeDS destination definition. A *destination* is the actual server-side service or object that you call. You configure destinations in the services-config.xml file or a file that it includes by reference.

For information about specifying a destination in an RPC component, see "Creating RPC Clients" on page 30.

Topics

Destination configuration 65
Configuring destination properties 67
Configuring the Proxy Service 69

Destination configuration

An RPC service *destination* is the object or service that you connect to using an `<mx:RemoteObject>`, `<mx:WebService>`, or `<mx:HTTPService>` tag or the corresponding ActionScript API. Configuring destinations is the most common task that you perform in the services-config.xml file or a files that it includes by reference. You define remote object destinations in *Remoting Service* definitions; by convention, these are in the remoting-config.xml file. You define web service and HTTP service destinations in *Proxy Service* definitions; by convention, these are in the proxy-config.xml file.

When you configure a destination, you reference one or more messaging channels that you can use to contact the corresponding server-side object or service. You also reference one or more adapters. An *adapter* is server-side code that interacts directly with the object or service. You can configure default adapters to avoid explicitly referencing them in each destination.

The following example shows a basic Remoting Service definition. The service contains a destination that references a security constraint, which is also shown. The destination uses a default adapter, `java-object`, defined at the service level.

```
<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="SampleEmployeeRO">
        <properties>
            <source>samples.explorer.EmployeeManager</source>
```

```
            <scope>application</scope>
        </properties>
        <security>
            <security-constraint ref="privileged-users"/>
        </security>
    </destination>
</service>

...
<security>
    <security-constraint id="privileged-users">
        <auth-method>Custom</auth-method>
        <roles>
            <role>privilegedusers</role>
            <role>admins</role>
        </roles>
    </security-constraint>
...
</security>
```

Message channels

A destination references one or more message channels, which are defined elsewhere in the same configuration file. Optionally, you can reference default channels at the service level in the configuration file instead of referencing them implicitly in specific destinations.

For more information about messaging channels, see "Securing destinations" on page 153.

Destination adapters

A service definition includes definitions for one or more adapters that act on a service request. For example, an HTTP service sends HTTP request messages to the Proxy Service, which uses its HTTP proxy adapter to make an HTTP request for a given URL. An adapter definition must reference an adapter class that is an implementation of flex.messaging.services.ServiceAdapter.

You can configure a default adapter to avoid explicitly referencing an adapter in each destination definition. To make an adapter into a default adapter, you set the value of its default property to `true`.

The following example shows a Java object adapter that you use with Remoting Service destinations. This adapter is configured as a default adapter.

```
...
<adapters>
    <adapter-definition id="java-object"
        class="flex.messaging.services.remoting.adapters.JavaAdapter"
        default="true"/>
</adapters>
...
```

Security

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 that are required for authorization.

You can declare a security constraint inline in a destination definition, or you can declare it globally and reference it by its `id` in a destination definition.

For more information about security, see "Securing destinations" on page 153.

Default HTTP service destination

You can configure a default destination for HTTP services in the services- config.xml file or a file that it includes by reference. This lets you use more than one HTTP service `url` property value in client-side service tags or Action-Script code, and still go through the Proxy Service for cross-domain support and security.

The default destination always has an `id` value of `defaultHTTP`. Typically, you use `dynamic-url` parameters to specify one or more URL wildcard patterns for the HTTP proxy adapter.

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

```
...
<destination id="defaultHTTP">
    <channels>
        <channel ref="my-amf"/>
        </channels>
    <properties>
            <dynamic-url>http://mysite.com/myservices/*</dynamic-url>
    </properties>
    <security>
...
    </security>
</destination>
...
```

For more information about HTTP service adapters, see “HTTP service properties” on page 68.

Configuring destination properties

To communicate with Remoting Service and Proxy Service destinations, you configure specific types of destination properties in the `properties` section of a destination definition.

Remote object properties

You use the `source` and `scope` elements of a Remoting Service destination definition to specify the Java object that the destination uses and whether it should be 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. Objects in the request scope are stateless. Objects in the application scope are available to the web application that contains the object. Objects in the session scope are available to the entire client session. The valid values are `request`, `application`, and `session`. The default value is `request`.

The following example shows a Remoting Service destination definition that contains `source` and `scope` properties:

```
...
    <destination id="SampleEmployeeRO">
        <properties>
            <source>samples.explorer.EmployeeManager</source>
            <scope>application</scope>
        </properties>
        <adapter ref="java-object"/>
    </destination>
...
```

Web service properties

You use the `wsdl` and `soap` elements to configure web service URLs. 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.

The following example shows a destination definition for a web service:

```
...
    <destination id="ContactManagerWS">
        <adapter ref="soap-proxy"/>
        <properties>
            <wsdl>{context.root}/services/ContactManagerWS?wsdl</wsdl>
            <soap>{context.root}/services/ContactManagerWS</soap>
        </properties>
    </destination>
...
```

HTTP service properties

You use the `url` and `dynamic-url` elements to configure HTTP service URLs. 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.

The following example shows a destination definition for an HTTP service:

```
...
    <destination id="samplesProxy">
        <channels>
            <channel ref="samples-amf"/>
        </channels>

        <properties>
            <dynamic-url>{context.root}/photoviewer/*</dynamic-url>
            <dynamic-url>{context.root}/blogreader/*</dynamic-url>
            <dynamic-url>{context.root}/services/*</dynamic-url>
        </properties>
    </destination>
...
```

Configuring the Proxy Service

The Proxy Service, in which you define both web service and HTTP service destinations, contains a `properties` element with child elements for configuring the Apache connection manager, self-signed certificates for SSL, and external proxies. The following table describes these XML elements:

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`	Indicates whether to use chunked content. The default values is `false`. Flash Player does not support chunked content.
`allow-lax-ssl`	Allows self-signed certificates when using SSL, when set to `true`. Do not set the value to `true` in a production environment.
`external-proxy`	Specifies the location of an external proxy as well as a user name and a password for situations where the Proxy Service must contact an external proxy before getting access to the Internet.

The following example shows `max-connections` and `external-proxy` configurations:

```
...
<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 self-signed certificates; should not be used in production -->
        <allow-lax-ssl>true</allow-lax-ssl>

        <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>
```

Chapter 7: Serializing Data

Adobe® 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 70
Serializing between ActionScript and web services 79

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
Boolean String of `"true"` or `"false"`	java.lang.Boolean	Boolean, boolean, String
flash.utils.ByteArray	byte []	
flash.utils.IExternalizable	java.io.Externalizable	
Date	java.util.Date (formatted for Coordinated Universal Time (UTC))	java.util.Date, java.util.Calendar, java.sql.Timestamp, java.sql.Time, java.sql.Date
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 when you use `[RemoteClass]` metadata tag that specifies remote class name. Bean type must have a public no args constructor.	typed Object

ActionScript type (AMF 3)	Deserialization to Java	Supported Java type binding
undefined	null	null for Object, default values for primitives
XML	org.w3c.dom.Document	org.w3c.dom.Document
XMLDocument (legacy XML type)	org.w3c.dom.Document	org.w3c.dom.Document 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 74.

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 74.

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 74.

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 examples 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.

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

***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
`<ignore-property-errors> true </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> false </log-property-errors>`	Default value is `false`. When `true`, unexpected property errors are logged.
`<legacy-collection>false</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>false</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>false</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>false</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).
`<type-marshaller> className </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.
`<restore-references> false </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> true </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 show 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 to be special cased 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 wrap-ping).
`xsd:boolean`	Boolean Number Object	Always encoded as `true` or `false`. `Number == 1` then `true`, otherwise `false`. `Object.toString()` == "true" or "1" then `true`, other-wise `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.
`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.

XML schema type	Supported ActionScript 3 types	Notes
`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.

`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 cased and are supported only with RPC-encoded style 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 it 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.
`xsd:boolean`	Boolean	If content is "true" or "1" then `true`, otherwise `false`.
`xsd:date`	Date	If no timezone information is present, local time is assumed.
`xsd:dateTime`	Date	If no timezone 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.
`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.

XML schema type	Decoded ActionScript 3 types	Notes

`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()`.

`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 timezone 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 cased. 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 webservice 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 one 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.

Chapter 8: 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 87

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;
            }
        }

    }

}
```

Part 3: Messaging

BlazeDS Message Service 93
Creating Messaging Clients 96
Configuring Messaging on the Server 108
Serializing Data 70
Controlling Message Delivery with Adaptive Polling 118
The Ajax Client Library 124
Measuring Message Processing Performance 134

Chapter 9: BlazeDS Message Service

BlazeDS include a client-side messaging API that you use in conjunction with the server-side Message Service.

Topics

Messaging 93
Messaging architecture 94

Messaging

The BlazeDS messaging capability is based on established messaging standards and terminology. BlazeDS messaging provides a client-side API and a corresponding server-side Message Service (BlazeDS Message Service) for creating BlazeDS messaging applications. BlazeDS messaging also enables participation in Java Message Service (JMS) messaging. For more information, see "Configuring Messaging on the Server" on page 108.

You can also send messages to a ColdFusion component (CFC) by using the ColdFusion Event Gateway Adapter. For more information, see the ColdFusion documentation.

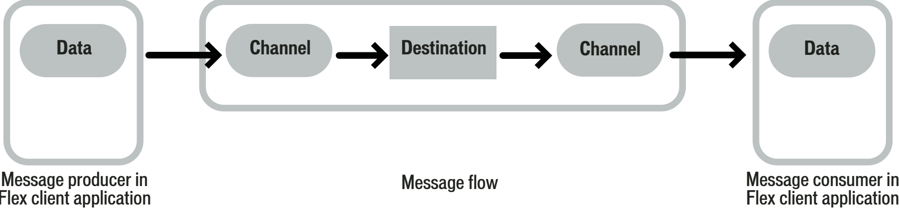
Messaging systems let separate applications communicate asynchronously as peers by passing packets of data called *messages* back and forth through a Message Service. A message usually consists of a header and a body. The header contains an identifier and routing information. The body contains application data.

Applications that send messages are called *producers*. Applications that receive messages are called *consumers*. In most messaging systems, producers and consumers do not need to know anything about each other. Producers send messages to specific message destinations, and the Message Service routes the messages to the appropriate consumers.

A *message channel* connects producers and consumers to message destinations. To send messages over a particular channel, an application connects to the message endpoint associated with the message channel. A *message endpoint* is the code responsible for encoding data into messages, and decoding messages into a format that consumers can use. In some messaging systems, an endpoint can pass decoded messages to a message broker, which routes them to appropriate destinations.

A *message adapter* is code that acts as a conduit between the BlazeDS Message Service and other messaging systems. For example, the Java Message Service (JMS) adapter is a message adapter that lets Flex applications subscribe to JMS topics and queues. This adapter lets a pure Java JMS message application share the same destination as a Flex application: Java applications can publish messages to BlazeDS destinations, and Java code can respond to messages that Flex applications send.

The following image shows the flow of data from a message producer to a message consumer. Data is encoded as a message and sent over a channel to a destination. The message is then sent over a channel to the message consumer and decoded into data that the consumer can use.



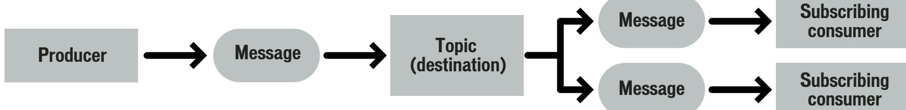
BlazeDS supports publish-subscribe messaging, which is also know as topic-based messaging. You can support point-to-point messaging, also known as queue-based messaging, between Flex clients by using a simple Action-Script-adapter-based message destination in conjunction with message headers and selectors or subtopics on the client side. For more information, see “Filtering messages with a message selector” on page 102 and “Using subtopics” on page 103.

Publish-subscribe messaging and point-to-point messaging are the two most common types of messaging that enterprise messaging systems use. You use publish-subscribe messaging for applications that require a one-to-many relationship between producers and consumers. You use point-to-point messaging for applications that require a one-to-one relationship between producers and consumers.

In BlazeDS messaging, a topic or queue is represented by a Message Service destination. 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 might use publish-subscribe messaging are auction sites, stock quote services, and other applications that require one message to be sent to many subscribers. In queue-based messaging, each message is delivered to a single consumer.

Producers publish messages to specific topics on a message server, and consumers subscribe to those topics to receive messages. Consumers can consume messages that were published to a topic only after they subscribed to the topic.

The following image shows a simple publish-subscribe message flow:



Including messaging in an application

BlazeDS provides MXML and ActionScript APIs that let you add messaging to Flex applications. You can create applications that act as producers, consumers, or both. Flex applications send messages over channels declared on the BlazeDS server to destinations also declared on the server. For more information, see “Securing destinations” on page 153.

You configure channels and destinations for the Message Service in the BlazeDS configuration file. A message destination is a remote resource; you can configure its security policy and the messaging adapters that it requires. For more information about messaging configuration, see “Configuring Messaging on the Server” on page 108.

Messaging architecture

BlazeDS messaging lets a Flex application connect to a message destination, send messages to it, and receive messages from other messaging clients. Those messaging clients can be Flex applications or other types of clients, such as Java Message Service (JMS) clients. JMS is a Java API that lets applications create, send, receive, and read messages. JMS clients can publish and subscribe to the same message destinations as Flex applications. This means that Flex applications can exchange messages with Java client applications. However, you can create a wide variety of messaging applications using just BlazeDS messaging.

Message Service

The four components of the BlazeDS Message Service are channels, destinations, producers, and consumers. When a Flex client application uses the ActionScript or MXML messaging API to publish a message to a destination, the client application sends the message to the server-side Message Service.

The Message Service provides an abstraction on top of the transport protocols that Adobe Flash Player supports, such as Action Message Format (AMF), Real-Time Messaging Protocol (RTMP), and XMLSocket. You configure the Message Service to transport messages over one or more channels. Each channel corresponds to a specific transport protocol. Within the services-config.xml file, you specify which channels to use for a specific destination.

Message channels

Flex applications can access the Message Service over several different message channels. Each channel corresponds to a specific network protocol and has a corresponding server-side endpoint. There is client-side code that corresponds to each channel. When a message arrives at an endpoint, the endpoint decodes the message and passes it to a message broker, which determines where the message should be sent. For publish-subscribe messaging scenarios, the message broker sends messages to the Message Service. The message broker also routes messages for the remote procedure call (RPC) service . The Flex client tries the channels in the order specified until an available channel is found or there are no more channels in the list.

For more information about message channels, see “Message channels” on page 109.

JMS message adapter

JMS is a Java API that lets applications create, send, receive, and read messages. BlazeDS messaging uses the JMS message adapter to interact with external messaging systems.

This adapter lets Flex applications subscribe to JMS topics and queues. It allows Flex applications to participate in existing messaging oriented middleware (MOM) systems.

The JMS message adapter allows a client to consume messages off of a JMS queue or publish messages to a JMS queue. JMS queues are point-to-point, unlike topics which are one-to-many. However, due to the administrative overhead of JMS destinations (topics or queues) and the fact that the JMS message adapter does not support the use of temporary topics or queues, the JMS message adapter is not the best choice for point-to-point messages between clients. A better choice for point-to-point messaging between Flex clients is to use a simple ActionScript-adapter-based message destination in conjunction with message headers and selectors or subtopics on the client side. For more information, see “Filtering messages with a message selector” on page 102 and “Using subtopics” on page 103.

The JMS message adapter supports the use of message headers and selectors for JMS topics, but hierarchical topics and subtopics are not supported.

Chapter 10: Creating Messaging Clients

BlazeDS includes a client-side messaging API that you use to create Flex applications that send and receive messages. You use the client-side messaging API in conjunction with the server-side Message Service. For information about configuring the server-side Message Service, see "Configuring Messaging on the Server" on page 108.

Topics

Using messaging in a Flex application 96
Working with Producer components 96
Working with Consumer components 100
Using subtopics 103
Using a pair of Producer and Consumer components in an application 106

Using messaging in a Flex application

A Flex client application uses the client-side messaging API to send messages to, and receive messages from, a server-side destination. Messages are sent to and from destinations over a protocol-specific message channel.

The two primary client-side messaging components are Producer and Consumer components. A Producer component sends messages to a server-side destination. A Consumer component subscribes to a server-side destination and receives messages that a Producer component sends to that destination. You can create Producer and Consumer components in MXML or ActionScript.

Producer and Consumer components both require a valid message destination that you configure in the services-config.xml file. For information about message destinations, see "Configuring Messaging on the Server" on page 108.

A Flex application often contains at least one pair of Producer and Consumer components. This enables each application to send messages to a destination and receive messages that other applications send to that destination. It is very easy to test this type of application by running it in different browser windows and sending messages from each of them.

Working with Producer components

You can create Producer components in MXML or ActionScript. To send a message, you create an AsyncMessage object and then call a Producer component's `send()` method to send the message to a destination.

You can also specify acknowledge and fault event handlers for a Producer component. An acknowledge event is broadcast when a destination successfully receives a message that a Producer component sends. A fault event is dispatched when a destination cannot successfully process a message due to a connection-, server-, or application-level failure.

A Producer component can send messages to destinations that use the message service with no message adapters or to destinations that use message adapters. A *message adapter* is server-side code that acts as a conduit between the Message Service and other messaging systems. For example, the Java Message Service (JMS) adapter enables Producer components to send messages to JMS topics. On the client side, the APIs for non-adapter and adapter-enabled destinations are identical. The definition for a particular destination in the BlazeDS services configuration file determines what happens when a Producer component sends messages to the destination or a Consumer component subscribes to the destination. For information about configuring destinations, see “Configuring Messaging on the Server” on page 108.

For reference information about the Producer class, see *Adobe Flex Language 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. It should also specify a `destination` that is defined in the server-side 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"?>
<!-- fds\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 messageHandler(event:MessageEvent):void {
                // Handle message event.
            }
            private function acknowledgeHandler(event:MessageAckEvent):void{
                // Handle message event.
            }
            private function faultHandler(event:MessageFaultEvent):void {
            // Handle fault event.
            }
        ]]>
    </mx:Script>
    <mx:Producer id="producer"
        destination="ChatTopicJMS"
        acknowledge="acknowledgeHandler(event)"
        fault="faultHandler(event)"/>
</mx:Application>
```

Creating a Producer component in ActionScript

You can create a Producer component in an ActionScript method. The following code shows a Producer component that is created in a method in an `<mx:Script>` tag. The import statements import the classes required to create a Producer component, create Message objects, add event listeners, and create message handlers.

```
<?xml version="1.0"?>
<!-- fds\messaging\CreateProducerAS.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 var producer:Producer;
            private function acknowledgeHandler(event:MessageAckEvent):void{
                // Handle message event.
            }
            private function faultHandler(event:MessageFaultEvent):void{
            // Handle fault event.
            }
            private function logon():void {
                producer = new Producer();
                producer.destination = "ChatTopicJMS";
               producer.addEventListener(MessageAckEvent.ACKNOWLEDGE,  acknowledgeHandler);
                producer.addEventListener(MessageFaultEvent.FAULT, faultHandler);
            }
        ]]>
    </mx:Script>
</mx:Application>
```

Sending a message to a destination

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 component's `send()` method. You can create text messages and messages that contain objects.

The following code creates a message, populates the body of the message with text, and sends the message by calling a Producer component's `send()` method:

```
<?xml version="1.0"?>
<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 var producer:Producer;

            private function faultHandler(event:MessageFaultEvent):void{
            // Handle fault event.
            }
            private function logon():void {
                producer = new Producer();
                producer.destination = "ChatTopicJMS";
               producer.addEventListener(MessageAckEvent.ACKNOWLEDGE,  acknowledgeHandler);
               producer.addEventListener(MessageFaultEvent.FAULT, faultHandler);

            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>
```

The following code creates a message, populates the body of the message with an object, and sends the message by calling a Producer component's `send()` method.

```
<?xml version="1.0"?>
<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 var producer:Producer;

            private function faultHandler(event:MessageFaultEvent):void{
            // Handle fault event.
            }
            private function logon():void {
                producer = new Producer();
                producer.destination = "ChatTopicJMS";
               producer.addEventListener(MessageAckEvent.ACKNOWLEDGE,  acknowledgeHandler);
               producer.addEventListener(MessageFaultEvent.FAULT, faultHandler);

            private function sendMessage():void {
                var message:AsyncMessage = new AsyncMessage();
                message.body=new Object();
                message.body.prop1 = "abc";
                message.body.prop2 = "abc";
                message.body.theCollection=['b','a',3,new Date()];
                producer.send(message);
            }
        ]]>
    </mx:Script>
</mx:Application>
```

Adding extra information to a message

You can include extra information for a message in the form of message headers. You can send strings and numbers in message headers. The headers of a message are contained in an associative array where the key is the header name. The `headers` property of a message class lets you add headers for a specific message instance.

The following code adds a message header called prop1 and sets its value:

```
<?xml version="1.0"?>
<!-- fds\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 var producer:Producer;

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

You can use a Consumer component's `selector` property to filter the messages that the component receives, based on message header values. For more information, see "Filtering messages with a message selector" on page 102.

***Note:** Do not start message header names with the text JMS or DS. These prefixes are reserved.*

Resending messages and timing out requests

A Producer component sends a message once. If the delivery of a message that a Producer component sends is in doubt, the Producer component dispatches a fault event, which indicates that it never received an acknowledgment for the specified message. When the event is dispatched, the handler code can then make a decision to abort or attempt to resend the faulted message. Two events can trigger a fault that indicated delivery is in doubt. It can be triggered when the value of the `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 `faultCode` property of the associated ErrorMessage for the ErrorMessage.MESSAGE_DELIVERY_IN_DOUBT code.

Working with Consumer components

You can create Consumer components in MXML or ActionScript. To subscribe to a destination, you call a Consumer component's `subscribe()` method.

You can also specify message and fault event handlers for a Consumer component. A message event is broadcast when a message is received by the destination to which a Consumer component is subscribed. A fault event is broadcast when the channel to which the Consumer component is subscribed can't establish a connection to the destination, the subscription request is denied, or if a failure occurs when the Consumer component's `receive()` method is called. The `receive()` method retrieves any pending messages from the destination.

For reference information about the Consumer class, see the *Adobe Flex Language 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 should also specify 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"?>
<!-- fds\messaging\CreateConsumerMXML.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 messageHandler(event:MessageEvent):void {
                // Handle message event.
            }
            private function faultHandler(event:MessageFaultEvent):void {
            // Handle fault event.
            }
        ]]>
    </mx:Script>
    <mx:Consumer id="consumer"
        destination="ChatTopicJMS"
        message="messageHandler(event)"
        fault="faultHandler(event)"/>
</mx:Application>
```

Creating a Consumer component in ActionScript

You can create a Consumer component in an ActionScript method. The following code shows a Consumer component created in a method in an `<mx:Script>` tag. The import statements import the classes required to create a Consumer component, create AsyncMessage objects, add event listeners, and create message handlers.

```
<?xml version="1.0"?>
<!-- fds\messaging\CreateConsumerAS.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 var consumer:Consumer;
            private function acknowledgeHandler(event:MessageAckEvent):void{
                // Handle message event.
            }
            private function faultHandler(event:MessageFaultEvent):void{
            // Handle fault event.
            }
            private function logon():void {
                consumer = new Consumer();
                consumer.destination = "ChatTopicJMS";
                consumer.addEventListener
                    (MessageAckEvent.ACKNOWLEDGE,  acknowledgeHandler);
                consumer.addEventListener
                    (MessageFaultEvent.FAULT, faultHandler);
            }
        ]]>
    </mx:Script>
</mx:Application>
```

Subscribing to a destination

Whether you create a Consumer component in MXML or ActionScript, you still must call the component's `subscribe()` method to subscribe to a destination and receive messages from that destination.

A Consumer component can subscribe to destinations that use the BlazeDS Message Service with no message adapters or to destinations that use message adapters. For example, the Java Message Service (JMS) adapter enables Consumer components to subscribe to JMS topics. On the client side, the API for non-adapter and adapter-enabled destinations is identical. The definition for a particular destination in the services configuration file determines what happens when a Consumer component subscribes to the destination or a Producer component sends messages to the destination. For information about configuring destinations, see "Configuring Messaging on the Server" on page 108.

The following code shows a call to a Consumer component's `subscribe()` method:

```
<?xml version="1.0"?>
<!-- fds\messaging\Subscribe.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 var consumer:Consumer;

            private function logon():void {
                consumer = new Consumer();
```

```
                consumer.destination = "ChatTopicJMS";
                consumer.addEventListener
                    (MessageAckEvent.ACKNOWLEDGE,  acknowledgeHandler);
                consumer.addEventListener
                    (MessageFaultEvent.FAULT, faultHandler);
                consumer.subscribe();
            }
        ]]>
    </mx:Script>
</mx:Application>
```

You can unsubscribe a Consumer component from a destination by calling the component's `unsubscribe()` method.

Filtering messages with a message selector

You can use a Consumer component's `selector` property to filter the messages that the component should receive. 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. For information about creating message headers, see “Adding extra information to a message” on page 99.

The following code sets a Consumer component's `selector` property in an `<mx:Consumer>` tag:

```
<?xml version="1.0"?>
<!-- fds\messaging\CreateConsumerMXML.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 messageHandler(event:MessageEvent):void {
                // Handle message event.
            }
            private function faultHandler(event:MessageFaultEvent):void {
            // Handle fault event.
            }
        ]]>
    </mx:Script>
    <mx:Consumer id="consumer"
        destination="ChatTopicJMS"
        selector="prop1 > 5"
        message="messageHandler(event)"
        fault="faultHandler(event)"/>
</mx:Application>
```

The following code sets a Consumer component's `selector` property in ActionScript:

```
<?xml version="1.0"?>
<!-- fds\messaging\SelectorAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.events.*;
            private var consumer:Consumer;

            private function logon():void {
                consumer = new Consumer();
                consumer.destination = "ChatTopic";
                consumer.selector = "prop1 > 5";
                consumer.subscribe();
            }
        ]]>
    </mx:Script>
</mx:Application>
```

Note: *For advanced messaging scenarios, you can use the mx.messaging.MultiTopicConsumer and mx.messaging.MultiTopicProducer classes.*

Using subtopics

The subtopic capability lets you divide the messages that a Producer component sends to a destination into specific categories at 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 send or receive messages from more than one subtopic.

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 "Filtering messages with a message selector" on page 102.

In the `subtopic` property of a Producer component, you specify the subtopics that the component sends messages to. In the `subtopic` property of a Consumer component, you specify the subtopic(s) to which the Consumer is subscribed.

To send a message from a Producer component to a destination and a subtopic, you set the `destination` and `subtopic` properties, and then call the `send()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- fds\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 acknowledgeHandler(event:MessageAckEvent):void {
                // Handle message acknowledgement event.
            }
            private function faultHandler(event:MessageFaultEvent):void {
            // Handle fault event.
            }
            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="ChatTopicJMS"
        acknowledge="acknowledgeHandler(event)"
        fault="faultHandler(event)"/>
</mx:Application>
```

To subscribe to a destination and a subtopic with a Consumer component, you 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"?>
<!-- fds\messaging\Subtopic2.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 messageHandler(event:MessageEvent):void {
                // Handle message event.
            }
            private function faultHandler(event:MessageFaultEvent):void {
            // Handle fault event.
            }
            private function useSubtopic():void {
                consumer.destination = "ChatTopic";
                consumer.subtopic = "chat.fds.*";
                consumer.subscribe();
            }
        ]]>
    </mx:Script>
    <mx:Consumer id="consumer" destination="ChatTopicJMS"
         message="messageHandler(event)"
         fault="faultHandler(event)"/>
</mx:Application>
```

To allow subtopics for a destination, you must set the `allow-subtopics` element to `true` in the destination definition in the services-config.xml file or a file that it includes by reference, such as the messaging-config.xml file. The `subtopic-separator` element is optional; the default value is `"."` (period).

```
<destination id="ChatTopic">
    <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>false</durable>
            <allow-subtopics>true</allow-subtopics>
            <subtopic-separator>.</subtopic-separator>
        </server>
    </properties>
    <channels>
        <channel ref="my-rtmp"/>
    </channels>
</destination>
```

The following example is a runnable application that uses subtopics. In this case, the Producer and Consumer components are declared in MXML tags. The Consumer component uses a wildcard character (*) to receive all messages under the chat.fds subtopic.

```
<?xml version="1.0"?>
<!-- fds\messaging\Subtopic3.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 {
                myCon.subscribe();
                var message:AsyncMessage = new AsyncMessage();
                message.body = userName.text;
                myPub.send(message);
            }
            private function messageHandler(event: MessageEvent):void {
                ta.text += event.message.body + "\n";
            }
        ]]>
    </mx:Script>

    <mx:Producer id="myPub" destination="mike" subtopic = "chat.fds.blue"/>
    <mx:Consumer id="myCon" destination="mike" subtopic="chat.fds.*"
        message="messageHandler(event);"/>


    <mx:TextInput id="userName" width="20%"/>
    <mx:TextInput id="msg" width="20%"/>
    <mx:Button label="Send" click="sendMessage();"/>
    <mx:TextArea id="ta"/>
</mx:Application>
```

***Note:** For advanced messaging scenarios, you can use the mx.messaging.MultiTopicConsumer and mx.messaging.MultiTopicProducer classes.*

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 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 excerpts from a simple chat application that contains a pair of Producer and Consumer components. The user types messages in a `msg` 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. The user can switch between non-JMS and JMS-enabled destinations by selecting items in the topics List control.

```
<?xml version="1.0"?>
<!-- fds\messaging\ProducerConsumer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[

            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            private function logon():void {
                producer.destination = topics.selectedItem.toString();
                consumer.destination = topics.selectedItem.toString();
                consumer.subscribe();
                topics.selectedIndex = 1;
            }

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

            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="ChatTopicJMS"/>
    <mx:Consumer id="consumer" destination="ChatTopicJMS"
        message="messageHandler(event)"/>

    <mx:List id="topics">
        <mx:dataProvider>
            <mx:ArrayCollection>
                <mx:String>chat-topic</mx:String>
                <mx:String>chat-topic-jms</mx:String>
            </mx:ArrayCollection>
        </mx:dataProvider>
    </mx:List>
```

```
    <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>
```

Chapter 11: Configuring Messaging on the Server

To enable messaging in an Adobe Flex client application, you connect to the Message Service by declaring a connection to a server-side destination in `<mx:Producer>` and `<mx:Consumer>` tags or the corresponding ActionScript API. A Message Service *destination* is the endpoint that you send messages to and receive messages from when performing publish-subscribe or point-to-point messaging. You configure destinations as part of the Message Service definition in the Flex services configuration file.

For information about using the client-side messaging API and connecting to a destination in MXML or ActionScript, see "Creating Messaging Clients" on page 96.

Topics

Understanding Message Service configuration . . . 108
Configuring Message Service destinations . . . 110
Creating a custom Message Service adapter . . . 116

Understanding Message Service configuration

The most common tasks that you perform when configuring the Message Service are defining message destinations, applying security to message destinations, and modifying logging settings. A *Message Service destination* is the server-side code that you connect to using Producer and Consumer components. You configure Message Service destinations in the Message Service section of the Flex services configuration file or a file that it includes by reference. By default, the Flex services configuration file is named services-config.xml and is located in the WEB_INF/flex directory of the web application that contains BlazeDS.

The following example shows a basic Message Service configuration in the services configuration file. It contains one destination that is configured for a chat application that uses the default message adapter, which in this example is the ActionScript 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">
        <properties>
            <network>
                <session-timeout>0</session-timeout>
                <throttle-inbound policy="ERROR" max-frequency="50"/>
                <throttle-outbound policy="REPLACE" max-frequency="500"/>
            </network>
            <server>
                <max-cache-size>1000</max-cache-size>
```

```
                <message-time-to-live>0</message-time-to-live>
                <durable>true</durable>
                <durable-store-manager>
                    flex.messaging.durability.FileStoreManager
                </durable-store-manager>
            </server>
        </properties>
        <channels>
            <channel ref="samples-rtmp"/>
            <channel ref="samples-amf-polling"/>
        </channels>
    </destination>
...
</service>
...
```

Message Service destinations

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. The ActionScript adapter lets you use messaging with Flex clients as the sole producers and consumers of the messages. The Java Message Service (JMS) message adapter lets you interact with a JMS implementation. For more information, see "Message Service adapters" on page 109.

You also set network- and server-related properties. You can also reference or define security constraints for a destination. For more information, see "Configuring Message Service destinations" on page 110.

Message channels

A destination references one or more message channels, which are defined in the services-config.xml configuration file. The two most common channels that you use for messaging are the Real-Time Messaging Protocol (RTMP) channel and the Action Message Format (AMF) channel with message polling enabled.

The RTMP channel maintains a connection between the client and the server, so the client does not need to poll the server. The AMF channel with polling enabled polls the server for new messages.

For more information about securing messaging channels, see "Securing destinations" on page 153.

Message Service adapters

A *Message Service adapter* is the server-side code that facilitates messaging when your application uses ActionScript objects only or interacts with another system, such as an implementation of the Java Message Service (JMS). You reference adapters in a destination definition. You can also specify adapter-specific settings. For more information about message adapters, see "Configuring Message Service destinations" on page 110.

Security

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.

You can declare a security constraint inline in a destination definition, or you can declare it globally and reference it by its `id` in a destination definitions.

For more information about security, see "Securing destinations" on page 153.

Configuring Message Service destinations

Referencing message channels

Messages are transported to and from a destination over a message channel. A destination references one or more message channels that are also defined in the fds.config.xml configuration file. The destination can be contacted using only one of the listed channels.

Any attempt to contact the destination on an unlisted channel results in an error. Flex applications use this configuration information to determine which channels to use to contact the specified destination. The Flex application attempts to use the referenced channels in the order specified. You can also configure default message channels, in which case you do not have to explicitly reference any channels in the destination.

The following example shows a destination's channel references. Because the `samples-rtmp` channel is listed first, the destination attempts to use it first.

```
...
    <destination id="chat-topic">
...
        <channels>
            <channel ref="samples-rtmp"/>
            <channel ref="samples-amf-polling"/>
        </channels>
...
    </destination>
...
```

For more information about message channels, see “Securing destinations” on page 153.

Setting network properties

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>
                <session-timeout>0</session-timeout>
                <throttle-inbound policy="ERROR" max-frequency="50"/>
                <throttle-outbound policy="REPLACE" max-frequency="500"/>
            </network>
    ...
        </properties>
    </destination>
...
```

Message Service destinations use the following network-related properties:

Property	Description
`session-timeout`	Idle time in minutes before a subscriber is unsubscribed. When the value is set to 0 (zero), subscribers are not forced to unsubscribe automatically.
`throttle-inbound`	The `max-frequency` attribute controls how many messages per second the server can receive. The `policy` attribute indicates what to do when the message limit is reached. A `policy` value of `ERROR` indicates that an error should be returned if the limit is reached. A `policy` value of `IGNORE` indicates that no error should be returned if the limit is reached.
`throttle-outbound`	The `max-frequency` attribute controls how many messages per second the server can send. The `policy` attribute indicates what to do when the message limit is reached. A `policy` value of `ERROR` indicates that an error should be returned if the limit is reached. A `policy` value of `IGNORE` indicates that no error should be returned if the limit is reached. A `policy` value of `REPLACE` indicates that the previous message should be replaced when the limit is reached.

Setting server properties

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>
                <max-cache-size>1000</max-cache-size>
                <message-time-to-live>0</message-time-to-live>
                <durable>true</durable>
                <durable-store-manager>
                    flex.messaging.durability.FileStoreManager
                </durable-store-manager>
            </server>
        </properties>
    </destination>
...
```

Message Service destinations use the following server-related properties:

Property	Description
`max-cache-size`	Maximum number of messages to maintain in memory cache.
`message-time-to-live`	How many milliseconds that a message is kept on the server. A value of 0 means the message lasts forever.
`durable`	Boolean value that indicates whether messages should be saved in a durable message store to ensure that they survive connection outages and reach destination subscribers. When the JMS adapter is used, it inherits the `durable` value.
`durable-store-manager`	Durable store manager class to use when not using the JMS adapter. By default, the flex.messaging.durability.FileStoreManager, included with BlazeDS, stores messages in files in the /WEB-INF/flex/message_store/*DESTINATION_NAME* of your Flex web application. You can change this location by setting the `file-store-root` property. **Note:** Using the BlazeDS FileStoreManager for durable messaging is not cluster-safe.
`batch-write-size`	(Optional) Number of durable message files to write in each batch.

Property	Description
file-store-root	(Optional) Location for storing durable message files.
max-file-size	(Optional) Maximum file size in kilobytes for durable message files.
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 where there are active subscriptions, but subscribe and unsubscribe messages are broadcast in the cluster. For more information, see "Using software clustering" on page 158.

Referencing Message Service adapters

To use a Message Service adapter, such as the ActionScript, JMS, or ColdFusion Event Gateway Adapter included with BlazeDS, you reference the adapter in a destination definition or use an adapter configured as the default adapter. In addition to referencing an adapter, you also set its properties in a destination definition; for information about the JMS adapter, see "Configuring the JMS adapter" on page 112.

The following example shows the definition for the JMS adapter and a reference to it in a destination:

```
...
<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-jms">
...
        <adapter ref="jms"/>
...
    </destination>
...
</service>
```

Configuring the JMS adapter

You use the JMS adapter to subscribe to JMS topics or queues configured on an implementation of the Java Message Service. This adapter lets Java JMS publishers and subscribers share the same destination as a Flex client application. Java code can publish messages to the Flex application, and Java code can respond to messages that the Flex application publishes.

You configure the JMS adapter individually for the destinations that use it. You must configure the adapter with the proper Java Naming and Directory Interface (JNDI) information and JMS ConnectionFactory information to look up the connection factory in JNDI, create a connection from it, create a session on behalf of the Flex client, and create a publisher and subscribers on behalf of the Flex client.

If two Flex clients listen to the same JMS queue using the JMS adapter and the JMS server sends a message to the queue, only one of the clients receives the message at a given time. This is the expected behavior because JMS queues are meant to be consumed by one consumer.

The JMS adapter 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	JMS message acknowledgment mode. The valid values are AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE, and CLIENT_ACKNOWLEDGE.
connection-credentials	Optional. The username and password used while creating the JMS connection. Use only if JMS connection level authentication is being used.
connection-factory	Name of the JMS connection factory in JNDI.
delivery-settings/mode	(Optional) Default value is sync. This message delivery mode used in delivery messages from JMS server. If async mode is specified but the app server cannot listen for messages asynchronously (i.e. javax.jms.MessageConsumer.setMessageListener is restricted), or the app server cannot listen for connection problems asynchronously (for example, javax.jms.Connection.setExceptionListener is restricted), there will be a configuration error asking the user to switch to sync mode.
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 may cause things to back up if many proxied consumers are not receiving a steady flow of messages. Only available when the mode value is sync.
delivery-mode	JMS DeliveryMode for producers. The valid values are PERSISTENT and NON_PERSISTENT. The PERSISTENT mode causes all sent messages to be stored by the JMS server and then forwarded to consumers. This 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 may be lost if the JMS server fails while processing the message; this setting is suitable for notification messages that do not require guaranteed delivery.
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.
message-type	Type of the 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, you must set the message-type to javax.jms.ObjectMessage.
message-type	The javax.jms.Message type which the adapter should use for this destination. Supported types are javax.jms.TextMessage and javax.jms.ObjectMessage.
message-priority	JMS priority for messages that producers send. The valid values are DEFAULT_PRIORITY or an integer value indicating what the priority should be. 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.

Property	Description
preserve-jms-headers	(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. These headers are set by the JMS server 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.
max-producers	The maximum number of producer proxies that a destination should use 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.
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 114.
destination-name	(Deprecated) Name of the destination in JMS.
transacted-sessions	(Deprecated) JMS session transaction mode.

The following example shows a destination that uses the JMS adapter:

```
...
<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>
...
```

Using a remote JMS provider

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 JMS adapter. The initial-context-environment element takes property subelements, which in turn take name and value subelements. In the name and value elements, you can specify javax.naming.Context constant names and corresponding values, or you can specify string literal names and corresponding values to establish the desired JNDI environment.

The boldfaced 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 the 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, and you can specify these by using the string literal name and corresponding value that the provider requires. For example, the FioranoMQ JMS provider configures failover to backup 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.

Depending on the JNDI environment that you are using, the connection-factory and destination-jndi-name configuration elements must correctly reference the target named instances in the directory; naming conventions across JNDI providers for topic connection factories and destinations may vary. You must also include your JMS provider's client library JAR files in the WEB-INF/lib directory of your Flex web application or in another location from which the class loader will load 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 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 JMS adapter and the implications of the restrictions.

API	Implication of restriction
javax.jms.MessageConsumer.get/setMessageListener	No asynchronous message delivery
javax.jms.Connection.setExceptionListener	No notification of connection problems
javax.jms.Connection.setClientID	No durable subscribers
javax.jms.Connection.stop	Not an important implication

The JMS Adapter 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 to ask the user to switch to synchronous message delivery.
- When asynchronous message delivery is specified and setExceptionListener is restricted, a clear error message 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`.

Using a JMSConsumer object instead of the JMS adapter

You use a JMSConsumer to listen for JMS messages without relying on the JMS adapter. Customer code is responsible from receiving JMS messages, converting them to Flex messages and using BlazeDS APIs to pass the message to Flex clients.

In the following code example, a bootstrap service is used to create a JMSConsumer object when the server starts. When a JMS message is received, it is printed (rather than converting it to a Flex message and handing it to BlazeDS). For different workflows, a RemoteObject could be used to create JMSConsumer objects.

Creating a custom Message Service adapter

You can create a custom Message Service adapter for situations where you need functionality that is not provided by one of the standard adapters. A Message Service adapter class must extend the flex.messaging.services.ServiceAdapter 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. Within 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 from a client.

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.ServiceAdapter;
import flex.messaging.services.MessageService;
import flex.messaging.messages.Message;
import flex.messaging.Destination;

public class SimpleCustomAdapter extends ServiceAdapter {

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. To do this, you override 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 class's `getAdapterState()` and `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 there is another node running.

To use an adapter class, you must specify it in an `adapter-definition` element in the Message Services configuration, 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 lets you expose properties for getting and setting in the administration console. For more information, see “Monitoring and managing services” on page 161.

Chapter 12: Controlling Message Delivery with Adaptive Polling

The adaptive polling capability lets you write custom logic to control how messages are queued for delivery to an Adobe® Flex™ client applications on a per-client basis.

Topics

Adaptive polling 118
Using a custom queue processor. 119

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 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-samples/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 13: 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 124
Using the Ajax client library 125
Ajax client library API reference. 128

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 message 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 SWF file:

- FABridge.as
- FDMSBase.as

You use the following command line (on a single line) to compile the SWF file:

```
mxmlc.exe --stacktrace-verbose -services=<path_to_services-
config.xml_for_your_app>\services-config.xml <path_to_FDMSBridge.as>\FDMSBridge.as -o
<path_to_store_compiled_swf>\FDMSBridge.swf
```

This command line assumes that you added mxmlc to the system path, or you run the command from the sdk\bin directory.

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 (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.

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

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>

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 RTMPChannel("my-secure-amf",
```

```
            "https://servername:8100/app/messagebroker/amfsecure"));

        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 async calls required by the Messaging APIs.

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 14: 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 134
Measuring message processing performance 139

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 this 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 137.

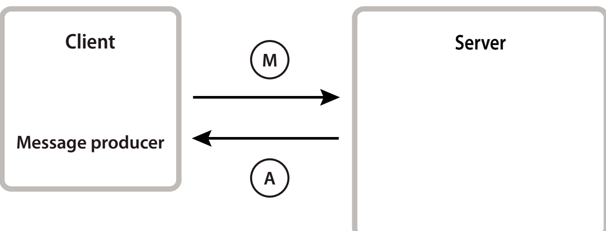
Measuring performance for different channel types

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:



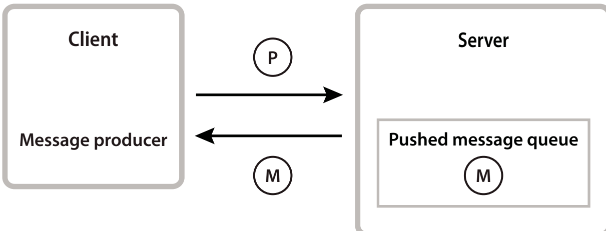
M. Message sent to server. A. Acknowledge message.

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:



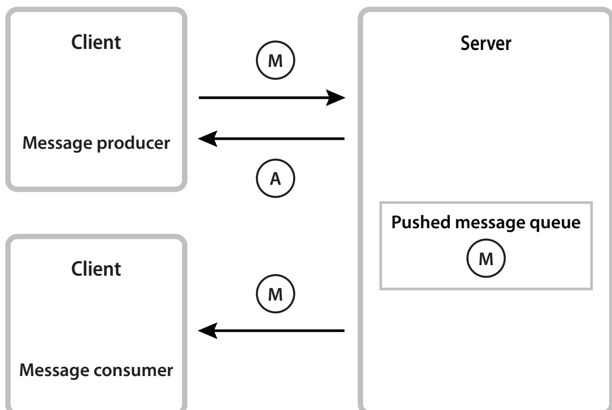
P. Polling message sent. M. Message pushed to client from server.

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 134.

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 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 corresponds to the time that the message was processed by Blaze DS.
`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 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 consumer's polling message and any message returned in response to the poll. For more information on these scenarios, see .
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, there are several considerations to take into account 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 a 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 that were deserialized by this endpoint during its lifetime.

- `getBytesSerialized()`

  Returns the total number of bytes that were 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
```

Part 4: Administering BlazeDS Applications

Configuring BlazeDS 147
Run-Time Configuration 167

Chapter 15: Configuring BlazeDS

The services in BlazeDS use a common set of configuration features in addition to features that are service-specific. The following features are common to all of the service types:

- Service configuration files
- Authentication and authorization
- Software clustering
- Service monitoring and management
- Class loading
- Server-side logging

Note: BlazeDS does not include any *Flex* compilers. You can use Flex Builder, the mxmlc command-line compiler, or the *Flex* web tier compiler module for J2EE to compile applications.

Topics

About service configuration files 147
Securing destinations 153
Using software clustering 158
Monitoring and managing services 161
About data services class loading 163
Server-side service logging 164

About service configuration files

You configure the Remoting Service, Proxy Service, and Message Service in the services section of the Data Services configuration file, services-config.xml. The default location of this file is the WEB-INF/flex directory of your BlazeDS web application. This location is set in the configuration for the MessageBrokerServlet in the WEB-INF/web.xml file.

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

Note: *If you use server tokens (for example, {server.name} and {server.port}) 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 Channel objects in ActionScript rather in a configuration file; for more information, see* *“Channel configuration” on page 10*.

The following table describes the typical set up of the configuration files. Commented versions of these files are available in the resources/config subdirectory of the BlazeDS installation directory; you can use those files as templates for your own configurations.

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 "Configuring RPC Services on the Server" on page 65.
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 "Configuring RPC Services on the Server" on page 65.
messaging-config.xml	The Message Service configuration file, which defines Message Service destinations for performing publish subscribe messaging. For information about configuring the Messaging Service, see "Configuring Messaging on the Server" on page 108.

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>
```

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. Adobe includes a separate configuration file for each type of standard service; these include the Proxy Service, Remoting Service, and Message 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: • "Configuring RPC Services on the Server" on page 65 • "Configuring Messaging on the Server" on page 108 • "Configuring Data Management on the Server" on page 148
`services/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.
`services/service`	Contains the definition for a data service. (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.
`services/service/properties`	Contains elements for service-level properties.
`services/service/adapters`	Contains definitions for service adapters that are referenced in a destination to provide specific types of functionality.
`services/service/adapters/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. **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.
`services/service/default-channels`	Contains references to a service's 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.
`services/service/default-channels/channel`	Contains references to the `id` of a channel definition. **Attributes:** • `ref` The `id` value of a channel definition.
`services/service/destination`	Contains a destination definition.
`services/service/destination/adapter`	Contains a reference to a service adapter. If this element is omitted, the destination uses the default adapter.

XML element	Description
services/service/destination/properties	Contains destination properties. The properties available depend on the type of service, which is determined by the service class specified.
services/service/destination/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.
services/service/destination/channels/channel	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.
services/service/destination/security	Contains references to security constraint definitions 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 "Securing destinations" on page 153.
services/service/destination/security/security-constraint	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.
services/service/destination/security/security-constraint/auth-method	Specifies the authentication method that the security constraint employs. A security constraint can use either basic or custom authentication. The valid values are basic and custom.
services/service/destination/security/security-constraint/roles	Specifies the names of roles that are defined in the application server's user store.
services/service/destination/security/login-command	Contains references to the id value of a login command definition that is used for performing custom authentication. **Attributes:** • ref The id value of a login command definition. For more information, see "Login adapters" on page 157
security	Contains security constraint definitions and login command definitions for authentication and authorization. For more information, see "Securing destinations" on page 153.
security/security-constraint	Defines a security constraint.
security/security-constraint/auth-method	Specifies the authentication method that the security constraint employs. A security constraint can use basic or custom authentication.
security/security-constraint/roles	Specifies the names of roles that are defined in the application server's user store.

XML element	Description
security/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 attribure is set to false by default. For more information, see "Login adapters" on page 157.
channels	Contains the definitions of message channels that are used to transport data between the server and clients. For more information, see "Securing destinations" on page 153.
channels/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.
channels/channel-definition/endpoint	Specifies the endpoint URI and the endpoint class of the channel definition. **Attributes:** • uri Endpoint URI. • class Fully qualified name of the endpoint class.
channels/channel-definition/properties	Contains the properties of a channel definition. The properties available depend on the type of channel specified. For more information, see "Securing destinations" on page 153.
clusters	Contains cluster definitions, which configure software clustering across multiple hosts. For more information, see "Using software clustering" on page 158.

XML element	Description
`clusters/cluster`	Defines a cluster that is enabled by default and disables the url-load-balancing. By default when you use a cluster, it gathers the endpoint URLs from all servers and sends them to the clients so the clients can implement failover between servers with different domain names. **Attributes:** • `id` Identifier of the cluster definition. • `default` A value of `true` sets a cluster definition as the default cluster that is used when destinations do not specify a `<cluster ref="..">` tag to enable clustering. • `properties` The name of a JGroups properties file. • `url-load-balancing` 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 your URLs. Instead, you must specify the unique server name and port clients will use to reach each server in that server's configuration.
`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. If `timeout-minutes` 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, so 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 "Server-side service logging" on page 372.
`logging/target`	Specifies the logging target class and the logging level. **Attributes:** • `class` Fully qualified logging target class name. • `level` The logging level; see "Server-side service logging" on page 372.
`logging/target/properties`	Contains the logging properties listed in the following rows. All of these properties are optional.
`logging/target/properties/prefix`	Specifies a prefix to include on log messages.
`logging/target/properties/includeDate`	Boolean value that indicates whether to include the date in log messages.
`logging/target/properties/includeTime`	Boolean value that indicates whether to include the time in log messages.
`logging/target/properties/includeLevel`	Boolean value that indicates whether to include the log level in log messages.

XML element	Description
`logging/target/properties/includeCategory`	Boolean value that indicates whether to include the log category in log messages. The log categories correspond to the values of the `filters/pattern` elements described below. Notice that the server logs against specific logging categories, not wildcard patterns.
`logging/target/filters`	Contains patterns to match against when creating log messages. Only messages that match specified patterns are logged.
`logging/target/filters/pattern`	A pattern to match against when creating log messages. Only messages that match specified patterns are logged. The valid pattern values are listed in “Server-side service logging” on page 372.
`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.
`system/locale`	(Optional) Locale string; for example, `"en"`, `"de"`, `"fr"`, and `"es"` are valid locale strings.
`system/default-locale`	The default locale string. If no `default-locale` element is provided, a base set of English error messages is used.
`system/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.
`system/redeploy/enabled`	Boolean value that indicates whether redeployment is enabled.
`system/redeploy/watch-interval`	Number of seconds to wait before checking for changes to configuration files.
`system/redeploy/watch-file`	A Data Services configuration file to be 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`
`system/redeploy/touch-file`	The file that the application server uses to force redeployment of a web application. The `touch-file` value must start with `{context.root}` or be an absolute path. Check the application server documentation to determine what the `touch-file` value should be. For Tomcat, the `touch-file` value is the web.xml file, as the following example shows: `{context.root}/WEB-INF/web.xml`

Securing destinations

When a destination is not public, you can restrict access to a privileged group of users by applying a security constraint in a destination definition in the services configuration file. A *security constraint* ensures that a user is authenticated, by using custom or basic authentication, before accessing the destination. By default, BlazeDS security constraints use custom authentication. A security constraint can also require that a user is authorized against roles defined in a user store before accessing a destination to determine if the user is a member of a specific role. A *user store* is a repository that contains security attributes of users.

Authentication is the process by which a user proves his or her identity to a system. *Authorization* is the process of determining what types of activities a user is permitted to perform in a system. After users are authenticated, they can be authorized to access specific resources.

You can declare a security constraint for a destination inline in a destination definition when the security constraint is used with only one destination. The following example shows a security constraint that is declared in a destination definition:

```
<service>
...
    <destination id="roDest">
...
        <security>
            <security-constraint>
                <auth-method>Custom</auth-method>
                <roles>
                    <role>roDestUser</role>
                </roles>
            </security-constraint>
        </security>
    </destination>
...
</service>
```

You can also declare a security constraint globally. When several destinations use the same security settings, you should define one security constraint in the security section of the Flex services configuration file and refer to it in each destination. The following example shows a security constraint that is referenced in two destination definitions:

```
<service>
    <destination id="SecurePojo1">
...
        <security>
            <security-constraint ref="trusted"/>
        </security>
    </destination>
    <destination id="SecurePojo2">
...
        <security-constraint ref="trusted"/>
    </destination>
...
</service>
...
<security>
    <security-constraint id="trusted">
        <auth-method>Custom</auth-method>
        <roles>
            <role>trustedUsers</role>
        </roles>
    </security-constraint>
...
</security>
```

For Remoting Service destinations, you can declare destinations tthat 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. 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 constraint(s) is checked.

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">
    <security>
        <security-constraint ref="sample-users" />
    </security>
    <properties>
        <source>my.company.SampleService</source>
        <include-methods>
            <method name="fooMethod"/>
            <method name="barMethod" security-constraint="admin-users"/>
        </include-methods>
    </properties>
</destination>
```

You can also use an `exclude-methods` element, which is similar to the `include-methods` element, but operates in the reverse direction. All public methods on the source class are visible to clients with the exception of the methods in this list. Use this if only a few methods on the source class need to be hidden and no methods require a tighter security constraint than the destination.

Passing credentials from client-side components

To send user credentials to a destination that uses a security constraint, you specify user name and password values as the parameters of the `setCredentials()` method of the RemoteObject, HTTPService, WebService, Producer, or Consumer component that you are using in your application. You can remove credentials by calling the component's `logout()` method.

Although you set credentials on a service component, credentials are actually applied to the underlying ChannelSet and Channel(s). If you have two service components and you set different credentials on each of them but they use the same ChannelSet/Channel, the last credentials you set are used. Also, if you have multiple components that use the same authenticated ChannelSet/Channel and you invoke the `logout()` method on one of them, they are all logged out.

You can configure a destination to use either basic or custom authentication. The type of authentication you specify determines the type of response that the server sends back to the client. For basic authentication, an HTTP 401 error causes a browser challenge. For custom authentication, the server sends a fault to the client to indicate that authentication is required.

If you use basic authentication and do you pass no credentials or you pass invalid credentials in the `setCredentials()` method, the web browser displays an authentication challenge dialog box when an application attempts to access the destination. If you use custom authentication and you pass no credentials or pass invalid credentials in the `setCredentials()` method, a fault is sent back to the client; this gives you the option of responding to the fault with a custom MXML-based login form that matches the appearance of your application.

Note: *Calling the `setCredentials()` or `setRemoteCredentials()` method has no effect when a service component's `useProxy` property is set to `false`.*

The following example shows ActionScript code for sending user name and password values from an HTTPService component to a destination:

```
import mx.rpc.http.HTTPService;

var employeeHTTP:HTTPService = new HTTPService();
employeeHTTP.destination = "SecureDest";
employeeHTTP.setCredentials("myUserName", "myPassword");
empoyeeHTTP.send({param1: 'foo'});
```

Passing credentials to a remote service

You use the `setRemoteCredentials()` method to pass credentials to a remote HTTP service or web service that requires them. For example, when using an HTTPService component, you can pass credentials to a secured JSP page. Passing remote credentials is distinct from passing user name and password credentials to satisfy a security constraint that you defined in the BlazeDS configuration file. 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 that are 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>
...
```

Basic authentication

Basic authentication relies on standard J2EE basic authentication from the web application container. When you use basic authentication to secure access to destinations, you usually also 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 in order to be challenged for a user name (principal) and password (credentials). The web browser performs the challenge, which happens independently of Flex. The web application container authenticates the user's credentials.

The following example shows a configuration for a secured channel endpoint in a web.xml file:

```
...
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Protected Channel</web-resource-name>

        <url-pattern>/messagebroker/amf</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>

    <auth-constraint>
        <role-name>sampleusers</role-name>
    </auth-constraint>
```

```
</security-constraint>

<login-config>
    <auth-method>BASIC</auth-method>
</login-config>

<security-role>
    <role-name>sampleusers</role-name>
</security-role>
...
```

When users successfully log in, they remain logged in until the browser is closed. When applying a basic authentication security constraint to a destination, Flex checks that there is a currently authenticated principal before routing any messages to the destination. You enable authorization by specifying roles in the destination's security constraint definition. The roles referenced in the web.xml file and in the security constraints that are defined in BlazeDS configuration files are all defined in the application server's user store.

Note: *How you define users and roles is specific to your application server. For example, by default, you define Apache Tomcat users and roles in the tomcat/conf/tomcat-users.xml file.*

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>
```

Login adapters

For both custom and basic authentication, Flex uses a login adapter, known as a login command, to check a user's credentials and log a principal into the application server. A login command must implement the `flex.messaging.security.LoginCommand` API. The authentication method being used, basic or custom, determines the type of response sent back to the client when the server receives a message from a nonauthenticated client. For basic authentication, an HTTP 401 error causes a browser challenge. For custom authentication, the server sends a fault to the client to indicate that authentication is required.

BlazeDS includes login command implementations for Apache Tomcat, Oracle Application Server, BEA WebLogic, IBM WebSphere, and Adobe JRun. Use the TomcatLoginCommand class for Tomcat or JBoss. You should enable only one login command at a time; comment out all others.

The following example shows a login command configured for Tomcat:

```
<security>
...
    <login-command class="flex.messaging.security.TomcatLoginCommand"
        server="Tomcat"/>
<!--
    <login-command class="flex.messaging.security.OracleLoginCommand"
        server="Oracle"/>

    <login-command class="flex.messaging.security.WeblogicLoginCommand"
        server="Weblogic"/>
    <login-command class="flex.messaging.security.WebSphereLoginCommand"
        server="WebSphere"/>
    <login-command class="flex.messaging.security.JRunLoginCommand"
        server="JRun"/>

-->
```

```
...
</security>
```

You can use a login command without roles for custom authentication only. If you also want to use authorization, you must link the specified role references to roles that are defined in your application server's user store.

The `login-command` element takes an optional `per-client-authentication` attribute, which is set to `false` by default. You can only set this attribute to `true` for a custom login command and not an application-server-based login command. Setting this attribute to `true` allows multiple clients that share the same session to have distinct authentication states. For example, two tabs of the same web browser could authenticate users independently. With shared authentication state, you cannot have different users logged into the separate instances of the application in the two tabs.

Another reason to use per-client authentication is to reset authentication state based on a browser refresh. Refreshing a browser does not get you a new server session, so 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.

Using software clustering

The BlazeDS software clustering capability handles failover for all channel types. It is most relevant to Messaging Service destinations. BlazeDS uses JGroups, an open source clustering platform. For information about JGroups, see http://www.jgroups.org/javagroupsnew/docs/index.html.

Another form of clustering uses a set of load balancers, usually in the form of hardware, which is positioned in front of HTTP servers. These load balancers direct requests to individual HTTP servers and pin client requests from the same client to that HTTP server. This form of clustering is possible with Flex without any feature implementation. It can also work in conjunction with the BlazeDS software clustering capability.

BlazeDSs 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 peers in the cluster. 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 there are three servers in the cluster, the elements in the first jgroups-tcp.xml config file would look like the following example:

```
<TCP bind_addr="server1.com" start_port="7800" loopback="true"/>
<TCPPING timeout="3000" initial_hosts="server2.com[7800],server3.com[7800]" port_range="1"
num_initial_members="2"/>
```

The `bind_addr` and `initial_hosts` attributes vary for each server in the cluster.

You should ensure that the server has a license key that supports clustering. Define a `<cluster>` in services-config.xml and add `<cluster ref="…"/>` to the destinations you have to cluster, where the `ref` attribute refers to the ID for the cluster that is defined.

A clustered destination exchanges the following information with corresponding destinations on the other server peers.

For channel endpoint URIs When a client connects to a clustered destination on a server, it receives the aggregated set of channel endpoint URIs for the destination across all servers in the cluster. This lets the client reconnect to the clustered destination on a different server or channel if its current connection fails.

For inbound and outbound message traffic A Consumer connected to one server can receive messages or changes that a different client sends or commits to another server. Messaging and data service destinations broadcast messages or changes to the corresponding destination on the other server peers. This lets all clients that interact with the clustered destination to stay synchronized

When a Flex client subscribes to a particular service on one host in a subnet, and the host fails, the client can direct further messages to another host in the same subnet. This feature is available for all service types defined in the Flex services configuration file. For the Message Service, both failover and replication of an application's 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. This 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.

Messaging components provide configurable connection and subscription retries when the connection to the server is lost. Producers provide `reconnectAttempts` and `reconnectInterval` properties to configure their reconnect behavior. Consumers provide `resubscribeAttempts` and `resubscribeInterval` properties that attempt to reconnect and resubscribe if a connection is lost. Messages sent by Producers or calls made by RPC services are not automatically queued or resent. These operations are not idempotent, and the application determines whether it is safe to retry a message send or RPC invocation.

To speed up the process when channels are failing over, but take too long to do so, you can set a `connectTimeout` property on your channels. You can do this directly in ActionScript or by defining a `<connect-timeout-seconds/>` in your channel config, for example:

```
<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>
```

The `requestTimeout` property that you can define on `RemoteObject` is used to force the client to give up on a request if it hasn't responded (either fault or result) from the remote endpoint within the interval. This is more useful with read-only calls over HTTP if the regular network request timeout is longer than desirable. Take care to use `requestTimeout` with calls that create/update/delete on the server side. If a request timeout comes back, query the server for its current state before you decide whether to retry the operation.

Processing messages

In addition to providing client failover, cluster nodes may have to process a message on all nodes. This is usually the process when no back-end resource is available to coordinate the common cluster state. When no shared back end is available, a cluster node directs the other node to reprocess and broadcast a message. When a shared back end is available, a node directs the other nodes to broadcast the message to connected clients, but it does not direct the other nodes to reprocess the message.

For publish-subscribe messaging, you have the option of using the server-to-server messaging feature to route messages set on any publish-subscribe messaging destination. You 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 where there are active subscriptions but subscribe and unsubscribe messages are broadcast in 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>
```

Defining and referencing a cluster

You define software clusters in the clusters section of the Flex services configuration file. You can define multiple clusters. Each `cluster` element must have an `id` attribute and a `properties` attribute that points to a JGroups properties file. You should use the same `id` attribute across all members of a cluster; you can use any value for the `id`.

The jgroups.jar file and the jgroups-*.xml properties files are located in the resources/clustering folder of the BlazeDS installation. You must copy the jgroups.jar file to the WEB-INF/lib and copy the jgroups-*.xml files to the WEB-INF/flex directory before you define clusters.

You reference a named cluster in the network section of the destinations that you include in the cluster. The channel that a clustered destination points to cannot contain any tokens in the URI specified as its endpoint. Tokens are not allowed unless `url-load-balancing` is disabled.

The following example shows the configuration 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>
```

In the services-config.xml file, you also must ensure that the definitions for the channels used in the cluster contain valid server names rather than `{server.name}` tokens. 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-samples/messagebroker/streamingamf"
        class="flex.messaging.endpoints.StreamingAMFEndpoint"/>
</channel-definition>
```

The cluster element can take the following attributes:

- `id` Identifier of the cluster definition.
- `default` A value of `true` sets a cluster definition as the default cluster.

- `properties` The name of a JGroups properties file.
- `url-load-balancing` 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 your URLs. Instead, you must specify the unique server name and port clients will use to reach each server in that server's configuration.

The following example shows a reference to a cluster in a service destination. The value of the `ref` attribute must match the value of the cluster's `id` attribute.

```
<destination id="MyDestination">
...
    <properties>
        <network>
            <cluster ref="default-cluster"/>
        </network>
    </properties>
...
</destination>
```

If you do not specify a `ref` value, the destination will use the default cluster definition when there is one.

Using of the `shared-backend="false"` configuration attribute does not guarantee a single consistent view of the data across the cluster. When two clients are connected to different servers in the cluster, if they perform conflicting updates to the same data value, each server applies these changes in a different order. The result is that clients connected to one server see one view of the data, while clients connected to other servers see a different view. The conflict detection mechanism does not detect these changes as conflicts. If you require a single consistent view of data where two clients may be updating the same data values at roughly the same time, you should use a database or other mechanism to ensure the proper locking is performed when updating data in your backend.

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 an example of 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 tab panes 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 pane provides a hierarchical tree of all the MBeans in the selected web application. Other views target specific types of information. For example, there are panes that provide server, channel endpoint, and destination information. These panes include dynamic data graphs for applicable properties.

There is a log manager pane that shows the log categories set in the services-config.xml file and lets you add or remove log categories at run time. You can change also change the log level (debug, info, warn, and so forth) for the log categories. For information about logging, see "Server-side service logging" on page 372.

***Note:** The run-time monitoring and management console exposes administrative functionality without authorization checks. You should deploy the flex-admin web application to the same application server that your Flex web application is deployed to, and you should lock down the flex-admin 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. This 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 `name` or `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

If you choose to write a custom MBean to expose metrics or management APIs for a custom ServiceAdapter, the ServiceAdapter class defines the following method to let you connect your corresponding 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. This lets your custom MBean be added 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. You can access this reference in your custom MBean by using the `protected ServiceAdapter serviceAdapter` instance variable. This lets your MBean query its corresponding service adapter for its state or invoke methods on it.

About data services class loading

For the most part, BlazeDS loads its classes in the same way as standard J2EE web applications by using the default class loader that the application server provides. As is typical with J2EE web applications, you might encounter issues when trying to use your own versions of JAR files that conflict with those that already exist in the application server source path. When troubleshooting any issues, always first ensure that your code runs in an application without BlazeDS.

Data services class loading

The MessageBrokerServlet creates the BlazeDS message broker. This servlet starts all of the services and endpoints. It is loaded by the standard class loader that is loaded by the application server to load classes for that web application. As with standard J2EE web applications, BlazeDS loads classes from the WEB-INF/lib directory using the web application class loader.

The following example shows the MessageBrokerServlet definition in the web.xml file:

```
<servlet>
    <servlet-name>MessageBrokerServlet</servlet-name>
    <display-name>MessageBrokerServlet</display-name>
    <servlet-class>flex.messaging.MessageBrokerServlet</servlet-class>
    <init-param>
        <param-name>services.configuration.file</param-name>
```

```
        <param-value>/WEB-INF/flex/services-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Servlet configuration

Each BlazeDS web application contains configuration files that you can edit to customize BlazeDS. You can use the *flex_app_root*/WEB-INF/flex/web.xml file to change settings such as the servlet mappings, paths, and other settings.

The following table describes the servlet defined in the web.xml file:

Servlet	Description
MessageBrokerServlet	Manages the Messaging Service. This servlet is opened for all requests matching the /messagebroker/* pattern. The definition of this servlet includes the location of the services-config.xml configuration file.

Server-side service logging

You can perform server-side logging for BlazeDS requests and responses. You configure server-side logging in the logging section of the Flex services configuration file. By default, output is sent to System.out. For information about client-side logging, see the Flex Help Resource Center.

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

- `all`
- `debug`
- `info`
- `warn`
- `error`
- `none`

In the filter pattern elements, you can specify the categories of information to log. In the `class` attribute of the `target` element, you can specify `flex.messaging.log.ConsoleTarget` to log messages to the standard output or the `flex.messaging.log.ServletLogTarget` to log messages to use your application server's default logging mechanism for servlets.

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>
```

In a production environment, you should set the logging level to `warn` so that both warnings and errors are displayed. If there are warnings that occur frequently that you prefer to ignore, change the level to `error` so that only errors are displayed.

The following table describes the logging levels:

Logging level	Description
`ALL`	Designates that all messages should be logged.
`DEBUG`	`DEBUG` messages indicate internal Flex activities. Select the `DEBUG` logging level to include `DEBUG`, `INFO`, `WARN`, and `ERROR` log messages in your log files.
`INFO`	`INFO` messages indicate general information to the developer or administrator. Select the `INFO` logging level to include `INFO`, and `ERROR` log messages in your log files.
`WARN`	`WARN` messages indicate that Flex encountered a problem with the application, but it does not stop running. Select the `WARN` logging level to include `WARN` and `ERROR` log messages in your log files.
`ERROR`	`ERROR` messages indicate when a critical service is not available or a situation has occurred that restricts use of the application. Select the `ERROR` logging level to include `ERROR` log messages in your log files.
`NONE`	Designates that no messages should be logged.

You can use the following matching patterns as the values of `pattern` elements:

- Client.*
- Client.FlexClient
- Client.MessageClient
- Configuration
- Endpoint.*
- Endpoint.General
- Endpoint.AMF
- Endpoint.FlexSession
- Endpoint.HTTP
- Endpoint.Type
- Message.*
- Message.General
- Message.Command.*
- Message.Command.*operation-name*

  where *operation-name* is one of the following: subscribe, unsubscribe, poll, poll_interval, client_sync, server_ping, client_ping, cluster_request, login, logout
- Message.coldfusion
- Message.Remoting
- Message.RPC
- Message.Selector
- Message.Timing
- Resource
- Protocol.*

- Service.*
- Service.Cluster
- Service.HTTP
- Service.Message
- Service.Message.*
- Service.Message.JMS (logs a warning if a durable JMS subscriber can't be unsubscribed successfully)
- Service.Remoting
- Security
- Startup.*
- Startup.MessageBroker
- Startup.Service
- Startup.Destination
- Timeout

Chapter 16: 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 167
Configuring components with a bootstrap service 168
Configuring components with a remote object 168
Accessing dynamic components with a Flex client application 170

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` param 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. Since the example for the first option creates a polling AMFChannel, it 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:8100/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 must define their ChannelSet, and you should 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 a 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.