Accessing Data with ADOBE® FLEX® 4.6

Legal notices

Contents

Chapter 1: Accessing data services overview

Data access in Flex compared to other technologies

The way that Flex works with data sources and data is different from applications that use HTML for their user interface.

Client-side processing and server-side processing

Unlike a set of HTML templates created using JSPs and servlets, ASP, PHP, or CFML, Flex separates client code from server code. The application user interface is compiled into a binary SWF file that is sent to the client.

When the application makes a request to a data service, the SWF file is not recompiled and no page refresh is required. The remote service returns only data. Flex binds the returned data to user interface components in the client application.

For example, in Flex, when a user clicks a Button control in an application, client-side code calls a web service. The result data from the web service is returned into the binary SWF file without a page refresh. Thus, the result data is available to use as dynamic content in the application.

```
<?xm l version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo" minWidth="1024" minHeight="768"
xmlns:employeesservice="services.employeesservice.*" xmlns:valueObjects="valueObjects.*">

    <fx:Declarations>
        <s:WebService
            id="RestaurantSvc"
        wsdl="http://examples.adobe.com/flex3app/restaurant_ws/RestaurantWS.xml?wsdl" />
        <s:CallResponder id="getRestaurantsResult"
            result="restaurants = getRestaurantsResult.lastResult as Restaurant"/>
    </fx:Declarations>

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

            protected function b1_clickHandler(event:MouseEvent):void {
                getRestaurantsResult.token = RestaurantWS.getRestaurantss();
            }
        ]]>
    </fx:Script>
. . .
    <s:Button id="b1" label="GetRestaurants" click="button_clickHandler(event)"/>
```

Compare this Flex example to the following example, which shows JSP code for calling a web service using a JSP custom tag. When a user requests the JSP, the web service request is made on the server instead of on the client. 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 data access component to connect to a remote service and to interact with the server-side data source.

The following example shows a ColdFusion page that accesses a data source directly:

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

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

Events, service calls, and data binding

Flex is an event driven technology. A user action or a program event can trigger access to a service. For example, a user clicking a button is a user action event that can be used to trigger a service call. An example of a program event is when the application completes the creation of a user interface component such as a DataGrid. The creationComplete event for the DataGrid can be used to call a remote service to populate the DataGrid.

Service calls in Flex are asynchronous. The client application does not have to wait for returned data. Asynchronous service calls are useful when retrieving or updating large sets of data. The client application is not blocked waiting for the data to be retrieved or updated.

Data returned from a service call is stored in a CallResponder that you associate with the service call. User interface components then use data binding to retrieve the returned data from the CallResponder.

Data binding in Flex allows you to dynamically update a user interface component with a data source. For example, a Flex component can associate its text attribute with the lastResult attribute of a CallResponder. When the data in the CallResponder changes, the Flex component automatically updates.

Flex also implements two-way data binding. With two-way data binding, when data changes in either the Flex component or the data source, the corresponding data source or Flex component automatically updates. Two-way data binding is useful when updating remote data from user inputs to a Form component or a Flex data component.

More Help topics

"Building data-centric applications with Flash Builder" on page 7

Using Flash Builder to access data services

In Flex Builder 3, you implement remote procedure calls to data services using Flex data access components. However, Flash Builder simplifies this process.

Flash Builder provides wizards and other tools that:

- Provide access to data services
- Configure data returned by the data service
- Assist in paging of data returned from the service
- Assist in data management functionality that synchronizes multiple updates to server data
- Generates client code for accessing data services
- Bind data returned from the service to user interface components

Flash Builder workflow for accessing services

Use the following workflow when using Flash Builder to create an application that accesses data services.

1 Depending on your circumstances, you start by connecting to a data service or by building the user interface.

Connect to remote service. If you start by connecting to the remote service, you then build the user interface.

Build user interface. If you start by building the user interface, you then connect to the remote service.

Note: *Where you start is a matter of personal preference. For example, if you already have a user interface design planned, you can build the user interface first. Conversely, you can connect to the data first and let Flash Builder assist you in generating application components.*

2 Bind data operations to application components.

3 (Optional) Manage the retrieval and update of data.

Flash Builder tools allow you to implement the paging of returned data and coordinate the update of sets of data.

When returning large amounts data records, you typically implement paging to retrieve a set of records on an "as needed" basis.

For applications that updates several records, you can implement data management features. Data Management features include:

- Commit functionality to update changed records simultaneously
- An undo mechanism to revert changes before they are written to the server

- Code generation that automatically updates user interface components as records are added, deleted, or changed

4 Run the application and monitor the data flow.

When the application is complete, run the application to view it in operation. Use the Flash Builder Network Monitor to view data passed between the application and the service. The Network Monitor is useful for diagnosing errors and analyzing performance.

Flash Builder also provides robust debugging and profiling environments. The Network Monitor and Flash Profiler are available with Flash Builder Premium.

More Help topics

"Building data-centric applications with Flash Builder" on page 7

Extending services supported by Flash Builder

Flash Builder wizards and tools support access to the following type of service implementations:

- PHP services
- ColdFusion services
- BlazeDS
- LiveCycle Data Services
- HTTP (REST-style) services
- Web services (SOAP)
- Static XML files

If you need tooling support for additional types of services, such as Ruby on Rails, you can extend the Flash Builder implementation. See Flash Builder Extensibility Reference.

Data access components

Data access components let a client application call operations and services across a network. Data access components use remote procedure calls to interact with server environments. The three data access components are the RemoteObject, HTTPService, and WebService components.

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

A data access component calls a remote service. It then stores response data from the service in an ActionScript object or any other format the service returns. Use data access components in the client application to work with three types of services:

- remote object services (RemoteObject)
- SOAP-based web services (WebService)
- HTTP services, including REST-based web services (HTTPService)

Adobe® Flash® Builder™ provides wizards and tools to wrap the implementation of a data access component into a service wrapper. The service wrapper encapsulates the functionality of the data access component, shielding you from much of the lower-level implementation. This allows you to concentrate on implementing services and building client applications to access the services. For more information on using Flash Builder to access data services, see "Building data-centric applications with Flash Builder" on page 7.

Providing access to services

By default, Adobe Flash Player blocks access to any host that is not exactly equal to the one used to load an application. If you do not use a server side application, such as LiveCycle Data Services or BlazeDS, to proxy requests, an HTTP service or web service must either be on the server hosting your application, or the remote server that hosts the HTTP or web service must define a `crossdomain.xml` file. A `crossdomain.xml` file 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 application is contacting.

HTTPService components

Use HTTPService components to send HTTP GET or POST requests and include the data from HTTP responses in a client application. If you are using Flex to build desktop applications (runs in Adobe AIR®), HTTP PUT and DELETE are supported.

If you use LiveCycle Data Services or BlazeDS, you can use an HTTPProxyService, which allows you to to use additional HTTP methods. With an HTTPProxyService, you can send GET, POST, HEAD, OPTIONS, PUT, TRACE, or DELETE requests.

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.

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.

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.

Flash Builder provides workflows that allow you to interactively connect to HTTP services. For more information, see "Accessing HTTP services" on page 11.

More Help topics

"Accessing HTTP services" on page 11

Dissertation: Representational State Transfer (REST) by Roy Thomas Fielding

WebService components

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

Client 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 supports 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 an application built with Flex, and a web service.

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

Flash Builder provides workflows that allow you to interactively connect to web services. For more information, see "Accessing web services" on page 15.

RemoteObject components

Remote object services let you access business logic directly in its native format rather than formatting it as XML, as you do with REST-style services or web services. This saves you the time required to expose existing logic as XML. Another benefit of remote object services is the speed of communication across the wire. Data exchanges still happen over HTTP or https, but the data itself is serialized into a binary representation. Using RemoteObject components results in less data going across the wire, reduced client-side memory usage, and reduced processing time.

ColdFusion, PHP, BlazeDS, and LiveCycle Data Services can use server-side typing when accessing data on the server. The client application accesses a Java object, ColdFusion component (which is a Java object internally), or PHP class directly by remote invocation of a method on a designated object. The object on the server uses its own native data types as arguments, queries a database with those arguments, and returns values in its native data types.

When server-side typing is not available, Flash Builder has tools to implement client-side typing. Use the Flash Builder to configure and define types for data returned from the service. Client-side typing allows the client application to query a database and retrieve properly typed data. Client-side typing is required for any service that does not define the type of data returned by the service.

Flash Builder provides workflows that allow you to interactively connect to remote object services. For more information, see "Connecting to data services" on page 8.

Chapter 2: Building data-centric applications with Flash Builder

Flash Builder tools can assist you in creating applications that access data services. You start by creating a Flex project for your applications. You can then connect to a data service, configure the access of data from the service, and build a user interface for an application. In some cases, you first create the user interface and then access the data service.

Creating a Flex project to access data services

Flex accesses data services as a remote object, an HTTP (REST-style) service, or a SOAP web service.

You use a remote object to access the following types of data services:

- ColdFusion services
- AMF-based PHP services
- BlazeDS
- LiveCycle Data Services

For information on using the LiveCycle Service Discovery wizard, see Using LiveCycle Discovery.

For any service accessed as a remote object, create a Flex project configured for the appropriate application server type. The New Flex Project wizard guides you through configuring a project for the application server types listed below:

Server Type	Remote Object Services Supported
PHP	• AMF-based PHP services
ColdFusion	• ColdFusion Flash Remoting • BlazeDS • LiveCycle Data Services
J2EE	• BlazeDS • LiveCycle Data Services

You can connect to HTTP (REST-style) services and SOAP web services from any Flex project configuration, including projects that do not specify a server technology.

A project configured to access a remote object can only access a remote object service for which it is configured. For example, you cannot access an AMF-based PHP service from a project configured for ColdFusion. However, you could connect to a PHP service from this project if you connect to the PHP service as a web service or HTTP service.

More Help topics

"Accessing data services overview" on page 1

Changing the server type of a project

Flash Builder notifies you if you attempt to access a service for which a Flex project is not configured. If the Flex project does not specify the correct server configuration, Flash Builder provides a link to the Project Properties dialog. In the Project Properties dialog, you can configure the project to access the data service. For example, Flash Builder warns you if you attempt to access an AMF-based PHP service from a project that does not specify a server configuration.

If the Flex project has previously been configured to access a different type of service, configure a new Flex project or change the configuration of the current project. If you change the server configuration of a project, then you can no longer access any services previously configured. For example, if you change a project configuration from ColdFusion to PHP, any ColdFusion services you access in the project are no longer available.

If you want to access different types of services from the same project, you can access the services as either HTTP services or web services.

Cross-domain policy file

A cross-domain policy file is necessary when accessing services that are on a different domain from the SWF file for the application. Typically, AMF-based services do not need a cross-domain policy file because these services are on the same domain as the application.

Connecting to data services

Use the Flash Builder Service wizard to connect to data services.

For remote object services, you typically create a Flex project with the corresponding application server type. Flash Builder introspects the service and can configure return types for data returned by the service.

Remote object services include data services implemented in ColdFusion, PHP, BlazeDS, and LiveCycle Data Services.

For information on using the LiveCycle Service Discovery wizard, see Using LiveCycle Discovery.

More Help topics

"Creating a Flex project to access data services" on page 7

Accessing ColdFusion services

Use the Flash Builder Service wizard to access a ColdFusion data service that has been implemented as a ColdFusion component (CFC). Flex accesses these services as remote objects.

Use a Flex project that specifies ColdFusion as the application server type. When creating the Flex project, specify Use Remote Object Access Service and use ColdFusion Flash Remoting.

Connecting to ColdFusion data services

This procedure assumes that you have implemented a ColdFusion service and have created a Flex project for accessing ColdFusion services.

1 From the Flash Builder Data menu, select Connect to ColdFusion to open the Service window.

2 In the Configure ColdFusion Service dialog, browse to the location of the CFC implementing the service.

Note: *If you have not implemented a ColdFusion service, Flash Builder can generate a sample service from a single database table. Use the generated sample as an example of how to access data services. See "Generating a sample ColdFusion service from a database table" on page 9.*

3 (Optional) Modify the service details.

Service Name	Specify a name for the service. Flash Builder generates a name for the service, based on the filename for the service. There are restrictions to names you can use for a service. See "Naming data services" on page 19.
Service Package	Specify a name for the package that contains generated ActionScript files that access the service. Flash Builder generates a package based on the service name and places it in a `services` package.
Data Types Package	Specify a name for the package that contains generated ActionScript class files that define data types retrieved from the service. By default, Flash Builder creates the `valueObjects` package.

4 (Optional) Click Next to view the service operations.

5 Click Finish to generate ActionScript files that access the service.

Note: *After connecting to a service, you can modify the service properties. Select the service in the Data/Services view. From the context menu, select Properties.*

Next step: "Configuring data types for data service operations" on page 25.

Generating a sample ColdFusion service from a database table

Flash Builder can generate a sample ColdFusion service that you can use as a prototype for your own services. The sample service accesses a single database table and has methods for create, read, update, and delete.

Flash Builder configures return data types for the generated services and enables data access functionality such as paging and data management.

Important: *Use the generated service only in a trusted development environment. The generated code allows anyone with network access to your server to be able to access, modify, or delete data in the database table. Before deploying this service, be sure to increase security and restrict access appropriately. For information on writing secure services, see Securing Data Services.*

The following procedure assumes that you have created a Flex project for accessing ColdFusion services and have ColdFusion data sources available.

1 From the Flash Builder Data menu, select Connect to ColdFusionto open the Service wizard.

2 In the Configure ColdFusion Service dialog, click the link to generate a sample service.

3 Select Generate from RDS Datasource and specify a ColdFusion Data Source and Table.

If the table does not define a primary key, select a Primary Key for the table.

Note: *If you do not have ColdFusion data source available, select Generate from Template. Flash Builder writes a sample ColdFusion component (CFC) with typical service operations. Uncomment specific functions in the CFC and modify the operations to create a sample service that you can use as a prototype.*

4 Use the default location or specify a new location. Click OK.

Flash Builder generates the sample service. Modify the Service Name and package locations to override the default values.

5 (Optional) Click Next to view operations in the service.

6 Click Finish.

Flash Builder generates ActionScript files that access the sample service. Flash Builder also opens the sample service in an editor on your system that is registered to edit ColdFusion CFC files.

Accessing PHP services

Use the Flash Builder Service wizard to connect to data services implemented in PHP. Flex uses Action Message Format (AMF) to serialize data between the client application and the data service. Flash Builder installs the Zend AMF framework to provide access to services implemented in PHP. See "Installing Zend Framework" on page 19.

Access PHP data services from a Flex project that specifies PHP as the application server type. The data service must be available under the web root you specified when configuring the project for PHP. Place the service in a services directory, as listed below:

```
<webroot>/MyServiceFolder/services
```

More Help topics

"Creating a Flex project to access data services" on page 7

Connecting to PHP data services

This procedure assumes that you have implemented a PHP service and have created a Flex project for accessing PHP services.

1 From the Flash Builder Data menu, select Connect to PHPto open the Service wizard.

2 In the Configure PHP Service dialog, browse to the PHP file implementing the service:

Note: *If you have not implemented a PHP service, Flash Builder can generate a sample service from a single database table. Use the generated sample as an example of how to access data services. See "Generating a sample PHP service from a database table" on page 11.*

3 (Optional) Modify the service details.

Service Name	Specify a name for the service. Flash Builder generates a name for the service, based on the filename for the service. There are restrictions to names you can use for a service. See "Naming data services" on page 19.
Service Package	Specify a name for the package that contains generated ActionScript files that access the service. Flash Builder generates a package based on the service name and places it in a `services` package.
Data Types Package	Specify a name for the package that contains generated ActionScript class files that define data types retrieved from the service. By default, Flash Builder creates the `valueObjects` package.

4 Click Next to view the service operations.

If you do not have the supported version of the Zend Framework for accessing PHP services, Flash Builder prompts you to install the minimal version of the Zend Framework. See "Installing Zend Framework" on page 19.

5 Click Finish.

Flash Builder generates ActionScript files that access the service.

Note: *After connecting to a service, you can modify the service properties. Select the service in the Data/Services view. From the context menu, select Properties.*

Next step: "Configuring data types for data service operations" on page 25.

Generating a sample PHP service from a database table

Flash Builder can generate a sample PHP service that you can use as a prototype for your own services. The sample service accesses a single MySQL database table and has methods for create, read, update, and delete.

Flash Builder configures return data types for the generated services and enables data access functionality such as paging and data management.

Important: *Use the generated service only in a trusted development environment. The generated code allows anyone with network access to your server to be able to access, modify, or delete data in the database table. Before deploying this service, be sure to increase security and restrict access appropriately. For information on writing secure services, see Securing Data Services.*

The following procedure assumes that you have created a Flex project for accessing PHP services and have a MySQL data source available.

1. From the Flash Builder Data menu, select Connect to PHP to open the Service wizard.
2. In the Configure PHP Service dialog, click the link to generate a sample service.
3. Select Generate from Database and specify the information to connect to a database. Click Connect to Database.

   Note: *If you do not have PHP data source available, select Generate from Template. Flash Builder writes a sample project with typical service operations. Uncomment specific areas of the project and modify the operations to create a sample service that you can use as a prototype.*
4. Select a table in the database and specify the primary key.
5. Use the default location or specify a new location. Click OK.

   If you do not have the supported version of the Zend Framework for accessing PHP services, Flash Builder prompts you to install the minimal version of the Zend Framework. See "Installing Zend Framework" on page 19.

   Flash Builder generates the sample service. Modify the Service Name and package locations to override the default values.
6. (Optional) Click Next to view operations in the service.
7. Click Finish.

   Flash Builder generates ActionScript files that access the sample service. Flash Builder also opens the sample service in an editor on your system that is registered to edit PHP files.

Accessing HTTP services

Use the Flash Builder Service wizard to connect to REST-based HTTP services. You can connect to HTTP services from any Flex project. You do not have to specify a server technology for the project.

A cross-domain policy file is necessary when accessing services that are on a different domain from the SWF file for the client application. See Using cross-domain policy files.

Configuring HTTP services

When accessing REST-based HTTP services, there are various ways to configure access to the service. The Configure HTTP Service wizard supports the following:

- Base URL as prefix

  Base URL as prefix is convenient if you are accessing multiple operations from a single service. If you specify a base URL to the service, then for each operation you specify only the relative path to the HTTP operations.

  If you want access to multiple services, then you cannot use Base URL.

- URLs with query parameters

  When specifying a URL to an operation, you can include the query parameters for the service operations. The Configure HTTP Service wizard fills the Parameter table with each parameter included in the operation URL.

- RESTful services with delimited parameters

  Flash Builder supports access to RESTful services that use delimited parameters instead of GET query parameter. For example, suppose you use the following URL to access a RESTful service:

  `http://restfulService/items/itemID`

  Use curly brackets ({}) to specify the parameters in the operation URL. For example:

  `http://restfulService/{items}/{itemID}`

  Then the Configure HTTP Service wizard populates the Parameter Table:

Name	Data Type	Parameter Type
items	String	URL
itemID	String	URL

  When specifying RESTful service parameters, the Data Type and Parameter Type are always configured as String and URL respectively.

  Note: *You can mix RESTful service parameters with query parameters when specifying the URL to an operation.*

- Path to a local file for an operation URL

  For an operation URL, you can specify a path to a local file that implements HTTP services. For example, specify the following for an operation URL:

  ```
  c:/MyHttpServices/MyHttpService.xml
  ```

- Adding GET and POST operations

  You can add additional operations when configuring an HTTP service. Click Add in the Operations table to add operations.

  Specify GET or POST for the operation method.

- Adding parameters to an operation

  You can add parameters to selected operations in the Operations table. Select the operation in Operations table and click Add in the Parameters table.

  Specify a name and Data Type for the added parameter. The Parameter Type, GET or POST, corresponds to the operation method.

- Content type for POST operations

For POST operations, you can specify the content type. The content type can be either `application/x-www-form-urlencoded` or `application/xml`.

If you choose `applicaton/xml` for the content type, Flash Builder generates a query parameter that cannot be edited. strXML is the default name. You specify the actual parameter at runtime.

Name	Data Type	Parameter Type
strXML	String	POST

You cannot add additional parameters for `application/xml` content type.

Connecting to HTTP services

1 From the Flash Builder Data menu, select Connect to HTTP to open the Service wizard.

2 (Optional) Specify a base URL to use as a prefix to all operations.

3 Under Operations, specify the following for each operation you want to access:

- Specify the operation method (GET or POST)
- URL to the service operation

  Include any parameters to the operation in the URL. Use curly brackets ({}) to specify REST-style service parameters.

  Flash Builder supports access to the following protocols:

  `http://`

  `https://`

  Standard absolute paths, such as `C:/` or `/Applications/`
- A name for the operation

4 For each operation parameter in a selected URL, specify the Name and Data Type of the parameter

5 (Optional) Click Add or Delete to add or remove parameters to the selected operation.

6 (Optional) Modify the service details.

Service Name	Specify a name for the service. Flash Builder generates a name for the service, based on the filename for the service. There are restrictions to names you can use for a service. See "Naming data services" on page 19
Service Package	Specify a name for the package that contains generated ActionScript files that access the service. Flash Builder generates a package based on the service name and places it in a `services` package.
Data Types Package	Specify a name for the package that contains generated ActionScript class files that define data types retrieved from the service. By default, Flash Builder creates the `valueObjects` package.

7 (Optional) Modify the generated package name for the service.

8 Click Finish.

Flash Builder generates ActionScript files that access the service.

After connecting to the HTTP service, configure the return types for service operations. When configuring the return type, you also configure the type for parameters to the operation. See "Configuring data types for data service operations" on page 25.

Note: *After connecting to a service, you can modify the service properties. Select the service in the Data/Services view. From the context menu, select Properties.*

Next step: "Configuring data types for data service operations" on page 25.

Accessing an XML file implementing HTTP services

You can access a static XML file that implements an HTTP service. The static XML file can be a local file or available as a URL.

The service uses a GET method that returns an XML response. This feature is useful for learning about HTTP services in Flex and for prototyping mock data in client applications.

When accessing the service, you specify the node returning the XML response. Flash Builder uses this node to automatically configure a return type for the data. After connecting to the service, you can bind operations to the service to user interface components.

Connecting to an XML service file

1. From the Flash Builder Data menu, select HTTP to open the Service wizard.
2. Specify Local File or URL and browse to the file.
3. Select a node in the file that contains the response you want.

   Indicate if the response is an Array.

   Flash Builder configures a return type for the selected node.
4. Modify the service details.

Service Name	Specify a name for the service. Flash Builder generates a name for the service, based on the filename for the service. There are restrictions to names you can use for a service. See "Naming data services" on page 19
Service Package	Specify a name for the package that contains generated ActionScript files that access the service. Flash Builder generates a package based on the service name and places it in a `services` package.
Data Types Package	Specify a name for the package that contains generated ActionScript class files that define data types retrieved from the service. By default, Flash Builder creates the `valueObjects` package.

5. (Optional) Modify the generated package name for the service.
6. Click Finish.

   Flash Builder generates ActionScript files that access the service.

Note: *After connecting to a service, you can modify the service properties. Select the service in the Data/Services view. From the context menu, select Properties.*

Accessing web services

Use the Flash Builder Service wizard to connect to web services (SOAP). You can connect to web services from any Flex project. You do not have to specify a server technology for the project.

A cross-domain policy file is necessary when accessing services that are on a different domain from the SWF file for the client application.

More Help topics

Using cross-domain policy files

Connecting to web services

1 From the Flash Builder Data menu, select Web Service to open the Service wizard.

2 (BlazeDS/Data Services) If you have installed LiveCycle Data Services or BlazeDS, you can access the web service through a proxy.

Select Through a BlazeDS/Data Services proxy destination.

Specify a destination. Click Next and proceed to step 5.

Note: *Accessing web services through a proxy is only enabled if your Flex project specifies J2EE as the application server type.*

3 Enter a URI to the SOAP service.

4 (Optional) Modify the service details.

Service Name	Specify a name for the service. Flash Builder generates a name for the service, based on the WSDL URI. There are restrictions to names you can use for a service. See "Naming data services" on page 19
Service Package	Specify a name for the package that contains generated ActionScript files that access the service. Flash Builder generates a package based on the service name and places it in a `services` package.
Data Types Package	Specify a name for the package that contains generated ActionScript class files that define data types retrieved from the service. By default, Flash Builder creates the `dataValues` package.

5 (Optional) Configure the code generation for the service:

Service **Port**	Select a service from the services available. Flash Builder generates a name for the service, based on the WSDL URI.
Operation List	Select the operations from the service that you want to access in your client application.

6 Click Finish.

Flash Builder generates ActionScript files that access the service.

Note: *After connecting to a service, you can modify the service properties. Select the service in the Data/Services view. From the context menu, select Properties.*

'After connecting to the web service, configure the return types for service operations. See "Configuring data types for data service operations" on page 25 for details.

Accessing BlazeDS

You can only access BlazeDS services if you have installed Adobe® BlazeDS and configured a Remote Development Services (RDS) server. See the LiveCycle Data Services ES documentation for information on installing and configuring BlazeDS.

You typically access BlazeDS data services from a Flex project configured with J2EE as the application server type.

More Help topics

"Creating a Flex project to access data services" on page 7

Connecting to BlazeDS services

This procedure assumes that you have installed BlazeDS, configured a Remote Development Server, and have created a Flex project for accessing BlazeDS services.

1 From the Flash Builder Data menu, select Connect to BlazeDS to open the Service wizard.

2 Select a destination to import.

3 (Optional) Modify the service details.

Service Name	Specify a name for the service. Flash Builder generates a name for the service, based on the destination. There are restrictions to names you can use for a service. See "Naming data services" on page 19
Service Package	Specify a name for the package that contains generated ActionScript files that access the service. Flash Builder generates a package based on the service name and places it in a `services` package.
Data Types Package	Specify a name for the package that contains generated ActionScript class files that define data types retrieved from the service. By default, Flash Builder creates the `valueObjects` package.

4 Click Finish.

Flash Builder generates ActionScript files that access the service.

Note: *After connecting to a service, you can modify the service properties. Select the service in the Data/Services view. From the context menu, select Properties.*

Accessing LiveCycle Data Services

You can only access services available from LiveCycle Data Services if you have installed LiveCycle Data Services and configured a Remote Development Services (RDS) server. See the LiveCycle Data Services documentation for information.

You can access LiveCycle Data Services from a Flex project configured with either J2EE or ColdFusion as the application server type.

Service types for LiveCycle Data Services

When connecting to LiveCycle Data Services, the following types of data services are available as destinations:

- Remoting service

  Remoting services are implemented using AMF typing. These services do not provide server-side data management. You can use Flash Builder tools to configure client side data management. See "Enabling data management" on page 31.

- Data service

  Data services are services that implement server-side data management. For more information, see your LiveCycle Data Services documentation.

- Web service

  Web services available through a LiveCycle Data Services proxy that is configured as an LiveCycle Data Services destination. Server-side typing is not typically provided when connecting to a web service.

Data type configuration and data management

Flash Builder provides tools for client side data configuration and client side data management. The Flash Builder tools that are available depends on the type of the LiveCycle Data Services destination:

- Remoting service

  Remoting services implement AMF typing on the service. You do not configure return data types for remoting service destinations.

  However, you can use Flash Builder to generate code for client side data management. See "Enabling data management" on page 31.

- Data service

  Data services implement server-side data types. You do not configure return data type for data service destinations.

  Data service destinations also provide server-side data management. You do not use client side data management with data service destinations.

- Web service

  Web service destinations available through a LiveCycle Data Service proxy typically do not implement server side typing. You can use Flash Builder tools to configure return types for web service operations. See "Configuring data types for data service operations" on page 25.

  You can use Flash Builder to generate code for client side data management. See "Enabling data management" on page 31.

Connecting to LiveCycle Data Service destinations (Data service and remoting service destinations)

This procedure assumes that you have installed LiveCycle Data Services, configured a Remote Development Server, and have created a Flex project for accessing LCDS services.

1. From the Flash Builder Data menu, select Connect to Data/Service to open the Service wizard.
2. In the Select Service Type dialog, select LCDS. Click Next.
3. Provide login credentials, if needed.
4. (Optional) Modify the service details.

Service Name	You do not provide a service name. Flesh Builder generates a service name. Flash Builder generates a name for the service, based on the destination.
Service Package	Specify a name for the package that contains generated ActionScript files that access the service. Flash Builder generates a package based on the service name and places it in a `services` package.
Destinations	Specify one or more destinations available from the LiveCycle Data Services server.
Data Types Package	Specify a name for the data type package. this package contains generated ActionScript class files that define data types retrieved from the service. By default, Flash Builder creates the `valueObjects` package.

5 Click Finish.

Flash Builder generates ActionScript files that access the service.

Note: *After connecting to a service, you can modify the service properties. Select the service in the Data/Services view. From the context menu, select Properties.*

Connecting to LiveCycle Data Service destinations (Web service destinations)

This procedure assumes that you have installed LiveCycle Data Services, configured a Remote Development Server, and have created a Flex project for accessing DS services.

1 From the Flash Builder Data menu, select Connect to Data/Service to open the Service wizard.

2 In the Select Service Type dialog, select Web Service. Click Next.

3 Select through a LCDS/BlazeDS proxy destination.

4 Provide login credentials, if needed.

5 Select the destination.

6 (Optional) Modify the service details. Click Next.

Service Name	Specify a name for the service. Flash Builder generates a name for the service, based on the destination name. There are restrictions to names you can use for a service. See "Naming data services" on page 19
Service Package	Specify a name for the package that contains generated ActionScript files that access the service. Flash Builder generates a package based on the service name and places it in a `services` package.
Data Types Package	Specify a name for the package that contains generated ActionScript class files that define data types retrieved from the service. By default, Flash Builder creates the `dataValues` package.

7 (Optional) Configure the code generation for the service:

Service **Port**	Select a service and Port from the services and ports available.
Operation List	Select the operations from the service that you want to access in your client application.

8 Click Finish.

Flash Builder generates ActionScript files that access the service.

Note: *After connecting to a service, you can modify the service properties. Select the service in the Data/Services view. From the context menu, select Properties.*

More Help topics

"Creating a Flex project to access data services" on page 7

Naming data services

Data services that are accessed from Flash Builder have restrictions on the names allowed for the service. Some of these restrictions are not apparent until you compile your application.

The naming guidelines for services are:

- The initial letter of the service cannot be a number.
- Service names cannot be ActionScript keywords.
- Do not use any ActionScript class name, including custom classes, as a service name.
- (PHP only) If a service name contains underscores, Flash Builder cannot import the service.

Note: *It is good practice to use service names that are different from the names of your MXML files.*

Installing Zend Framework

When initially accessing PHP services, Flash Builder determines if the supported version of the Zend Framework is installed. If the supported version of the Zend Framework is not found, Flash Builder prompts you to confirm installation of the Zend Framework. If you accept, then Flash Builder installs a minimal version of the Zend Framework. If you decline, then manually install the Zend Framework if you are going to access PHP services.

Default Flash Builder installation

Flash Builder installs the Zend Framework into a `ZendFramework` folder in the root directory of your web server.

```
<web root>/ZendFramework/
```

For Flex projects that access PHP services, Flash Builder creates the following configuration files in the project output folder:

- `amf_config.ini`
- `gateway.php`

Production servers

For productions servers, Adobe recommends that you move the `ZendFramework` folder outside the web root. Update the `zend_path` variable defined in `amf_config.ini`.

If the `zend_path` variable is commented out, uncomment the `zend_path` variable. Specify the location of your Zend Framework installation.

Manual installation of Zend Framework

You can choose to manually install the Zend Framework.

1 Download the latest release of the Zend Framework.

You can install either the minimal or full package. Flash Builder installs the minimal package.

2 Extract the downloaded version to a location on your system.

3 In the Flex project folder for accessing PHP services, update the `zend_path` variable defined in `amf_config.ini`.

If the `zend_path` variable is commented out, uncomment the `zend_path` variable. Specify the absolute path to the location of your Zend Framework installation.

Troubleshooting a Zend Framework installation

If you get an error connecting to the Zend Framework, here are some tips in resolving the error.

Manual installation of the Zend Framework

If you manually installed the Zend Framework, examine the zend_path variable in `amf_config.ini`.

`amf_config.ini` is in the project output folder.

Check the following:

- Make sure `zend_path` is uncommented.
- The specified path to your Zend Framework installation is correct:
  - The path is an absolute path to a destination on the local files system. You cannot specify a path to a mapped network resource.
  - The path is to the library folder of your Zend Framework installation. Typically, the library folder is in the following locations:

    (Windows) `C:\apache\PHPFrameworks/ZendFramework/library`

    (Mac OS) `/user/apache/PHP/frameworks/ZendFramework/library`

Flash Builder installation of the Zend Framework

If Flash Builder installed the Zend Framework, check the following:

- The location of the web root folder

  Flash Builder installs the Zend Framework in the project's web root folder. Check the location of the web root folder. Select Project > Properties> Flex Server.

- Ensure that the web server is configured to use PHP.
- Examine the zend_path variable in `amf_config.ini`.

  `amf_config.ini` is in the project output folder.

  Check the following:

  - `zend_path` is uncommented.
  - The specified path points to the Zend Framework installation at the project's web root
  - The path is an absolute path to a destination on the local files system. You cannot specify a path to a mapped network resource.

Using a single server instance

After you connect to a data service, each application in a project can access the service. By default, each application creates its own service instance when accessing the server.

You can modify this behavior so that there is only a single service instance in a project. Each application in the project accesses the same service instance. Typically, you create a single server instance when you want to coordinate the access of data from multiple applications.

You can specify to access a single service instance on a project basis or as a preference for all projects.

Access a single server instance for a project

1 Select Project > Properties > Data/Services

2 Enable the check box for using a single server instance. Click OK.

Specify single server instance as a preference

1 Open the Preferences dialog.

2 Select Flash Builder > Data/Services

3 Enable the check box for using a single server instance. Click OK.

Building the client application

You use the MXML code editor to create a user interface.

After defining the components for the application using the code editor, you can bind data returned from the service to user interface components. Generate event handlers as needed for user interaction with the application.

You can also generate a Form from service operations that are available in the Data/Services view.

Binding service operations to controls

Use the Bind To Data dialog to bind a service operation to a user interface component.

The Bind to Data dialog is available from the Data menu in toolbar in the Data/Services view.

When you bind a service operation to a component, Flash Builder generates MXML and ActionScript code to access the service operation from the client application.

Return types for service operations

When binding a service operation to a control, Flash Builder uses the data type of the data returned by the operation. Often, you configure the return type for a service operation before binding it to a component.

If the return type for a service operation has not been configured, the Bind to Data dialog prompts you to complete this step.

See “Configuring data types for data service operations” on page 25.

Bind a DataGrid control to a service operation (Bind to Data dialog)

This procedure assumes that you have connected to a data service.

1 In the Outline view, select the DataGrid control. Or, place your cursor within the <s:DataGrid> tag in the MXML editor.

2 With the DataGrid selected, open the Bind to Data dialog by selecting Bind to Data from the Flash Builder Data menu.

3 Select New Service Call, then select a Service and Operation.

If you previously bound a service operation to a component, you can use those results. In this case, specify Existing Call Result and select the operation to use.

4 (Optional) Select Change Return Type:

Select Change Return Type if you want to reconfigure the return type for the service operation.

If the return type for the operation has not been previously configured, select Configure Return Type.

See "Configuring data types for data service operations" on page 25.

5 Click OK.

The DataGrid component changes to show the fields retrieved from the database.

See Configure DataGrid and AdvancedDataGrid components.

6 Save and run the application.

Generating a service call to an operation

Flash Builder can generate an ActionScript method that calls an operation of a service. The method is not bound to a user interface component, but is available to use in your application code.

In addition to generating the ActionScript method, Flash Builder creates a CallResponder that provides access to the data returned from the service call. See "Call Responder" on page 35.

Generate a service call to an operation

This procedure assumes that you have connected to a data service.

1 In Data/Services view, select an operation.

2 From the context menu for the operation, select Generate Service Call.

Flash Builder generates a method for calling the operation and displays the generated method in Source mode of the MXML editor. Flash Builder creates a CallResponder to hold the results of the service call.

This option is also available from the Data/Services toolbar.

Generating a Form for an application

Forms are one of the most common methods that web applications use to collect information from users. Flash Builder can generate forms for data retrieved from service calls or for custom data types used to access remote data.

When generating a form, Flash Builder creates a Form layout container and adds components to display or edit the specific data retrieved from the service.

Flash Builder generates the following types of forms.

Form	Description
Data type	Contains components representing the fields of a data type.
Master-detail Form	The "master" component is typically a data control listing data retrieved from a service. The "detail" form represents individual items selected in the master component.
Service call	Create two forms. One form specifies the inputs to an operation. The other form displays the returned data.

When generating a form, specify which fields to include, the type of user interface control to represent each field, and whether to make the form editable.

Generating a form

This procedure shows how to generate a form for a service call. The procedures for generating other types of forms are similar.

1 To run the Generate Form wizard, fom the Data/Services view, select an operation. Then, do one of the following:

- From the context menu for the operation, select Generate Form.
- From the Flash Builder Data menu, select Generate Form.

2 In the Generate Form wizard, select Generate Form for Service Call.

3 Select New Service Call or Existing Call Result.

Specify Existing Call Result to use the code previously generated for a service call.

Otherwise, specify New Service Call and select a Service and Operation for the Form.

4 (Optional) Depending on the operation, you have several options on the Form that is generated.

If the Operation accepts parameters, then you can choose to include a form for the parameters.

If the Operation returns a value, then you can choose to include a form for the return value.

You can choose whether to make the form editable or read-only.

5 (Optional) Configure the Input Types or Return Types.

If the operation you select has input parameters or returns a value, you can configure the input type or return types.

Configure the input types and return type for the operation before you can generate the form. If you previously configured these types, you have the option to configure them again.

See "Configuring data types for data service operations" on page 25.

6 Click Next. In the Property Control Mapping dialog, select which fields to include in the form and the type of control to represent the data.

7 Click Finish.

When Flash Builder generates Forms, one Form can be placed on top of another Form. To rearrange the generated Forms, make sure that you select and move a Form, and not a component of the Form.

When Flash Builder places one Form on top of another, it can be confusing on how to select the Form. In the code editor, select the tag for one of the Forms.

Generating a master-detail form

To create a master-detail form, first add a data control component to the application and bind the results of an operation to the control.

For example, add a DataGrid component and bind the results of an operation such as `getItems_paged()` to the DataGrid.

1 In the Outline view, select a data control, such as a DataGrid.

2 From the Data menu, select Generate Details Form.

3 Continue generating the Form, as described in Generating a Form.

Generating a form for a data type

To generate a form with components representing the fields of a custom data type, first configure the data type. See "Configuring data types for data service operations" on page 25.

1 In the Data/Services view, select a custom data type.

2 From the context menu, select Generate Form.

3 Make sure that Generate Form for Data Type is selected.and choose a Data Type.

4 (Optional) You can choose whether to make the form editable.

5 Click Finish.

Generating event handlers to retrieve remote data

When you bind a data service operation to a component, Flash Builder generates an event handler that retrieves data from the service to populate the component.

For example, if you bind an operation such as `getAllItems()` to a DataGrid, Flash Builder generates a `creationComplete` event handler. The DataGrid references the generated event handler. The results of the call become the data provider for the DataGrid.

```
. . .
protected function dataGrid_creationCompleteHandler(event:FlexEvent):void
{
    getAllItemsResult.token = productService.getAllItems();
}
. . .
<mx:DataGrid creationComplete="dataGrid_creationCompleteHandler(event)"
dataProvider="{getAllItemsResult.lastResult}">
. . .
</mx:DataGrid>
. . .
```

When you run the application, after the DataGrid has been created the event handler populates the DataGrid with data retrieved from the service.

When generating event handlers, you can accept the generated handlers or replace them with others according to your needs. For example, you can replace the `creationComplete` event handler on the DataGrid with a `creationComplete` handler on the Application.

You can also generate or create event handlers for controls that accept user input, such as Buttons or Text.

Generate an event handler for a user interface component

1 Create an application that contains a user interface component, such as a DataGrid or a Button.

2 Flash Builder provides Content Assist to help you generate the event handler. Press Control+Space or Cmd+Space (Mac) and select Generate Event Handler.

3 Flash Builder generates a unique name for the event handler and places the event handler in the Script block.

Flash Builder highlights the generated stub for the event handler in the code editor. Fill in the remaining code for the event handler. Use Content Assist to help you code the event handler.

Configuring data types for data service operations

When connecting to a data service, Flash Builder needs to know the data type for the data returned by a service operation. The data types supported are those types recognized by AMF to exchange data with a data service or remote service.

Many data services define the type of returned data on the server (server-side typing). However, if the server does not define the type, then the client application must configure the type for returned data (client-side typing).

Service operations that specify parameters must also specify a type corresponding to data accessed on the service. With client-side typing, you configure the type for input parameters.

When configuring types for client-side typing, Flash builder recognizes only AMF data types. The type can also be a custom data type representing complex data, or void to indicate the operation does not return any data.

You can configure user-defined types for service operations that return complex data. For example, if you are retrieving records from an employee database, then you would define a complex data return as Employee. In this case, the custom data type for Employee would contain entries for each field in the database record.

Data Types for client side typing

Data Type	Description
ActionScript types	Boolean Boolean[] ByteArray ByteArray[] Date Date[] int int[] Number Number[] Object Object[] String String[]
No data returned	void
User-defined type	*CustomType* *CustomType*[]

User-defined type (Employee)

Field	Data Type
emp_no	Number
first_name	String
last_name	String
hire_date	Date
birth_date	Date

Authenticating access to services

Typically data services require user authentication before allowing access to the services. PHP, BlazeDS, and ColdFusion services that provide access using the HTTP protocol can require additional authentication. In some cases, these types of services require both HTTP and remote authentication.

Flash Builder provides an option for service authentication when you are doing the following:

- Configuring the return type for an operation

  See "Configuring the return type for data from an operation" on page 27.

- Using the Test Operation interface

  See "Testing service operations" on page 29.

When you specify Authentication Required, Flash Builder opens the Service Authentication dialog. Depending on the type of service you are accessing, you can specify Basic Authentication or Remote Authentication.

Basic authentication

Basic authentication provides access to HTTP and web services. Provide the user name and password for access to these services.

Specify Remember Username and Password if you want Flash Builder to use the specified credentials throughout the session.

Remote authentication

Remote authentication provides access to remote object services. Remote object services are services implemented as remote objects using ColdFusion, PHP, BlazeDS, or LiveCycle Data Services.

Flash Builder does not provide the remote authentication login interface for projects that do not implement remote object services.

Provide the user name and password for access to the remote object services.

Specify Remember Username and Password if you want Flash Builder to use the specified credentials throughout the session.

Configuring input parameters to an operation

For client side typing, you configure input parameters to operations available from the data service.

The following procedure assumes that you have connected to a data service in Flash Builder, and the data service has operations that require configurable input parameters.

1. In the Data/Services view, select an operation that contains configurable input parameters. From the context menu for the operation, select Configure Input Types.
2. In the Configure Input Types dialog, for each argument to the operation, select a data type from the list of available type. Click OK.

   If you previously defined custom return data types for the service, those types are available for selection.

For server-side typing, the service specifies the data type for input parameters.

Configuring the return type for data from an operation

A service that defines data types returned by an operation provides server-side typing. If a service does not define the data type returned by an operation, then Flash Builder uses client-side typing to define the returned data type.

Flash Builder introspects the data returned from a service operation to determine the data type. When you configure the return type of an operation, you have two options:

- Auto-detect the Return Type from Sample Data.

  If the service implements server-side typing, Flash Builder detects the data type defined by the service.

  If the service does not implement server-side typing, Flash Builder creates a custom type for the client application. For client-side typing, you provide a name for the custom type. Typically, the name describes the data returned. For example, if the operation returns an array of books from a database table, then you name the type Book.

- Use an existing type

  An existing type can be a type defined by the service, an ActionScript type, or a previously defined custom type.

The procedures Flash Builder uses for introspecting data differs slightly, depending on the type of data service. For example, the procedure to introspect and configure the return type for an HTTP service differs from the procedure for PHP or ColdFusion services.

Merging and changing data types

During the introspection of server data, you can merge fields from another data type or create a data type based on an existing data type. Here are some of the way you can modify a custom data type:

- Use a new name for an existing data type

  Use a new name if you plan to use returned data in different ways in the client application.

  For example, retrieving employee data that can be used in employee summary and employee detail tables in the client application.

- Merge fields

  You can add returned fields to an existing data type. Adding additional fields is useful when associating data from multiple sources. For example, for a JOIN operation that returns data retrieved from multiple database tables.

  Another example is data received from different services. For example, merging Book data received from both an HTTP service and a ColdFusion service.

Configuring a custom data type (PHP or ColdFusion services)

This procedure assumes that you have connected to a data service implemented with PHP or ColdFusion.

1 In the Data/Services view, from the context menu for an operation, select Configure Return Type.

2 If the operation has arguments, enter the argument values. Specify the correct data type for the argument.

3 (New or Modified Custom Type) Select Auto Detect Type of Data Returned by this Operation.

  If the service requires authentication, select Authentication Required and provide credentials as needed. See "Authenticating access to services" on page 26.

  Flash Builder introspects the operation and builds a custom data type.

  Specify a name for the custom data type.

  If you have previously defined a custom data type, you can choose to add the returned fields to the definition of the existing custom data type.

4 (Use Existing Type) Use this option to specify an ActionScript type or a type you previously configured.

5 Click Finish.

Configuring a custom data type (HTTP service)

This procedure assumes that you have connected to an HTTP service.

1 In the Data/Services view, from the context menu for an operation, select Configure Return Type.

2 (New Custom Type) Select Auto Detect Type of Data Returned by this Operation.

  If the service requires authentication, select Authentication Required and provide credentials as needed.

  Flash Builder introspects the operation and builds a custom data type. Choose a method for Flash Builder to pass parameter values for the operation and click Next:

  - (Enter Parameter Values) For each parameter, specify a value.

    You can also specify the data type for a parameter. Flash Builder automatically selects the default data type.

- (Enter Service URL) Enter the URL to the HTML service, including parameters and values in the URL. For example:

  `http://httpserviceaddress/service_operation?param1=94105`

- (Enter XML/JSON Response) Copy the XML/JSON response to the text box.

  Use this option if you are offline, or if the HTTP service is still under development, but you know the response from the server.

3 (New Custom Type, continued) Specify a name for a custom data type or select a node from the returned data.

If you select a node for the returned data, Flash Builder creates a custom data type for data returned for that node.

Indicate if the returned data is returned as an array.

If the service returns an XML file, the Select Root drop-down list is enabled. Select a node from the XML file to specify a data type.

4 (Use Existing Type) Use this option to specify an ActionScript type or a type you previously configured.

5 Click Finish.

Testing service operations

You can use Flash Builder to test service operations and view data returned from an operation. This feature is useful to verify the behavior of services.

Important: *Some operations, such as update and delete, modify data on the server.*

Test a service operation

This procedure assumes you connected to a data service.

1 In the Data/Service view, select an operation in a service. From the context menus, select Test Operation.

The Test Operation view opens, displaying the selected operation. If the operation requires input parameters, the Test Operation view lists the parameters.

2 For any required input parameters, click the Enter Value field and specify a value for the parameter.

If the parameter requires a complex type, click the Ellipsis button in the field to open an Input Argument Editor. Specify the value in editor.

The Input Argument Editor accepts JSON notation for representing complex types in ActionScript.

3 If authentication is needed from the server, select Authentication Required. Click Test.

Provide authentication credentials as needed. See "Authenticating access to services" on page 26.

Flash Builder displays the data returned from the service.

4 (Optional) In the Test Operation view, select additional services and operations that are available to test.

Managing the access of data from the server

Paging Paging is the incremental retrieval of large data sets from a remote service.

For example, suppose you want to access a database that has 10,000 records and then display the data in a DataGrid that has 20 rows. You can implement a paging operation to fetch the rows in 20 set increments. When the user requests additional data (scrolling in the DataGrid), the next page of records is fetched and displayed.

Data management In Flash Builder, data management is the synchronization of updates to data on the server from the client application. Using data management, you can modify one or more items in a client application without making any updates to the server. You then commit all the changes to the server with one operation. You can also revert the modifications without updating any data.

Data management involves coordinating several operations (create, get, update, delete) to respond to events from the client application, such as updating an Employee record.

When enabling data management with Flash Builder, Flash Builder also generates code that automatically updates user interface components. For example, Flash Builder generates code to keep DataGrids synchronized with data on the server.

Enabling paging

You can enable paging for a data service that implements a paging function with the following signature:

```
getItems_paged(startIndex:Number, numItems:Number): myDataType
```

function name	You can use any valid name for the function.
startIndex	The initial row of data to retrieve. Define the data type for startIndex as Number in the client operation.
numItems	The number of rows of data to retrieve in each page. Define the data type for numItems as Number in the client operation.
myDataType	The data type returned by the data service.

When implementing paging from a service, you can also implement a `count()` operation. A `count()` operation returns the number of items returned from the service. Flash Builder requires that the `count()` operation implement the following signature:

```
count(): Number
```

function name	You can use any valid name for the function.
Number	The number of records retrieved from the operation.

Flex uses the count operation to properly display user interface components that retrieve large data sets. For example, the `count()` operation helps determine the thumb size for a scroll bar of a DataGrid.

Some remote services do not provide a `count()` operation. Paging still works, but the control displaying the paged data does not properly represent the size of the data set.

Paging operations for filtered queries

You can enable paging for queries that filter results from the database. When filtering results in the query, use these signatures for the paging and count functions.

```
getItems_pagedFiltered(filterParam1:String, filterParam2:String, startIndex:Number,
numItems:Number): myDataType
```

```
countFiltered(filterParam1:String, filterParam2:String)
```

filterParam1	Optional filter parameter. This parameter is the same in getItems_PagedFiltered() and countFiltered().
filterParam2	Optional filter parameter. This parameter is the same in getItems_PagedFiltered() and countFiltered().

Here is a code snippet of a `getItems_pagedFiltered()` function that is implemented in PHP to access a MySQL database. The code snippet shows how to use the optional filter parameter.

```
get_Items_paged($expression, $startIndex, $numItems) {
    . . .
    SELECT * from employees where name LIKE $expression LIMIT $startIndex, $numItems;
    . . .
}
```

Enable paging for an operation

This procedure assumes that you have coded both `getItems_paged()` and `count()` operations in your remote service. It also assumes that you have configured the return data type for the operation, as explained in "Configuring data types for data service operations" on page 25.

1. In the Data/Services view, from the context menu for the `getItems_paged()` operation, select Enable Paging.
2. If you have not previously identified a unique key for your data type, specify the attributes that uniquely identify an instance of this data type. Click Next.

   Typically, this attribute is the primary key.
3. (Optional) Specify the number of items to fetch to define a custom page size.

   If you do not specify a custom page size, a default page size is set at the service level. The default page size is 20 records per page.
4. (Optional) Specify the `count()` operation. Click Finish.

   The `count()` operation allows Flash Builder to properly display user interface elements, such as the thumb size for a scroll bar.

   Paging is now enabled for that operation.

   In Data/Services View, the signature of the function that implements paging no longer includes the startIndex and numItems parameters. Flash Builder now dynamically adds these values. Flash Builder determines these values based on a custom page size you provided or the default page size of 20 records per page.

Enabling data management

To enable data management for a service, the service implements one or more of the following functions. The Data management feature uses these functions to synchronize updates to data on the remote server:

- Add (createItem)

  ```
  createItem(item: myDatatype):int
  createItem(item: myDatatype):String
  createItem(item: myDatatype):myDataType
  ```

  The return type for createItem() is the type of the primary key of the database.
- Get All Properties (getItem)

  ```
  getItem(itemID:Number): myDatatype
  ```
- Update (updateItem)

```
updateItem((item: myDataType):void
updateItem((item: myDataType, originalItem: myDataType):void
updateItem((item: myDataType, originalItem: myDataType, changes: String[]):void
```

- Delete (deleteItem)

```
deleteItem(itemID:Number):void
```

Flash Builder requires these functions to have the following signatures:

function name	You can use any valid name for the function.
item **originalItem**	An item of the data type returned by the data service.
itemID	A unique identifier for the item, usually the primary key in the database.
changes[]	An array corresponding to fields in a specified Item. This argument is only used in one version of `updateItem()`.
myDataType	The data type of the item available from the data service. Typically, you define a custom data type when retrieving data from a service.

autoCommit flag

Data management allows you to synchronize updates to data on the server. Changes to data made in the client application are not updated on the server until you call the *`service`*`.commit()` method.

However, if you want to disable this feature, set the autoCommit flag to true. If autoCommit is true, then updates to server data are not cached, but are made immediately. See "Enabling data management for a service" on page 38.

deleteItemOnRemoveFromFill flag

When you delete items with data management enabled, use the deleteItemOnRemoveFromFill flag. By default, this flag is set to true. When you delete an item, the item is immediately removed from the database.

Set deleteItemOnRemoveFromFill to false to defer the deletion until the commit() method is called. The following example shows the code for a creation complete event handler for a DataGrid. If the user deletes a selected Employee item in the DataGrid, the selected item is not removed from the database until the commit() method is called.

```
protected function dg_creationCompleteHandler(event:FlexEvent):void
{
    employeeService.    getDataManager(employeeService.DATA_MANAGER_EMPLOYEE).autoCommit=false;
    employeeService.getDataManager(  e m p l
oyeeService.DATA_MANAGER_EMPLOYEE).deleteItemOnRemoveFromFill=      true;
    getAllEmployeesResult.token = employeeService.getAllEmployees();
}
```

Enable data management for an operation

This procedure assumes that you have implemented the required operations in your remote service. It also assumes that you have configured the return data type for the operations that use a custom data type. See "Configuring data types for data service operations" on page 25.

1. In the Data/Services view, expand the Data Types node.
2. From the context menu for a data type, select Enable Data Management.
3. If you have not previously identified a unique key for your data type, specify the attributes that uniquely identify an instance of this data type. Click Next.

Typically, this attribute is the primary key.

4 Specify the Add, Get All Properties, Update, and Delete operations that you have implemented. Click Finish.

Note: *You do not have to implement all of these functions. Implement only those functions required for your application.*

Data management is now enabled for that operation.

Flash Builder code generation for client applications

Flash Builder generates client code that provides access to remote service operations. Flash Builder generates code in the following circumstances:

- Connecting to a data service
- Refreshing the data service in Data/Services View
- Configuring a return type for an operation
- Binding a service operation to a user interface control
- Enabling paging for a service operation
- Enabling data management for a service
- Generating an event handler or a service call

Service classes

Use the Service wizard to connect to a data service. When you connect to a service, Flash Builder generates an ActionScript class file that provides access to the service operations.

For services that access a RemoteObject, the generated class extends the RemoteObjectServiceWrapper class. Services implemented with PHP, ColdFusion, BlazeDS, and LiveCycle Data Services typically access a RemoteObject.

For HTTP services, the generated class extends the HTTPServiceWrapper class.

For web services, the generated class extends WebServiceWrapper class.

Flash Builder bases the name of the generated class file on the name you provided for the service in the Service wizard. By default, Flash Builder places this class in the main source folder for a project. Typically, this folder is `src`. The name of the package is based on the service name. For example, Flash Builder generates the following ActionScript classes for an EmployeeService class.

```
- project
    |
    - src
        |
        + (default package)
        |
        + services
        |   |
        |   +employeeservice
        |       |
        |       + _Super_EmployeeService.as
        |       |
        |       + EmployeeService.as
```

The super class contains the implementation for the EmployeeService class.

Never edit the super class, which is a generated class. Modifications you make to the super class can be overwritten. Any changes you make to the implementation can result in undefined behavior.

In this example, use `EmployeeService.as` to extend the generated super class and add your implementation.

More Help topics

"Connecting to data services" on page 8

Classes for custom data types

Many remote data services provide server-side typing. These services return complex data as a custom data type.

For remote data services that do not return typed data, Flash Builder provides client-side typing. With client-side typing, you use the Flash Builder Connect wizard to define and configure the data type for complex data returned by the service. For example, for a service that returns employee database records, you define and configure an Employee data type.

Flash Builder generates an ActionScript class for the implementation of each custom data type returned by the service. Flash Builder uses this class to create value objects, which it then uses to access data from the remote service.

For example, Flash Builder generates the following ActionScript classes for an EmployeeService class that contains an Employee data type:

```
- project
    |
    - src
        |
        + (default package)
        |
        + services
        |   |
        |   +employeeservice
        |       |
        |       + _Super_EmployeeService.as
        |       |
        |       + EmployeeService.as
        |
        + valueObjects
            |
            + _Super_Employee.as
            |
            + Employee.as
```

The super classes contain the implementation for the EmployeeService and the Employee data type, respectively.

Never edit a generated super class. Modifications you make to the super class can be overwritten. Any changes you make to the implementation can result in undefined behavior.

In this example, use `EmployeeService.as` and `Employee.as` to extend the generated super class and add your implementation.

Binding a service operation to a user interface control

"Binding service operations to controls" on page 21 shows how you can bind data returned from service operations to a user interface control. When you bind a service operation to a control, Flash Builder generates the following code:

- Declarations tag containing a CallResponder and service tag
- Event handler for calling the service call
- Data binding between the control and the data returned from the operation

Declarations tag

A Declarations tag is an MXML element that declares non-default, non-visual properties of the current class. When binding a service operation to a user interface, Flash Builder generates a Declarations tag containing a CallResponder and a service tag. The CallResponder and generated service class are properties of the container element, which is usually the Application tag.

The following example shows a Declarations tag providing access to a remote EmployeeService:

```
<fx:Declarations>
    <s:CallResponder id="getAllEmployeesResult"/>
    <employeesservice:EmployeesService id="employeesService"
        fault="Alert.show(event.fault.faultString + '\n'
        + event.fault.faultDetail)" showBusyCursor="true"/>
</fx:Declarations>
```

Call Responder

A CallResponder manages results for calls made to a service. It contains a token property that is set to the Async token returned by a service call. The CallResponder also contains a lastResult property, which is set to the last successful result from the service call. You add event handlers to the CallResponder to provide access to the data returned through the lastResult property.

When Flash Builder generates a CallResponder, it generates an id property based on the name of the service operation to which it is bound. The following code sample shows CallResponders for two operations of an EmployeeService. The `getAllItems()` operation is bound to the creationComplete event handler for a DataGrid. The delete operation is bound to the selected item in the DataGrid. The DataGrid displays the items retrieved from the `getAllItems()` service call immediately after it is created. The Delete Item Button control removes the selected record in the DataGrid from the database.

```
<fx:Script>
    <![CDATA[
        import mx.events.FlexEvent;
        import mx.controls.Alert;

        protected function dg_creationCompleteHandler(event:FlexEvent):void
        {
            getAllItemsResult.token = employeesService.getAllItems();
        }

        protected function button_clickHandler(event:MouseEvent):void
        {
            deleteItemResult.token =
                employeesService.deleteItem(dg.selectedItem.emp_no);
        }
    ]]>
</fx:Script>

<fx:Declarations>
    <s:CallResponder id="getAllItemsResult"/>
    <employeesservice:EmployeesService id="employeesService"
        fault="Alert.show(event.fault.faultString + '\n'
        + event.fault.faultDetail)" showBusyCursor="true"/>
    <s:CallResponder id="deleteItemResult"/>
</fx:Declarations>
<mx:DataGrid id="dg" editable="true"

creationComplete="dg_creationCompleteHandler(event)"dataProvider="{getAllItemsResult.lastRes
ult}">
    <mx:columns>
        <mx:DataGridColumn headerText="emp_no" dataField="emp_no"/>
        <mx:DataGridColumn headerText="last_name" dataField="last_name"/>
        <mx:DataGridColumn headerText="hire_date" dataField="hire_date"/>
    </mx:columns>
</mx:DataGrid>
<s:Button label="Delete Item" id="button" click="button_clickHandler(event)"/>
```

Event handlers

When you bind a service operation to a user interface component, Flash Builder generates an event handler for the CallResponder. The event handler manages the results of the operation. You can also create an event handler in an ActionScript code block, and reference that event handler from a property of a user interface component.

Typically, you populate controls such as Lists and DataGrids with data returned from a service. Flash Builder, by default, generates a creationComplete event handler for the control that fires immediately after the control is created. For other controls, Flash Builder generates a handler for the control's default event. For example, for a Button, Flash Builder generates an event for the Button's click event.

The control's event property is set to the generated event handler. The following example shows the generated creation complete event handler for a DataGrid:

```
<fx:Script>
    <![CDATA[
        import mx.events.FlexEvent;
        import mx.controls.Alert;

        protected function dg_creationCompleteHandler(event:FlexEvent):void
        {
            getAllItemsResult.token = employeesService.getAllItems();
        }
    ]]>
</fx:Script>
. . .

<mx:DataGrid id="dg" editable="true"
        creationComplete="dg_creationCompleteHandler(event)"
        dataProvider="{getAllItemsResult.lastResult}">
    <mx:columns>
        <mx:DataGridColumn headerText="emp_no" dataField="emp_no"/>
        <mx:DataGridColumn headerText="last_name" dataField="last_name"/>
        <mx:DataGridColumn headerText="hire_date" dataField="hire_date"/>
    </mx:columns>
</mx:DataGrid>
```

You can generate event handlers for controls that respond to user events, such as Buttons. The following example shows a generated event handler for a Button that populates a DataGrid:

```
<fx:Script>
    <![CDATA[
        import mx.events.FlexEvent;
        import mx.controls.Alert;

        protected function button_clickHandler(event:MouseEvent):void
        {
            deleteItemResult.token =
                employeesService.deleteItem(dg.selectedItem.emp_no);
        }
    ]]>
</fx:Script>
. . .

<s:Button label="Delete Item" id="button" click="button_clickHandler(event)"/>
```

Data binding

When you build a user interface, you bind service operations to controls. See "Binding service operations to controls" on page 21.

Flash Builder generates code that binds the data returned from a service operation to the user interface control that displays the data.

The following example shows the code that Flash Builder generates to populate a DataGrid control. The `getAllItems()` operation returns a set of employee records for the custom data type, Employee.

The dataProvider property of the DataGrid is bound to the results stored in the CallResponder, getAllItemsResult. Flash Builder automatically updates the DataGrid with DataGridColumns corresponding to each field returned for an Employee record. The headerText and dataField properties of each column are set according to the data returned in an Employee record.

```
<mx:DataGrid creationComplete="datagrid1_creationCompleteHandler(event)"
    dataProvider="{getAllItemsResult.lastResult}" editable="true">
        <mx:columns>
            <mx:DataGridColumn headerText="gender" dataField="gender"/>
            <mx:DataGridColumn headerText="emp_no" dataField="emp_no"/>
            <mx:DataGridColumn headerText="birth_date" dataField="birth_date"/>
            <mx:DataGridColumn headerText="last_name" dataField="last_name"/>
            <mx:DataGridColumn headerText="hire_date" dataField="hire_date"/>
            <mx:DataGridColumn headerText="first_name" dataField="first_name"/>
        </mx:columns>
    </mx:DataGrid>
```

Enabling paging for a service operation

When you enable paging, Flash Builder modifies the implementation of the generated service. When you populate a data control (such as a DataGrid or a List) with paged data, Flash Builder determines the number of records visible in the data control and the total number of records in the database. Flash Builder provides these values as arguments to the service operation that you used to implement paging.

You do not have to modify any client application code after paging is enabled.

See "Enabling paging" on page 30 for more information.

Enabling data management for a service

In Flash Builder, data management is the synchronization of a set of updates to data on the server. You can enable data management for custom data types returned from the service. With data management enabled, you can modify one or more items in a client application without making any updates to the server. You can then commit all the changes to the server with one operation. You can also revert the modifications without updating any data on the server. "Enabling data management" on page 31 shows how to implement this feature.

When you enable data management, Flash Builder modifies the implementation of the generated service class and the generated class for custom data types. Flash Builder creates a DataManager to implement this functionality.

Synchronizing updates to server data

When you call service operations for a managed data type, the changes are reflected in the client application. However, you can specify that data on the server is not updated until you call the DataManager's `commit()` method.

When data management is enabled for a service, the service has an autoCommit flag. By default, autoCommit is false.

The autoCommit flag determines whether to commit changes immediately or to wait until *`service`*`.commit()` is called.

If autoCommit is false, all updates to the service in the client application are cached until you call *`service`*`.commit()`. You can call the service's `revertChanges()` method to discard changes.

If autoCommit is true, then updates are sent to the server immediately. You cannot call `revertChanges()` to discard changes.

The deleteItemOnRemoveFromFill flag determines whether a deleted item is immediately removed from the database. If set to true, then the item is not deleted until *`service`*`.commit()` is called.

The following code disables data management synchronization of updates to server data. Changes to data for the managed type are updated immediately on the server.

```
bookService.getDataManager(bookService.DATA_MANAGER__BOOK).autoCommit = true;
```

The following code enables data management synchronization of updates to server data. Changes to data for the managed type are not updated until `commit()` is called for the service. Additionally, deleted items are not removed from the database until `commit()` is called.

```
bookService.getDataManager(bookService.DATA_MANAGER__BOOK).autoCommit = false;
bookService.getDataManager(bookService.DATA_MANAGER__BOOK).deleteItemOnRemoveFromFill= true;
```

Reverting changes

The DataManager provides a `revertChanges()` method. The `revertChanges()` method restores the data displayed in the client application to the values retrieved from the server before the last commit call.

Call `revertChanges()` before calling `commit()` to reverts changes to a managed data type in the client application:

```
bookService.getDataManager (bookService.DATA_MANAGER_BOOK).revertChanges();
```

To commit changes made for the managed data type, call the `commit()` method.

```
bookService.getDataManager (employeeService.DATA_MANAGER_EMPLOYEE).commit();
```

You can also call the `commit()` method directly from the bookService instance. Calling the commit method directly from the service instance commits all changes for all managed data types.

```
bookService.commit();
```

Note: *You cannot call `revertChanges()` directly from the service instance to revert changes to all managed data types. You can only call it for a specific managed data type.*

If you want to override the default behavior for data management, and disable the ability to revert changes, set the autoCommit to true. For example, if you have an instance of bookService and enabled data management for the Book data type, set autoCommit to true:

```
bookService.getDataManager(bookService.DATA_MANAGER__BOOK).autoCommit = true;
```

Now, changes made to data for the managed type are updated immediately on the server.

Deploying applications that access data services

There are many factors to consider when moving your application from a development environment to a deployment environment. The process of deploying an application is dependent on your application, your application requirements, and your deployment environment.

For example, the process of deploying an application on an internal website that is only accessible by company employees is different from the process for deploying the same application on a public website.

Deploying applications provides an overview of things to consider and includes a Deployment checklist. The checklist discusses some common system configuration issues that customers have found when deploying applications for production. The documentation also contains troubleshooting tips to diagnose common deployment problems.

Best practices for coding access to services

Using Flash Builder tools, you can generate client code to access data in a database. This feature is available for both PHP and ColdFusion. However, this code is for prototyping only. Do not use this code as a template for writing secure applications.

By default, this code allows anyone with network access to your server to insert, select, update, or delete from the database table. Here are some best practices to consider when modifying the generated code or writing any code that accesses services. See Securing Data Services for additional information.

Remove functions that are not used

Delete or comment out any functions that you do not plan to use within your application.

Add authentication

Add user authentication to ensure that only trusted users can access your database information.

Add authorization checks

If authentication is necessary, then add authorization checks. Even though you authenticated users to your application, you want to ensure that they are authorized to make specific queries.

For example, you can allow everyone to select but restrict which users have the authority to delete.

Another example is authorizing user A to retrieve their own information using a select query. But prevent user A from using a select query to access user B's information.

Data validation

Be sure to add data validation. For example, validate the data provided to any insert statement to ensure that bad or malicious data does not get accepted by the database.

Client side validation is not able to protect you from someone sending manual requests to your web server. Data validation protects against hackers and ensures the quality of the information that is stored.

Restrict the amount of data that is retrieved

Select methods can select everything from a table. In some cases, this practice leads to too much information going over the network. Only retrieve the data that you need.

For example, `SELECT *` from a user table can return the user name and password over the network.

Consider SSL for sensitive data

Using a secure protocol ensures the privacy of the information you are sending.

Exporting source files with release version of an application

When you export a release build of an application, Flash Builder provides the option Enable View Source. This option allows users to view the source files that implement the application. For server projects, the source files include the `services` folder, which contains the files providing access to your service implementation.

Important: *Use caution when including service files with the View Source option. The service files expose details on accessing your database, and possibly include sensitive information such as user names and password. If services are included in the View Source option, anyone who has access to the launched application has access to sensitive data.*

More Help topics

Flex Security

Writing secure services

The examples in Adobe documentation, including the tutorials and applications created using Flash Builder code generation, are instructive in nature. They illustrate how to access data services from a client application. However, because these examples are designed to ensure clarity, they do not illustrate best practices for secure access to data.

The Flash Builder documentation contains examples, including applications created from generated code. These examples are to be deployed in a trusted development environment. A trusted development environment can be a local machine or location inside a firewall. Without additional security measures, anyone with network access also has access to your database.

Some best practices when writing services include:

- Authenticate the user before calling any method on a service
- Use service authentication to allow only certain users to perform certain actions.

  For example, suppose you have an application that allows employee data to be modified through a RemoteObject call. In this case, use RemoteObject authentication to make sure that only managers can change the employee data.
- Use programmatic security to limit access to services.
- Apply declarative security constraints to entire services.
- When accessing a web service (`<mx:WebService>`) or HTTP service (`<mx:HTTPService>`) one of the following must be true:
  - The service implementation is in the same domain as the application that calls it.
  - The host system for the service has a `crossdomain.xml` file that explicitly allows access from the application's domain.

More Help topics

Flex Security

Securing Data Services

Writing secure applications

Adobe® Flash® Player runs applications built with Flash. Content is delivered as a series of instructions in binary format to Flash Player over web protocols in a precisely described SWF file format. The SWF files themselves are typically hosted on a server and then downloaded to, and displayed on, the client computer when requested. Most of the content consists of binary ActionScript instructions. ActionScript is the ECMA standards-based scripting language that Flash uses. ActionScript features APIs designed to allow the creation and manipulation of client-side user interface elements and for working with data.

The security model for Flex protects both client and the server. Consider the following two general aspects to security:

- Authorization and authentication of users accessing a server's resources
- Flash Player operating in a sandbox on the client

Flex supports working with the web application security of any J2EE application server. In addition, precompiled applications in Flex can integrate with the authentication and authorization scheme of any underlying server technology to prevent users from accessing your applications. The Flex framework also includes several built-in security mechanisms that let you control access to web services, HTTP services, and server-based resources such as EJBs.

Flash Player runs inside a security sandbox that prevents hijacking of the client by malicious application code.

Note: *SWF content running in Adobe AIR follows different security rules than content running in the browser. For details, see the Air Security topic in the AIR documentation.*

For links to various security topics, see the Security Topic Center at the Adobe Developer Connection.

More Help topics

Flex Security

Chapter 3: Implementing services for data-centric applications

Action Message Format (AMF)

Flex uses remote object services and AMF to access services implemented in ColdFusion, PHP, BlazeDS, and LiveCycle Data Services. AMF provides the messaging for exchanging data between a database and the client application.

ColdFusion, BlazeDS, and LiveCycle Data Services each provide an AMF framework for remote object services. For services implemented in PHP, Flash Builder uses the Zend AMF framework.

ColdFusion and PHP services can provide server-side typing. In server-side typing, the service defines the type of data returned. However, if the service implementation does not define the return data type, Flash Builder provides client-side typing. Flash Builder samples data from the service, allowing you to configure the return type in the client application.

Client-side and server-side typing

In Flex, a client application uses the data type of data returned from a service call in methods that access and display the data.

However, the services examples listed below return untyped data.

- "Implementing ColdFusion services" on page 43
- "Implementing PHP services" on page 50
- "Example implementing services from multiple sources" on page 64

For example, for the `getAllEmployees()` operation, the service returns an array of untyped objects that represent records from the database. Flash Builder provides tools that enable client-side typing. Using Flash Builder tools, you introspect the returned data and define a custom data type for the data.

For the returned object of employee records, you define the custom data type, Employee. Each column of the record becomes a property of the Employee data type.

Using the Employee custom data type, the client application can access the returned data and display it properly in the client application.

Flash Builder can also access services that implement server-side typing. For examples of server-side typing, see Flash Builder server-side type examples.

Implementing ColdFusion services

When implementing services in ColdFusion, implement the services as ColdFusion component (CFC) files. Each CFC contains functions that provide the service operations.

You can create ColdFusion services in any IDE, such as Adobe ColdFusion® Builder™. Flash Builder does not provide an editor optimized for editing ColdFusion files. However, if you open a ColdFusion file in Flash Builder, Flash Builder launches the application on your system that is associated with ColdFusion files.

Example ColdFusion services

You can implement a basic ColdFusion service by creating a ColdFusion component (CFC) that contains functions for the service operations. The following example, `employeeService.cfc`, illustrates an EmployeeService that implements two functions. The `getAllIEmployees()` function retrieves all employee records in the database. The `getEmployees()` function returns a single employee record based on the `emp_no` argument to the function.

This example illustrates client-side typing. The service returns untyped data. Flash Builder uses client-side typing to introspect the returned data and define a data type.

Subsequent examples illustrate how to implement services for paging and data management.

You can also use Flash Builder to access services that implement server-side typing. See "Client-side and server-side typing" on page 43.

Examples illustrating server-side typing were not available at the time this document was completed. For server-side typing examples, see Flash Builder server-side type examples.

ColdFusion example implementing a basic service

This example shows how to implement a basic service in ColdFusion. The example is based on code that Flash Builder generates when accessing a database table. See "Generating a sample ColdFusion service from a database table" on page 9.

This example implements client-side typing. See "Client-side and server-side typing" on page 43.

For examples of server-side typing, see Flash Builder server-side type examples.

Important: *Example services are for prototyping only. Use the example service only in a trusted development environment. Before deploying this service, be sure to increase security and restrict access appropriately. For information on writing secure ColdFusion services, see the ColdFusion documentation About User Security.*

```
<cfcomponent output="false">

<!---
    This sample service contains functions that illustrate typical service operations.
    This code is for prototyping only.

    Authenticate the user prior to allowing them to call these methods. You can find more
    information at http://www.adobe.com/go/cf9_usersecurity

--->
    <cffunction name="getAllemployees" output="false" access="remote" returntype="any" >

        <!--- Retrieve set of records and return them as a query or array.
                Add authorization or any logical checks for secure access to your data --->

        <cfset var qAllItems="">
        <cfquery name="qAllItems" datasource="employees">
            SELECT * FROM employees
        </cfquery>
        <cfreturn qAllItems>
```

```
    </cffunction>

    <cffunction name="getemployees" output="false" access="remote"  returntype="any" >
        <cfargument name="emp_no" type = "numeric" required="true" />

        <!--- Retrieve a single record and return it as a query or array.
                Add authorization or any logical checks for secure access to your data --->

        <cfset var qItem="">
        <cfquery name="qItem" datasource="employees">
            SELECT *
            FROM employees
            WHERE emp_no = <CFQUERYPARAM CFSQLTYPE="CF_SQL_INTEGER"
VALUE="#ARGUMENTS.emp_no#">
        </cfquery>

        <cfreturn qItem>

    </cffunction>

</cfcomponent>
```

Highlights of EmployeeService:

- Connects to the employees database and accesses the employees table in the database.
- Returns an array of objects.

  When programming using the Flex framework, services return data only. The client application handles the formatting and presentation of the data. This model differs from traditional ColdFusion CFM applications that return data formatted in an HTML template.

  Flex handles returned recordsets as an array of objects. Each row represents a retrieved record from the database. Each column of the database record becomes a property of the returned object. The client application can now access the returned data as objects with a set of properties.

  Configure the data type for the returned object. See "Client-side and server-side typing" on page 43.
- The ColdFusion server provides error handling.

  The error handling provided by ColdFusion is useful when debugging a service. In the ColdFusion Administrator, modify the Debugging and Logging settings to provide robust debugging information.

  The Flash Builder Test Operation interface displays information returned by ColdFusion server.

  See "Debugging remote services" on page 61 for more information on testing services.
- Uses `cfqueryparam` for constructing database queries.

  `cfqueryparam` is a defense against SQL injection statements in calls to the server. For more information, see Enhancing security with cfqueryparam in the ColdFusion documentation.
- Authenticate users before providing access to the functions in this service.

  The sample code does not illustrate how to authenticate users. See the ColdFusion documentation, About User Security.

More Help topics

"Configuring data types for data service operations" on page 25

"Accessing ColdFusion services" on page 8

"Generating a sample ColdFusion service from a database table" on page 9

ColdFusion example implementing paging

Flash Builder tools allow you to implement paging of data retrieved from a remote service. Paging is the incremental retrieval of large data sets.

Flash Builder requires specific function signatures to implement paging. The following code example provides an example of one way to implement a ColdFusion service for paged data.

The EmployeeServicePaged example is based on the code generated by Flash Builder when accessing a database table. See "Generating a sample ColdFusion service from a database table" on page 9.

Important: *Example services are for prototyping only. Use the example service only in a trusted development environment. The example allows anyone with network access to your server to be able to access, modify, or delete data in the database table. Before deploying this service, be sure to increase security and restrict access appropriately. For information on writing secure ColdFusion services, see the ColdFusion documentation About User Security.*

```
<cfcomponent output="false">

<!---
    This sample service contains functions that illustrate typical service operations.
    This code is for prototyping only.

    Authenticate the user prior to allowing them to call these methods. You can find more
    information at http://www.adobe.com/go/cf9_usersecurity
--->
    <cffunction name="count" output="false" access="remote" returntype="numeric" >

        <!---   Return the number of items in your table.
                Add authorization or any logical checks for secure access to your data --->
        <cfquery name="qread" datasource="employees">
            SELECT COUNT(emp_no) AS itemCount FROM employees
        </cfquery>

        <cfreturn qread.itemCount>

    </cffunction>

    <cffunction name="getemployees_paged" output="false" access="remote" returntype="any" >
        <cfargument name="startIndex" type="numeric" required="true" />
        <cfargument name="numItems" type="numeric" required="true" />

    <!---Return a page of numRows number of records as an array or
        query from the database for this startRow.
        Add authorization or any logical checks for secure access to your data --->
    <!---The LIMIT keyword is valid for mysql database only.
        Modify it for your database --->

        <cfset var qRead="">
        <cfquery name="qRead" datasource="employees">
            SELECT * FROM employees LIMIT #startIndex#, #numItems#
        </cfquery>
        <cfreturn qRead>

    </cffunction>
</cfcomponent>
```

The EmployeeServicePaged service returns untyped data. Use the Flash Builder tools to configure the return type for `getEmployees_Paged()`. After configuring the return type, enable paging on the `getEmployees_Paged()` operation.

More Help topics

"Example ColdFusion services" on page 44

"Configuring data types for data service operations" on page 25

"Managing the access of data from the server" on page 29

ColdFusion example implementing data management operations

Flash Builder tools allow you to implement data management functionality for remote services. Data management is the synchronization of updates to data on a server from the client application.

Flash Builder requires a combination of specific function signatures to implement data management. The following code example provides an example of one way to implement a ColdFusion service for data management.

The following EmployeeServiceDM example is based on the code generated by Flash Builder when accessing a database table. See "Generating a sample ColdFusion service from a database table" on page 9.

Important: *Example services are for prototyping only. Use the example service only in a trusted development environment. Before deploying this service, be sure to increase security and restrict access appropriately. For information on writing secure ColdFusion services, see the ColdFusion documentation About User Security.*

```
<cfcomponent output="false">

<!---
    This sample service contains functions that illustrate typical service operations.
    This code is for prototyping only.

    Authenticate the user prior to allowing them to call these methods. You can find more
    information at http://www.adobe.com/go/cf9_usersecurity
--->
    <cffunction name="getAllemployees" output="false" access="remote" returntype="any" >

        <!--- Auto-generated method
                Retrieve set of records and return them as a query or array.
                Add authorization or any logical checks for secure access to your data --->

        <cfset var qAllItems="">
        <cfquery name="qAllItems" datasource="employees">
            SELECT * FROM employees
        </cfquery>
        <cfreturn qAllItems>

    </cffunction>

    <cffunction name="getemployees" output="false" access="remote"  returntype="any" >
        <cfargument name="emp_no" type = "numeric" required="true" />

        <!---
                Retrieve a single record and return it as a query or array.
                Add authorization or any logical checks for secure access to your data --->

        <cfset var qItem="">
        <cfquery name="qItem" datasource="employees">
            SELECT *
            FROM employees
            WHERE emp_no = <CFQUERYPARAM CFSQLTYPE="CF_SQL_INTEGER"
VALUE="#ARGUMENTS.emp_no#">
        </cfquery>

        <cfreturn qItem>

    </cffunction>

    <cffunction name="createemployees" output="false" access="remote" returntype="any" >
        <cfargument name="item" required="true" />

        <!---   Insert a new record in the database.
            Add authorization or any logical checks for secure access to your data --->

        <cfquery name="createItem" datasource="employees" result="result">
            INSERT INTO employees (birth_date, first_name, last_name, gender, hire_date)
```

```
            VALUES (<CFQUERYPARAM cfsqltype="CF_SQL_DATE" VALUE="#item.birth_date#">,
                            <CFQUERYPARAM cfsqltype="CF_SQL_VARCHAR"
VALUE="#item.first_name#">,
                            <CFQUERYPARAM cfsqltype="CF_SQL_VARCHAR"
VALUE="#item.last_name#">,
                            <CFQUERYPARAM cfsqltype="CF_SQL_CHAR" VALUE="#item.gender#">,
                            <CFQUERYPARAM cfsqltype="CF_SQL_DATE" VALUE="#item.hire_date#">)

        </cfquery>

        <!--- The GENERATED_KEY is valid for mysql database only, you can modify it for your
database --->
        <cfreturn result.GENERATED_KEY/>

    </cffunction>

    <cffunction name="updateemployees" output="false" access="remote" returntype="void" >
        <cfargument name="item" required="true" />

        <!--- Update an existing record in the database.
              Add authorization or any logical checks for secure access to your data --->

         <cfquery name="updateItem" datasource="employees">
            UPDATE employees SET birth_date = <CFQUERYPARAM cfsqltype=CF_SQL_DATE
VALUE="#item.birth_date#">,
                             first_name = <CFQUERYPARAM cfsqltype=CF_SQL_VARCHAR
VALUE="#item.first_name#">,
                             last_name = <CFQUERYPARAM cfsqltype=CF_SQL_VARCHAR
VALUE="#item.last_name#">,
                             gender = <CFQUERYPARAM cfsqltype=CF_SQL_CHAR
VALUE="#item.gender#">,
                             hire_date = <CFQUERYPARAM cfsqltype=CF_SQL_DATE
VALUE="#item.hire_date#">

            WHERE emp_no = <CFQUERYPARAM CFSQLTYPE="CF_SQL_INTEGER" VALUE="#item.emp_no#">
        </cfquery>

    </cffunction>

    <cffunction name="deleteemployees" output="false" access="remote" returntype="void" >
        <cfargument name="emp_no" type="numeric" required="true" />

        <!--- Delete a record in the database.
              Add authorization or any logical checks for secure access to your data --->

        <cfquery name="delete" datasource="employees">
            DELETE FROM employees
            WHERE emp_no = <CFQUERYPARAM CFSQLTYPE="CF_SQL_INTEGER"
VALUE="#ARGUMENTS.emp_no#">
        </cfquery>

    </cffunction>
</cfcomponent>
```

The EmployeeServiceDM service returns untyped data. Use the Flash Builder tools to configure the return type for `getAllEmployeess()` and `getEmployees()`. Use Employee for the custom data type returned by these operations.

After configuring the return type, enable data management on the Employee data type.

More Help topics

"Example ColdFusion services" on page 44

"Configuring data types for data service operations" on page 25

"Managing the access of data from the server" on page 29

Generating CFCs using Adobe ColdFusion Builder

Adobe® ColdFusion® Builder™ provides the Adobe CFC Generator. Use CFC Generator to generate an ORM CFC or a traditional CFC from a set of database tables. The CFC generated by ColdFusion Builder can then be used as a data service in Flash Builder. The Adobe CFC Generator creates services that implement server side typing.

For details, see Using Adobe CFC Generator.

Note: *ColdFusion object relational mapping (ORM uses an object model to define a mapping strategy for storing and retrieving data from a relational database. See ColdFusion ORM.*

Implementing PHP services

When implementing services in PHP, you typically implement the services as PHP classes. The classes in PHP do not necessarily have to be object-oriented classes. Rather, each class can be a library of functions that provide the service operations.

You can create PHP services in any editing environment, such as Dreamweaver or Zend Studio. Flash Builder does not provide an editor optimized for editing PHP files. However, if you open a PHP file in Flash Builder, Flash Builder launches the application on your system that is associated with PHP files. For convenience, Flash Builder also provides a plain text editor that you can use to edit the PHP files.

Using AMF to access services implemented in PHP

PHP data services are available using Action Message Format (AMF). AMF provides messaging between a Flash client and a web server. Flash Builder uses the Zend AMF framework to implement AMF messaging for PHP data services.

For information on Zend AMF, see Zend Framework Programmer's Reference.

For information on installing Zend Framework, see "Installing Zend Framework" on page 19.

For information on using Zend with Flash Builder for PHP, see the Zend website.

Note: *Although Flash Builder uses the Zend AMF framework, you are not required to use Zend components when creating PHP services. Although Zend components work well with Flash Builder, you can also use any PHP development environment for creating services.*

Example PHP services

You can implement a basic PHP service by creating a PHP class file that contains functions for the service operations. The following example illustrates an EmployeeService that implements two functions:

- `getAllIEmployees()`

  Retrieves all employee records in the database.

- `getEmployeeByID($itemID)`

  Returns a single employee record.

This example illustrates client-side typing. The service returns untyped data. Flash Builder uses client-side typing to introspect the returned data and define a data type.

Subsequent examples illustrate how to implement services for paging and data management.

You can also use Flash Builder to access services that implement server-side typing. See "Client-side and server-side typing" on page 43.

Examples illustrating server-side typing were not available at the time this document was completed. For server-side typing examples, see Flash Builder server-side type examples.

PHP basic service example

This example shows how to implement a basic service in PHP. The example is based on code that Flash Builder generates when accessing a database table. See "Generating a sample PHP service from a database table" on page 11.

This example illustrates client-side typing. See "Client-side and server-side typing" on page 43.

Important: *Example services are for prototyping only. Use the example service only in a trusted development environment. Before deploying this service, be sure to increase security and restrict access appropriately. For information on writing secure services, see "Deploying applications that access data services" on page 39.*

```
<?php
/**
 * This sample service contains functions that illustrate typical service operations.
 * This code is for prototyping only.
 *
 *  Authenticate users before allowing them to call these methods.
 */
class EmployeeService {

    var $username = "root";
    var $password = "root";
    var $server = "localhost";
    var $port = "3306";
    var $databasename = "employees";
    var $tablename = "employees";

    var $connection;

    /**
     * The constructor initializes the connection to database. Everytime a request is
     * received by Zend AMF, an instance of the service class is created and then the
     * requested method is called.
     */
    public function __construct() {
        $this->connection = mysqli_connect(
                                $this->server,
                                $this->username,
                                $this->password,
                                $this->databasename,
                                $this->port
                            );

        $this->throwExceptionOnError($this->connection);
```

```
}

/**
 * Returns all the rows from the table.
 *
 * Add authroization or any logical checks for secure access to your data
 *
 * @return array
 */
public function getAllEmployees() {

    $stmt = mysqli_prepare($this->connection, "SELECT * FROM $this->tablename");
    $this->throwExceptionOnError();

    mysqli_stmt_execute($stmt);
    $this->throwExceptionOnError();

    $rows = array();

    mysqli_stmt_bind_result($stmt, $row->emp_no, $row->birth_date,
        $row->first_name, $row->last_name, $row->gender, $row->hire_date);

    while (mysqli_stmt_fetch($stmt)) {
      $rows[] = $row;
      $row = new stdClass();
      mysqli_stmt_bind_result($stmt, $row->emp_no, $row->birth_date,
            $row->first_name, $row->last_name, $row->gender, $row->hire_date);
    }

    mysqli_stmt_free_result($stmt);
    mysqli_close($this->connection);

    return $rows;
}

/**
 * Returns the item corresponding to the value specified for the primary key.
 *
 * Add authroization or any logical checks for secure access to your data
 *
 *
 * @return stdClass
 */
public function getEmployeesByID($itemID) {

    $stmt = mysqli_prepare($this->connection, "SELECT * FROM $this->tablename where emp_no=?");
    $this->throwExceptionOnError();

    mysqli_bind_param($stmt, 'i', $itemID);
    $this->throwExceptionOnError();

    mysqli_stmt_execute($stmt);
    $this->throwExceptionOnError();

    mysqli_stmt_bind_result($stmt, $row->emp_no, $row->birth_date,
            $row->first_name, $row->last_name, $row->gender, $row->hire_date);
```

```
        if(mysqli_stmt_fetch($stmt)) {
            return $row;
        } else {
            return null;
        }
    }

    /**
     * Utitity function to throw an exception if an error occurs
     * while running a mysql command.
     */
    private function throwExceptionOnError($link = null) {
        if($link == null) {
            $link = $this->connection;
        }
        if(mysqli_error($link)) {
            $msg = mysqli_errno($link) . ": " . mysqli_error($link);
            throw new Exception('MySQL Error - '. $msg);
        }
    }
}


?>
```

Highlights of EmployeeService:

- Connects to the employees database, which it accesses on port 3306 of localhost. Accesses the employees table in the database.
- Provides class variables for connecting to the service and accessing the tables in the database.

  These variables can be used in functions in the class.

  Replace the values for these variables with values for your system.
- Returns the array of objects to the client application.

  When programming using the Flex framework, services return data only. The client application handles the formatting and presentation of the data.

  This model differs from traditional PHP services, that return data formatted in an HTML template.
- `getEmployeesByID($itemID)` function binds the input parameter to data types.

  The number of variables and length of string types must match the parameters in the statement. The '`?`' in the prepare statement is a placeholder for the parameter.

  mysqli recognizes the following types:

  - integer (`i`)
  - double (`d`)
  - string (`s`)
  - blob (`b`)
- Binds the results, creating an array of objects (`$row[]`).

Flex handles recordsets as an array of objects. Each object represents a record retrieved from the database. Each column of the database record becomes a property of the returned object. The client application can now access the returned data as objects with a set of properties.

Because the server does not define a data type for the returned data, you configure the data type for the returned object. See "Client-side and server-side typing" on page 43.

- Provides a constructor function for initializing the connection to the database.
- Uses mysqli prepare statements for constructing database queries.

  Using prepare statements is a defense against SQL injection statements in calls to the server. Only after preparing the statement is the statement executed on the server.
- Authenticate users before providing access to the functions in this service.

  The sample code does not illustrate how to authenticate users. See the ColdFusion documentation, About User Security. The security principles on user authentication and authorization in this ColdFusion documentation apply to PHP services.
- Throws an exception on error.

  Information that you provide in exceptions is useful when debugging the service implementation. The Flash Builder Test Operation interface displays information returned by exceptions.

  See "Debugging remote services" on page 61 for more information on testing services.
- The filename, `EmployeeService.php`, matches the PHP class name for the service.

  If the filename and the class name do not match, you encounter errors when accessing the service.

More Help topics

"Configuring data types for data service operations" on page 25

"Accessing PHP services" on page 10

"Generating a sample PHP service from a database table" on page 11

PHP example implementing paging

Flash Builder tools allow you to implement paging of data retrieved from a remote service. Paging is the incremental retrieval of large data sets.

Flash Builder requires specific function signatures to implement paging. The following code example provides an example of one way to implement a PHP service for paged data.

This example is based on the code generated by Flash Builder when accessing a database table. See "Generating a sample PHP service from a database table" on page 11.

Important: *Example services are for prototyping only. Use the example service only in a trusted development environment. Before deploying this service, be sure to increase security and restrict access appropriately. For information on writing secure services, see "Deploying applications that access data services" on page 39.*

```
<?php

/**
 *  This sample service contains functions that illustrate typical service operations.
 *  This code is for prototyping only.
 *
 *  Authenticate the user prior to allowing them to call these methods.
 *
 */
class EmployeeServicePaged {

    var $username = "root";
    var $password = "root";
    var $server = "localhost";
    var $port = "3306";
    var $databasename = "employees";
    var $tablename = "employees";

    var $connection;

    /**
     * The constructor initializes the connection to database. Everytime a request is
     * received by Zend AMF, an instance of the service class is created and then the
     * requested method is invoked.
     */
    public function __construct() {
        $this->connection = mysqli_connect(
                                $this->server,
                                $this->username,
                                $this->password,
                                $this->databasename,
                                $this->port
                            );

        $this->throwExceptionOnError($this->connection);
    }


    /**
     * Returns the number of rows in the table.
     *
     * Add authroization or any logical checks for secure access to your data
     *
     *
     */
    public function count() {
        $stmt = mysqli_prepare($this->connection, "SELECT COUNT(*) AS COUNT
                                                    FROM $this->tablename");
        $this->throwExceptionOnError();

        mysqli_stmt_execute($stmt);
        $this->throwExceptionOnError();

        mysqli_stmt_bind_result($stmt, $rec_count);
        $this->throwExceptionOnError();

        mysqli_stmt_fetch($stmt);
```

```
        $this->throwExceptionOnError();

        mysqli_stmt_free_result($stmt);
        mysqli_close($this->connection);

        return $rec_count;
    }


    /**
     * Returns $numItems rows starting from the $startIndex row from the
     * table.
     *
     * Add authroization or any logical checks for secure access to your data
     *
     * @return array
     */
    public function getEmployees_paged($startIndex, $numItems) {

        $stmt = mysqli_prepare($this->connection, "SELECT * FROM
                                    $this->tablename LIMIT ?, ?");
        $this->throwExceptionOnError();

        mysqli_bind_param($stmt, 'ii', $startIndex, $numItems);
        mysqli_stmt_execute($stmt);
        $this->throwExceptionOnError();

        $rows = array();

        mysqli_stmt_bind_result($stmt, $row->emp_no, $row->birth_date,
                                $row->first_name, $row->last_name,
                                $row->gender, $row->hire_date);

        while (mysqli_stmt_fetch($stmt)) {
          $rows[] = $row;
          $row = new stdClass();
          mysqli_stmt_bind_result($stmt, $row->emp_no, $row->birth_date,
                                    $row->first_name, $row->last_name,
                                    $row->gender, $row->hire_date);
        }

        mysqli_stmt_free_result($stmt);
        mysqli_close($this->connection);
```

```
        return $rows;
    }


    /**
     * Utitity function to throw an exception if an error occurs
     * while running a mysql command.
     */
    private function throwExceptionOnError($link = null) {
        if($link == null) {
            $link = $this->connection;
        }
        if(mysqli_error($link)) {
            $msg = mysqli_errno($link) . ": " . mysqli_error($link);
            throw new Exception('MySQL Error - '. $msg);
        }
    }
}
?>
```

The EmployeeServicePaged service returns untyped data. Use the Flash Builder tools to configure the return type for `getEmployees_Paged()`. After configuring the return type, enable paging on the `getEmployees_Paged()` operation.

More Help topics

"Example PHP services" on page 50

"Configuring data types for data service operations" on page 25

"Managing the access of data from the server" on page 29

PHP example implementing data management

Flash Builder tools allow you to implement data management functionality for remote services. Data management is the synchronization of updates to data on a server from the client application.

Flash Builder requires a combination of specific function signatures to implement data management. The following code example provides an example of one way to implement a PHP service for data management.

This example is based on the code generated by Flash Builder when accessing a database table. See "Generating a sample PHP service from a database table" on page 11.

Important: *Example services are for prototyping only. Use the example service only in a trusted development environment. Before deploying this service, be sure to increase security and restrict access appropriately. For information on writing secure services, see "Deploying applications that access data services" on page 39.*

```
<?php

/**
 *  This sample service contains functions that illustrate typical service operations.
 *  This code is for prototyping only.
 *
 *  Authenticate the user prior to allowing them to call these methods.
 */
class EmployeeServiceDM {

    var $username = "root";
    var $password = "root";
    var $server = "localhost";
    var $port = "3306";
    var $databasename = "employees";
    var $tablename = "employees";

    var $connection;

    /**
     * The constructor initializes the connection to database. Everytime a request is
     * received by Zend AMF, an instance of the service class is created and then the
     * requested method is invoked.
     */
    public function __construct() {
        $this->connection = mysqli_connect(
                                $this->server,
                                $this->username,
                                $this->password,
                                $this->databasename,
                                $this->port
                            );

        $this->throwExceptionOnError($this->connection);
    }

    /**
     * Returns all the rows from the table.
     *
     * Add authroization or any logical checks for secure access to your data
     *
     * @return array
     */
    public function getAllEmployees() {

        $stmt = mysqli_prepare($this->connection, "SELECT * FROM $this->tablename");
        $this->throwExceptionOnError();

        mysqli_stmt_execute($stmt);
        $this->throwExceptionOnError();

        $rows = array();

        mysqli_stmt_bind_result($stmt, $row->emp_no, $row->birth_date,
                                $row->first_name, $row->last_name,
                                $row->gender, $row->hire_date);
```

```
        while (mysqli_stmt_fetch($stmt)) {
          $rows[] = $row;
          $row = new stdClass();
          mysqli_stmt_bind_result($stmt, $row->emp_no, $row->birth_date,
                                  $row->first_name, $row->last_name,
                                  $row->gender, $row->hire_date);
        }

        mysqli_stmt_free_result($stmt);
        mysqli_close($this->connection);

        return $rows;
    }

    /**
     * Returns the item corresponding to the value specified for the primary key.
     *
     * Add authroization or any logical checks for secure access to your data
     *
     *
     * @return stdClass
     */
    public function getEmployeesByID($itemID) {

        $stmt = mysqli_prepare($this->connection, "SELECT * FROM
                                        $this->tablename where emp_no=?");
        $this->throwExceptionOnError();

        mysqli_bind_param($stmt, 'i', $itemID);
        $this->throwExceptionOnError();

        mysqli_stmt_execute($stmt);
        $this->throwExceptionOnError();

        mysqli_stmt_bind_result($stmt, $row->emp_no, $row->birth_date,
                                $row->first_name, $row->last_name,
                                $row->gender, $row->hire_date);

        if(mysqli_stmt_fetch($stmt)) {
            return $row;
        } else {
            return null;
        }
    }

    /**
     * Returns the item corresponding to the value specified for the primary key.
     *
     * Add authroization or any logical checks for secure access to your data
     *
     *
     * @return stdClass
     */
    public function createEmployees($item) {

        $stmt = mysqli_prepare($this->connection, "INSERT INTO $this->tablename
                                (emp_no, birth_date, first_name, last_name,
```

```
                            gender, hire_date) VALUES (?, ?, ?, ?, ?, ?)");
    $this->throwExceptionOnError();

    mysqli_bind_param($stmt, 'isssss', $item->emp_no, $item->birth_date
                               $item->first_name, $item->last_name,
                               $item->gender, $item->hire_date);
    $this->throwExceptionOnError();

    mysqli_stmt_execute($stmt);
    $this->throwExceptionOnError();

    $autoid = mysqli_stmt_insert_id($stmt);

    mysqli_stmt_free_result($stmt);
    mysqli_close($this->connection);

    return $autoid;
}

/**
 * Updates the passed item in the table.
 *
 * Add authroization or any logical checks for secure access to your data
 *
 * @param stdClass $item
 * @return void
 */
public function updateEmployees($item) {

    $stmt = mysqli_prepare($this->connection, "UPDATE $this->tablename
                            SET emp_no=?, birth_date=?, first_name=?,
                            last_name=?, gender=?, hire_date=?
                            WHERE emp_no=?");
    $this->throwExceptionOnError();

    mysqli_bind_param($stmt, 'isssssi', $item->emp_no, $item->birth_date,
                        $item->first_name, $item->last_name, $item->gender,
                        $item->hire_date, $item->emp_no);
    $this->throwExceptionOnError();

    mysqli_stmt_execute($stmt);
    $this->throwExceptionOnError();

    mysqli_stmt_free_result($stmt);
    mysqli_close($this->connection);
}

/**
 * Deletes the item corresponding to the passed primary key value from
 * the table.
 *
 * Add authroization or any logical checks for secure access to your data
 *
 *
 * @return void
 */
public function deleteEmployees($itemID) {
```

```
        $stmt = mysqli_prepare($this->connection, "DELETE FROM $this->tablename
                                                    WHERE emp_no = ?");
        $this->throwExceptionOnError();

        mysqli_bind_param($stmt, 'i', $itemID);
        mysqli_stmt_execute($stmt);
        $this->throwExceptionOnError();

        mysqli_stmt_free_result($stmt);
        mysqli_close($this->connection);
    }

    /**
     * Utitity function to throw an exception if an error occurs
     * while running a mysql command.
     */
    private function throwExceptionOnError($link = null) {
        if($link == null) {
            $link = $this->connection;
        }
        if(mysqli_error($link)) {
            $msg = mysqli_errno($link) . ": " . mysqli_error($link);
            throw new Exception('MySQL Error - '. $msg);
        }
    }
}
?>
```

The EmployeeServiceDM service returns untyped data. Use the Flash Builder tools to configure the return type for `getAllEmployeess()` and `getEmployeesByID()`. Use Employee for the custom data type returned by these operations.

After configuring the return type, enable data management on the Employee data type.

More Help topics

"Example PHP services" on page 50

"Configuring data types for data service operations" on page 25

"Managing the access of data from the server" on page 29

Debugging remote services

There are several ways to debug applications that access remote services.

- Flash Builder Test Operation view

  The Flash Builder Test Operation view allows you to call service operations and view the returned data. The Test Operation view displays any error messages displayed by the service.

- Server-side scripts

  For additional debugging of services, you can write scripts that test server code and write output stream information to log files.

- Flash Builder Network Monitor

  Use the Network Monitor after building an application with Flash Builder that accesses a service. Use the Network Monitor to view data sent between the server and client.

Flash Builder Test Operation view

Use the Flash Builder Test Operation view to call operations from a service and view the results of the operation. Results include any error messages returned from the service.

You can use the Test Operation view to view data returned from operations on services you write or services available from HTTP or web services.

Test a service operation

This procedure assumes that you have written a service that you are testing or have access to an HTTP or web service.

1. In the Flash Builder Data/Services view, navigate to a service operation you want to test.
2. From the context menu for the service operation, select Test Operation.
3. (Optional) In the Test Operation view, select Authentication Required to provide login credentials to the service.
4. If the operation takes parameters, click the Enter Value field to provide a value for the parameter.

   If the parameter requires a complex type, click the Ellipsis button in the Enter Value field to open an editor that accepts JSON notation. Provide the value for the parameter using JSON notation.
5. Click Test to view the results of the operation.

Scripts to test server code

Use test scripts to view and debug server code before attempting to connect to the server in Flash Builder. Test scripts provide the following benefits:

- You can view test results from a web browser.

  As you modify the code, simply refresh the browser page to view the results.
- You can echo or print results to the output stream, which you cannot do directly from AMF.
- Error displays are nicely formatted and typically more complete than errors captured using AMF.

ColdFusion Scripts

Use the following script, `tester.cfm`, to dump a call to a function.

```
<!--- tester.cfm --->
<cfobject component="EmployeeService" name="o"/>
<cfdump var="#o.getAllItems()#">
```

In `tester2.cfm`, you specify the method and arguments to call in the URL.

```
<!--- tester2.cfm --->
<cfdump var="#url#">

<cfinvoke component="#url.cfc#" method="#url.method#" argumentCollection="#url#"
returnVariable="r">

<p>Result:

<cfif isDefined("r")>
    <cfdump var="#r#">
<cfelse>
    (no result)
</cfif>
```

For example, call the `getItemID()` method in EmployeeService with the following URL:

```
http://localhost/tester2.cfm?EmployeeService&method=getItemId&id=12
```

`tester3.cfm` writes a log that records calls to operations and dumps the input arguments using `cfdump`.

```
<!--- tester3.cfm --->
<cfsavecontent variable="d"><cfdump var="#arguments#"></cfsavecontent>

<cffile action="append"
file="#getDirectoryFromPath(getCurrentTemplatePath())#MyServiceLog.htm"
output="<p>#now()# operationName #d#">
```

PHP Scripts

Use the following script, `tester.php`, to dump a call to a function.

```
<pre>
<?php
include('MyService.php');
    $o = new MyService();
    var_dump($o->getAllItems());
?>
</pre>
```

Add the following code to your PHP service to log messages during code execution.

```
$message = 'updateItem: '.$item["id"];
$log_file = '/Users/me/Desktop/myservice.log';
error_log(date('d/m/Y H:i:s').' '.$message.PHP_EOL, 3, $log_file);
```

Add the following code to your PHP service to enable dumping to a log file:

```
ob_start();
var_dump($item);
$result = ob_get_contents();
ob_end_clean();

$message = 'updateItem: '.$result;
$log_file = '/Users/me/Desktop/myservice.log';
error_log(date('d/m/Y H:i:s').' '.$message.PHP_EOL, 3, $log_file);
```

Network Monitor

The Network Monitor is available in Flash Builder from the Flex Debugging Perspective. Enable the monitor before you can use it to monitor data. See Monitor applications that access data services for details about enabling and using the Network Monitor.

Example implementing services from multiple sources

Typically applications access data from different sources, displaying the results of the data association in an application. This example shows how to associate data from the following three tables in an employees database:

- Departments

  Each record contains the following fields: department number and department name.

- Dept_emp

  Each record contains the following fields: emp_no, dept_no, from_date, to_date

- Employees

  Each record contains the following fields: emp_no, birth_date, first_name, last_name, gender, hire_date

The sample application has two DataGrids, one for Departments and one for Employees.

The Departments lists all the departments. When you select a department, the Employees DataGrid lists all the employees in that department.

When you select an employee in the Employees DataGrid, a form is populated, allowing you to update the employee record.

Create the services

For this example, create a single service. The service contains the following operations:

- getAllDepartments()
- getEmployeesByDept()
- `getEmployeeByID()`
- `updateEmployee()`

EmployeeService (PHP)

`EmployeeService.php` implements a service that contains a single function. `GetEmployeesByID()` accepts the department ID as an argument, returning all the employees in the given department. The function also returns the dates an employee started and left the department. `GetEmployeesByDept()` executes the following SQL query:

```
SELECT
    employees.emp_no,
    employees.birth_date,
    employees.first_name,
    employees.last_name,
    employees.gender,
    employees.hire_date,
    dept_emp.from_date,
    dept_emp.to_date
FROM employees, dept_emp
WHERE dept_emp.emp_no = employees.emp_no and
    dept_emp.dept_no = departments.dept_no
```

Important: *Example services are for prototyping only. Use the example service only in a trusted development environment. Before deploying this service, be sure to increase security and restrict access appropriately. For information on writing secure services, see* *"Deploying applications that access data services" on page 39.*

```
<?php

/**
 * EmployeeService.php
 *
 * This sample service contains functions that illustrate typical service operations.
 * Use these functions as a starting point for creating your own service implementation.
 *
 *  This code is for prototyping only.
 *
 *  Authenticate the user before allowing them to call these methods.
 */
class EmployeeService {

    var $username = "admin2";
    var $password = "Cosmo49";
    var $server = "localhost";
    var $port = "3306";
    var $databasename = "employees";
    var $tablename = "employees";

    var $connection;

    /**
     * The constructor initializes the connection to database. Everytime a request is
     * received by Zend AMF, an instance of the service class is created and then the
     * requested method is called.
     */
    public function __construct() {
        $this->connection = mysqli_connect(
                                $this->server,
                                $this->username,
                                $this->password,
                                $this->databasename,
                                $this->port
                            );

        $this->throwExceptionOnError($this->connection);
    }
```

```
/**
 * Returns all the rows from the table.
 *
 * Add authroization or any logical checks for secure access to your data
 *
 * @return array
 */
public function getAllDepartments() {

    $stmt = mysqli_prepare($this->connection, "SELECT * FROM departments");
    $this->throwExceptionOnError();

    mysqli_stmt_execute($stmt);
    $this->throwExceptionOnError();

    $rows = array();

    mysqli_stmt_bind_result($stmt, $row->dept_no, $row->dept_name);

    while (mysqli_stmt_fetch($stmt)) {
      $rows[] = $row;
      $row = new stdClass();
      mysqli_stmt_bind_result($stmt, $row->dept_no, $row->dept_name);
    }

    mysqli_stmt_free_result($stmt);
    mysqli_close($this->connection);

    return $rows;
}

public function getEmployeesByDept($deptId) {
    $stmt = mysqli_prepare($this->connection, "select employees.emp_no,
                                            employees.first_name,
                                            employees.last_name,
                                            employees.gender,
                                            dept_emp.dept_no
                                    from employees, dept_emp
                            where dept_emp.emp_no = employees.emp_no
                                and dept_emp.dept_no = ?
                            limit 0,30;");
    $this->throwExceptionOnError();

    mysqli_bind_param($stmt, 's', $deptId);
    $this->throwExceptionOnError();

    mysqli_stmt_execute($stmt);
    $this->throwExceptionOnError();

    $rows = array();

    mysqli_stmt_bind_result($stmt, $row->emp_no, $row->first_name,
                        $row->last_name, $row->gender, $row->dept_no);

    while (mysqli_stmt_fetch($stmt)) {
      $rows[] = $row;
```

```
        $row = new stdClass();

        mysqli_stmt_bind_result($stmt, $row->emp_no, $row->first_name,
                          $row->last_name, $row->gender, $row->dept_no);
    }

    mysqli_stmt_free_result($stmt);
    mysqli_close($this->connection);

    return $rows;
}

/**
 * Returns the item corresponding to the value specified for the primary key.
 *
 * Add authroization or any logical checks for secure access to your data
 *
 *
 * @return stdClass
 */
public function getEmployeesByID($itemID) {

    $stmt = mysqli_prepare($this->connection, "SELECT * FROM employees
                                                    where emp_no=?");
    $this->throwExceptionOnError();

    mysqli_bind_param($stmt, 'i', $itemID);
    $this->throwExceptionOnError();

    mysqli_stmt_execute($stmt);
    $this->throwExceptionOnError();

    mysqli_stmt_bind_result($stmt, $row->emp_no, $row->birth_date,
                            $row->first_name, $row->last_name,
                            $row->gender, $row->hire_date);

    if(mysqli_stmt_fetch($stmt)) {
        return $row;
    } else {
        return null;
    }
}


/**
 * Updates the passed item in the table.
 *
 * Add authroization or any logical checks for secure access to your data
 *
 * @param stdClass $item
 * @return void
 */
public function updateEmployees($item) {

    $stmt = mysqli_prepare($this->connection, "UPDATE employees
                                SET emp_no=?, birth_date=?, first_name=?,
                                    last_name=?, gender=?, hire_date=?
```

```
                                    WHERE emp_no=?");
        $this->throwExceptionOnError();

        mysqli_bind_param($stmt, 'issssi', $item->emp_no, $item->birth_date,
                          $item->first_name, $item->last_name, $item->gender,
                          $item->hire_date, $item->emp_no);
        $this->throwExceptionOnError();

        mysqli_stmt_execute($stmt);
        $this->throwExceptionOnError();

        mysqli_stmt_free_result($stmt);
        mysqli_close($this->connection);
    }


    /**
     * Utitity function to throw an exception if an error occurs
     * while running a mysql command.
     */
    private function throwExceptionOnError($link = null) {
        if($link == null) {
            $link = $this->connection;
        }
        if(mysqli_error($link)) {
            $msg = mysqli_errno($link) . ": " . mysqli_error($link);
            throw new Exception('MySQL Error - '. $msg);
        }
    }
}
>?>
```

EmployeeService (ColdFusion)

`EmployeeService.cfc` implements a service that contains a single function. `GetEmployeesByID()` accepts the department ID as an argument, returning all the employees in the given department. The function also returns the dates an employee started and left the department. `GetEmployeesByDept()` executes the following SQL query:

```
SELECT
    employees.emp_no,
    employees.birth_date,
    employees.first_name,
    employees.last_name,
    employees.gender,
    employees.hire_date,
    dept_emp.from_date,
    dept_emp.to_date
FROM employees, dept_emp
WHERE dept_emp.emp_no = employees.emp_no and dept_emp.dept_no = departments.dept_no
```

Important: *Example services are for prototyping only. Use the example service only in a trusted development environment. Before deploying this service, be sure to increase security and restrict access appropriately. For information on writing secure ColdFusion services, see the ColdFusion documentation About User Security.*

```
<cfcomponent output="false">

<!---
    This sample service contains functions that illustrate typical service operations.
    Use these functions as a starting point for creating your own service implementation.

    This code is for prototyping only.

    Authenticate the user before allowing them to call these methods. You can find more
    information at http://www.adobe.com/go/cf9_usersecurity
--->
    <cffunction name="getEmployeesByDept" output="false" access="remote"  returntype="any" >
        <cfargument name="dept_no" type="string" required="true" />

        <cfset var qItem="">
        <cfquery name="qItem" datasource="employees">
        SELECT employees.emp_no,
            employees.birth_date,
            employees.first_name,
            employees.last_name,
            employees.gender,
            employees.hire_date,
            dept_emp.from_date,
            dept_emp.to_date
        FROM employees, dept_emp
        WHERE dept_emp.emp_no = employees.emp_no and
        dept_emp.dept_no = <CFQUERYPARAM CFSQLTYPE="CF_SQL_VARCHAR"
VALUE="#ARGUMENTS.dept_no#">
        </cfquery>

        <cfreturn qItem>

    </cffunction>

</cfcomponent>?>
```

Import the services into a server project.

1 In Flash Builder, create a Flex Project that is named Associations:

(PHP) When creating the project, specify PHP for the Application Server Type.

(PHP) After creating the project, Flash Builder creates an output folder in the folder that is the web root of your PHP configuration. The default name for the PHP_Associations project is `PHP_Associations-debug`.

(ColdFusion) When creating the project, specify ColdFusion for the Application Server Type. Then specify ColdFusion Flash Remoting.

2 (PHP) Within `PHP_Associations-debug`, create a folder named `services`. Copy `EmployeeService.php` into the `services` folder.

3 (ColdFusion) Create a folder named `Associations` in the web root of your ColdFusion configuration. Copy `EmployeeService.chc` into the `Associations` folder.

4 Import the EmployeeService into the project:

Make sure PHP_Associations is the active project in Flash Builder.

Select Data > Connect to PHP. To specify the PHP Class, browse to the `services` folder and select `EmployeeService.php`. Click Finish.

For more information, see "Connecting to PHP data services" on page 10.

5 Configure the return type for the operations in EmployeeService.

- DepartmentService

  From the context menu for `getAllDepartments()`, select Configure Return Type.

  Click Next to auto-detect the return type.

  Specify **Department** for the custom return type. Click Finish.

- EmployeeService

  For `getEmployeesByDept()`, select Configure Return Type.

  Click Next to auto-detect the return type.

  For the value of the parameter, specify **d007**. Click Next.

  Specify **Employee** for the custom return type. Click Finish.

For more information, see "Configuring data types for data service operations" on page 25.

Chapter 4: Accessing server-side data

Adobe® Flex® data access components use remote procedure calls to interact with server environments, such as PHP, Adobe ColdFusion, and Microsoft ASP.NET. These components provide data to client applications built with the Adobe Flex framework, and send data to back-end data sources. For an introduction to data access components, see "Data access components" on page 4.

Using HTTPService components

You can use an HTTPService component with any kind of server-side technology, including PHP pages, ColdFusion Pages, Javaserver Pages (JSPs), Java servlets, Ruby on Rails, and Microsoft ASP pages. Additionally, you use HTTPService to access REST-based web services.

For API reference information about the HTTPService component, see mx.rpc.http.mxml.HTTPService.

Working with PHP and SQL data

You can use an HTTPService component with PHP and a SQL database management system to display the results of a database query in an application. You can also use the component to insert, update, and delete data in a database. You can call a PHP page with GET or POST to perform a database query. You can then format the query result data in an XML structure and return the XML structure to the application in the HTTP response. When the result has been returned to the application, you can display it in one or more user interface controls.

MXML code

The application in the following example calls a PHP page with the POST method. The PHP page queries a MySQL database table called users. It formats the query results as XML and returns the XML to the application, where it is bound to the `dataProvider` property of a DataGrid control and displayed in the DataGrid control. The application also sends the user name and email address of new users to the PHP page, which performs an insert into the user database table.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:mx="library://ns.adobe.com/flex/mx" minWidth="955" minHeight="600"
        creationComplete="send_data()">
    <fx:Declarations>
        <s:HTTPService id="userRequest" url="http://myserver/myproj/request_post2.php"
                                                    useProxy="false" method="POST">
            <mx:request xmlns="">
                <username>{username.text}</username>
                <emailaddress>{emailaddress.text}</emailaddress>
            </mx:request>
        </s:HTTPService>
    </fx:Declarations>
    <fx:Script>
        <![CDATA[
        private function send_data():void {
        userRequest.send();
        }
        ]]>
    </fx:Script>
    <mx:Form x="20" y="10" width="300">
        <mx:FormItem>
            <s:Label text="Username" />
            <s:TextInput id="username"/>
        </mx:FormItem>
        <mx:FormItem>
            <s:Label text="Email Address" />
            <s:TextInput id="emailaddress"/>
        </mx:FormItem>
        <s:Button label="Submit" click="send_data()"/>
    </mx:Form>
    <mx:DataGrid id="dgUserRequest" x="20" y="160"
        dataProvider="{userRequest.lastResult.users.user}">
        <mx:columns>
            <mx:DataGridColumn headerText="User ID" dataField="userid"/>
            <mx:DataGridColumn headerText="User Name" dataField="username"/>
        </mx:columns>
    </mx:DataGrid>
    <s:TextInput x="20" y="340" id="selectedemailaddress"
text="{dgUserRequest.selectedItem.emailaddress}"/>

</s:Application>
```

The HTTPService's `send()` method makes the call to the PHP page. This call is made in the `send_data()` method in the Script block of the MXML file.

The `resultFormat` property of the HTTPService component is set to `object`, so the data is sent back to the application as a graph of ActionScript objects. This is the default value for the `resultFormat` property. Alternatively, you can use a `resultFormat` of `e4x` to return data as an XMLList object on which you can perform ECMAScript for XML (E4X) operations. Switching the `resultFormat` property to `e4x` requires the following minor changes to the MXML code.

Note: *If the result format is `e4x`, you do not include the root node of the XML structure in the dot notation when binding to the DataGrid.*

The XML returned in this example contains no namespace information. For information about working with XML that does contain namespaces, see "Handling results as XML with the e4x result format" on page 123.

```
...
<s:HTTPService id="userRequest" url="http://myserver/myproj/request_post2.php"
                            useProxy="false" method="POST" resultFormat="e4x">
...
<mx:DataGrid id="dgUserRequest" x="22" y="150"
    dataProvider="{userRequest.lastResult.user}">
...
```

When using the `e4x` result format, you can optionally bind the `lastResult` property to an XMLListCollection object and then bind that object to the `DataGrid.dataProvider` property, as the following code snippet shows:

```
<fx:Declarations>
...
    <mx:XMLListCollection id="xc"
         source="{userRequest.lastResult.user}"/>
...
</fx:Declarations>
...
    <mx:DataGrid id="dgUserRequest" x="22" y="128" dataProvider="{xc}">
...
```

MySQL database script

The PHP code for this application uses a database table called users in a MySQL database called sample. The following MySQL script creates the table:

```
CREATE TABLE `users` (
`userid` int(10) unsigned NOT NULL auto_increment,
`username` varchar(255) collate latin1_general_ci NOT NULL,
`emailaddress` varchar(255) collate latin1_general_ci NOT NULL,
PRIMARY KEY (`userid`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 COLLATE=latin1_general_ci AUTO_INCREMENT=3 ;
```

PHP code

This application calls the following PHP page. This PHP code performs SQL database inserts and queries, and returns query results to the application in an XML structure.

```
<?php
define( "DATABASE_SERVER", "servername" );
define( "DATABASE_USERNAME", "username" );
define( "DATABASE_PASSWORD", "password" );
define( "DATABASE_NAME", "sample" );

//connect to the database.
$mysql = mysql_connect(DATABASE_SERVER, DATABASE_USERNAME, DATABASE_PASSWORD);

mysql_select_db( DATABASE_NAME );

// Quote variable to make safe
function quote_smart($value)
{
    // Stripslashes
    if (get_magic_quotes_gpc()) {
    $value = stripslashes($value);
    }
    // Quote if not integer
    if (!is_numeric($value)) {
    $value = "'" . mysql_real_escape_string($value) . "'";
    }
    return $value;
}

if( $_POST["emailaddress"] AND $_POST["username"])
{
//add the user
$Query = sprintf("INSERT INTO users VALUES ('', %s, %s)",
        quote_smart($_POST['username']), quote_smart($_POST['emailaddress']));

$Result = mysql_query( $Query );
}

//return a list of all the users
$Query = "SELECT * from users";
$Result = mysql_query( $Query );

$Return = "<users>";

while ( $User = mysql_fetch_object( $Result ) )
{
$Return .= "<user><userid>".$User->userid."</userid><username>".
            $User->username."</username><emailaddress>".
            $User->emailaddress."</emailaddress></user>";
}
$Return .= "</users>";
mysql_free_result( $Result );
print ($Return)
?>
```

Working with ColdFusion and SQL data

You can use an HTTPService component with a ColdFusion page and a SQL database management system to display the results of a database query in an application. You can also use the component to insert, update, and delete data in a database. You call a ColdFusion page with GET or POST to perform a database query. You then format the query result data in an XML structure and return the XML structure to the application in the HTTP response. When the result has been returned to the application, you display it in one or more user interface controls.

MXML code

The application in the following example calls a ColdFusion page with the POST method. The ColdFusion page queries a MySQL database table called users. It formats the query results as XML and returns the XML to the application, where it is bound to the `dataProvider` property of a DataGrid control and displayed in the DataGrid control. The application also sends the user name and email address of new users to the ColdFusion page, which performs an insert into the user database table.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:mx="library://ns.adobe.com/flex/mx" minWidth="955" minHeight="600"
        creationComplete="userRequest.send()">

    <fx:Declarations>
    <s:HTTPService id="userRequest" url="http://server:8500/flexapp/returncfxml.cfm"
        useProxy="false" method="POST">
        <mx:request xmlns="">
            <username>{username.text}</username>
            <emailaddress>{emailaddress.text}</emailaddress>
        </mx:request>
    </s:HTTPService>
    </fx:Declarations>
    <mx:Form x="22" y="10" width="300">
        <mx:FormItem>
            <s:Label text="Username" />
            <s:TextInput id="username"/>
        </mx:FormItem>
        <mx:FormItem>
            <s:Label text="Email Address" />
            <s:TextInput id="emailaddress"/>
        </mx:FormItem>
        <s:Button label="Submit" click="userRequest.send()"/>
    </mx:Form>
    <mx:DataGrid id="dgUserRequest" x="22" y="128"
        dataProvider="{userRequest.lastResult.users.user}">
        <mx:columns>
            <mx:DataGridColumn headerText="User ID" dataField="userid"/>
            <mx:DataGridColumn headerText="User Name" dataField="username"/>
        </mx:columns>
    </mx:DataGrid>
    <s:TextInput x="22" y="300" id="selectedemailaddress"
text="{dgUserRequest.selectedItem.emailaddress}"/>
</s:Application>
```

The HTTPService's `send()` method makes the call to the ColdFusion page. This call is made in the `send_data()` method in the Script block of the MXML file.

The `resultFormat` property of the HTTPService component is set to `object`, so the data is sent back to the application as a graph of ActionScript objects. This is the default value for the `resultFormat` property. Alternatively, you can use a result format of `e4x` to return data as an XMLList object on which you can perform ECMAScript for XML (E4X) operations. Switching the `resultFormat` property to `e4x` requires the following minor changes to the MXML code.

Note: *If the result format is `e4x`, you do not include the root node of the XML structure in the dot notation when binding to the DataGrid.*

The XML returned in this example contains no namespace information. For information about working with XML that does contain namespaces, see "Handling results as XML with the e4x result format" on page 123.

```
...
<s:HTTPService id="userRequest" url="http://myserver:8500/flexapp/returncfxml.cfm"
                        useProxy="false" method="POST" resultFormat="e4x">
...
<mx:DataGrid id="dgUserRequest" x="22" y="128"
    dataProvider="{userRequest.lastResult.user}">
...
```

When using the `e4x` result format, you can optionally bind the `lastResult` property to an XMLListCollection object and then bind that object to the DataGrid `dataProvider` property, as the following code snippet shows:

```
<fx:Declarations>
...
    <mx:XMLListCollection id="xc"
         source="{userRequest.lastResult.user}"/>
...
</fx:Declarations>
...
    <mx:DataGrid id="dgUserRequest" x="22" y="128" dataProvider="{xc}">
...
```

SQL script

The ColdFusion code for this application uses a database table called users in a MySQL database called sample. The following MySQL script creates the table:

```
CREATE TABLE `users` (
`userid` int(10) unsigned NOT NULL auto_increment,
`username` varchar(255) collate latin1_general_ci NOT NULL,
`emailaddress` varchar(255) collate latin1_general_ci NOT NULL,
PRIMARY KEY (`userid`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 COLLATE=latin1_general_ci AUTO_INCREMENT=3 ;
```

ColdFusion code

The application that is listed in the Working with ColdFusion and SQL data section calls the following ColdFusion application, `returncfxml.cfm`. This ColdFusion code performs SQL database inserts and queries, and returns query results to the application. The ColdFusion page uses the `cfquery` tag to insert data into the database and query the database, and it uses the `cfxml` tag to format the query results in an XML structure.

```
<!--- returncfxml.cfm --->

<cfprocessingdirective pageencoding = "utf-8" suppressWhiteSpace = "Yes">
<cfif isDefined("username") and isDefined("emailaddress") and username NEQ "">
    <cfquery name="addempinfo" datasource="sample">
    INSERT INTO users (username, emailaddress) VALUES (
    <cfqueryparam value="#username#" cfsqltype="CF_SQL_VARCHAR" maxlength="255">,
        <cfqueryparam value="#emailaddress#" cfsqltype="CF_SQL_VARCHAR" maxlength="255"> )
</cfquery>
</cfif>
<cfquery name="alluserinfo" datasource="sample">
    SELECT userid, username, emailaddress FROM users
</cfquery>
<cfxml variable="userXML">
    <users>
        <cfloop query="alluserinfo">
            <cfoutput>
                <user>
                    <userid>#toString(userid)#</userid>
                    <username>#username#</username>
                    <emailaddress>#emailaddress#</emailaddress>
                </user>
        </cfoutput>
        </cfloop>
    </users>
</cfxml>
<cfoutput>#userXML#</cfoutput>
</cfprocessingdirective>
```

Working with Javaserver Pages

You can use a Flex HTTPService component with a JSP page and a SQL database management system to display the results of a database query in an application. You can also use the component to insert, update, and delete data in a database. You call a JSP page with GET or POST to perform a database query. You then format the query result data in an XML structure and return the XML structure to the application in the HTTP response. When the result has been returned to the application, you display it in one or more user interface controls.

MXML code

The application in the following example calls a JSP page that retrieves data from a SQL database. It formats database query results as XML and returns the XML to the application, where it is bound to the `dataProvider` property of a DataGrid control and displayed in the DataGrid control.

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:mx="library://ns.adobe.com/flex/mx" minWidth="955" minHeight="600">

    <fx:Declarations>
        <s:HTTPService id="srv" url="catalog.jsp"/>
    </fx:Declarations>

    <mx:DataGrid dataProvider="{srv.lastResult.catalog.product}"
                                    width="100%" height="100%"/>

    <s:Button label="Get Data" click="srv.send()"/>

</mx:Application>
```

The HTTPService's `send()` method makes the call to the JSP page. This call is made in the `click` event of the Button in the MXML file.

The `resultFormat` property of the HTTPService component is set to `object`, so the data is sent back to the application as a graph of ActionScript objects. This is the default value for the `resultFormat` property. Alternatively, you can use a result format of `e4x` to return data as an XMLList object on which you can perform ECMAScript for XML (E4X) operations. Switching the `resultFormat` property to `e4x` requires the following minor changes to the MXML code.

***Note:** If the result format is `e4x`, you do not include the root node of the XML structure in the dot notation when binding to the DataGrid.*

The XML returned in this example contains no namespace information. For information about working with XML that does contain namespaces, see "Handling results as XML with the e4x result format" on page 123.

```
...
    <s:HTTPService id="srv" url="catalog.jsp" resultFormat="e4x"/>
...
    <mx:DataGrid dataProvider="{srv.lastResult.product}" width="100%" height="100%"/>
```

When using the e4x result format, you can optionally bind the `lastResult` property to an XMLListCollection object and then bind that object to the DataGrid.dataProvider property:

```
<fx:Declarations>
...
    <mx:XMLListCollection id="xc"
         source="{userRequest.lastResult.user}"/>
...
</fx:Declarations>
...
    <mx:DataGrid id="dgUserRequest" x="22" y="128" dataProvider="{xc}">
...
```

JSP code

The following example shows the JSP page used in this application. This JSP page does not call a database directly. It gets its data from a Java class called ProductService, which in turn uses a Java class called Product to represent individual products.

```
<%@page import="flex.samples.product.ProductService,
                flex.samples.product.Product,
                java.util.List"%>
<?xml version="1.0" encoding="utf-8"?>
<catalog>
<%
    ProductService srv = new ProductService();
    List list = null;
    list = srv.getProducts();
    Product product;
    for (int i=0; i<list.size(); i++)
    {
        product = (Product) list.get(i);
%>
<product productId="<%= product.getProductId()%>">
<name><%= product.getName() %></name>
<description><%= product.getDescription() %></description>
<price><%= product.getPrice() %></price>
<image><%= product.getImage() %></image>
<category><%= product.getCategory() %></category>
<qtyInStock><%= product.getQtyInStock() %></qtyInStock>
</product>
<%
    }
%>
</catalog>
```

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.url = "catalog.jsp";
                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>
```

Using WebService components

Applications created with the Flex framework can interact with SOAP-based 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 Simple Object Access Protocol (SOAP) 1.1, XML Schema 1.0 (versions 1999, 2000, and 2001), and WSDL 1.1 RPC-encoded, RPC-literal, and document-literal (bare and wrapped style parameters). The two most common types of web services use remote procedure call (RPC) encoded or document-literal SOAP bindings; the terms *encoded* and *literal* indicate the type of WSDL-to-SOAP mapping that a service uses.

Flex supports web service requests and results that are formatted as 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 an application built with Flex, and a web service.

Adobe® Flash® Player operates within a security sandbox that limits what applications built with Flex and other applications built with Flash can access over HTTP. Applications built with Flash are allowed HTTP access only 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 proxy service, available in LiveCycle Data Services and BlazeDS, intercepts requests to remote web services, redirects the requests, and then returns the responses to the client.

If you are not using LiveCycle Data Services or BlazeDS, you can access web services in the same domain as your application; or a crossdomain.xml (cross-domain policy) file that allows access from your application's domain must be installed on the web server hosting the RPC service.

For API reference information about the WebService component, see mx.rpc.soap.mxml.WebService.

Sample WebService application

The following sample code is for an application that uses a WebService component to call web service operations.

MXML code

The application in the following example calls a web service that queries a SQL database table called users and returns data to the application, where it is bound to the `dataProvider` property of a DataGrid control and displayed in the DataGrid control. The application also sends the user name and e-mail address of new users to the web service, which performs an insert into the user database table. The back-end implementation of the web service is a ColdFusion component; the same ColdFusion component is accessed as a remote object in "Using RemoteObject components" on page 97.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:mx="library://ns.adobe.com/flex/mx" minWidth="955" minHeight="600">
    <fx:Declarations>
        <s:WebService
            id="userRequest"
            wsdl="http://localhost:8500/flexapp/returnusers.cfc?wsdl">

            <mx:operation name="returnRecords" resultFormat="object"
                          fault="mx.controls.Alert.show(event.fault.faultString)"
                          result="remotingCFCHandler(event)"/>

            <mx:operation name="insertRecord" result="insertCFCHandler()"
                          fault="mx.controls.Alert.show(event.fault.faultString)"/>
        </s:WebService>
    </fx:Declarations>
    <fx:Script>
        <![CDATA[
            import mx.rpc.events.ResultEvent;

            private function remotingCFCHandler(e:ResultEvent):void
            {
                dgUserRequest.dataProvider = e.result;
            }

            private function insertCFCHandler():void
            {
                userRequest.returnRecords();
            }
            private function clickHandler():void
```

```
            {
                userRequest.insertRecord(username.text, emailaddress.text);
            }
        ]]>
    </fx:Script>

    <mx:Form x="22" y="10" width="300">
        <mx:FormItem>
            <s:Label text="Username" />
            <s:TextInput id="username"/>
        </mx:FormItem>
        <mx:FormItem>
            <s:Label text="Email Address" />
            <s:TextInput id="emailaddress"/>
        </mx:FormItem>
        <s:Button label="Submit" click="clickHandler()"/>
    </mx:Form>

    <mx:DataGrid id="dgUserRequest" x="22" y="160">
        <mx:columns>
            <mx:DataGridColumn headerText="User ID" dataField="USERID"/>
            <mx:DataGridColumn headerText="User Name" dataField="USERNAME"/>
        </mx:columns>
    </mx:DataGrid>

    <s:TextInput x="22" y="320" id="selectedemailaddress"
text="{dgUserRequest.selectedItem.emailaddress}"/>
</s:Application>
```

WSDL document

The following example shows the WSDL document that defines the API of the web service:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://flexapp"
    xmlns:apachesoap="http://xml.apache.org/xml-soap"
    xmlns:impl="http://flexapp" xmlns:intf="http://flexapp"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:tns1="http://rpc.xml.coldfusion"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by ColdFusion version 8,0,0,171651-->
    <wsdl:types>
<schema targetNamespace="http://rpc.xml.coldfusion"
xmlns="http://www.w3.org/2001/XMLSchema">
    <import namespace="http://flexapp"/>
    <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
    <complexType name="CFCInvocationException">
<sequence/>
    </complexType>

    <complexType name="QueryBean">
<sequence>
    <element name="columnList" nillable="true" type="impl:ArrayOf_xsd_string"/>
    <element name="data" nillable="true" type="impl:ArrayOfArrayOf_xsd_anyType"/>
</sequence>
```

```
    </complexType>
</schema>
<schema targetNamespace="http://flexapp" xmlns="http://www.w3.org/2001/XMLSchema">
    <import namespace="http://rpc.xml.coldfusion"/>

    <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
    <complexType name="ArrayOf_xsd_string">
<complexContent>
    <restriction base="soapenc:Array">
<attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:string[]"/>
    </restriction>
</complexContent>
    </complexType>
    <complexType name="ArrayOfArrayOf_xsd_anyType">

<complexContent>
    <restriction base="soapenc:Array">
<attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:anyType[][]"/>
    </restriction>
</complexContent>
    </complexType>
</schema>
    </wsdl:types>

    <wsdl:message name="CFCInvocationException">

<wsdl:part name="fault" type="tns1:CFCInvocationException"/>
</wsdl:message>
<wsdl:message name="returnRecordsRequest">
</wsdl:message>
<wsdl:message name="insertRecordResponse">
</wsdl:message>
<wsdl:message name="returnRecordsResponse">
<wsdl:part name="returnRecordsReturn" type="tns1:QueryBean"/>
</wsdl:message>
<wsdl:message name="insertRecordRequest">
<wsdl:part name="username" type="xsd:string"/>
<wsdl:part name="emailaddress" type="xsd:string"/>
</wsdl:message>
<wsdl:portType name="returncfxml">
<wsdl:operation name="insertRecord" parameterOrder="username emailaddress">
<wsdl:input message="impl:insertRecordRequest" name="insertRecordRequest"/>
<wsdl:output message="impl:insertRecordResponse" name="insertRecordResponse"/>
<wsdl:fault message="impl:CFCInvocationException" name="CFCInvocationException"/>
</wsdl:operation>
<wsdl:operation name="returnRecords">
<wsdl:input message="impl:returnRecordsRequest" name="returnRecordsRequest"/>
<wsdl:output message="impl:returnRecordsResponse" name="returnRecordsResponse"/>
<wsdl:fault message="impl:CFCInvocationException" name="CFCInvocationException"/>
</wsdl:operation>
</wsdl:portType>
<wsdl:binding name="returncfxml.cfcSoapBinding" type="impl:returncfxml">
<wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
<wsdl:operation name="insertRecord">
<wsdlsoap:operation soapAction=""/>
<wsdl:input name="insertRecordRequest">
<wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
```

```
namespace="http://flexapp" use="encoded"/>
</wsdl:input>
<wsdl:output name="insertRecordResponse">
<wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://flexapp" use="encoded"/>
</wsdl:output>
<wsdl:fault name="CFCInvocationException">
<wsdlsoap:fault encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
name="CFCInvocationException" namespace="http://flexapp" use="encoded"/>
</wsdl:fault>
</wsdl:operation>
<wsdl:operation name="returnRecords">
<wsdlsoap:operation soapAction=""/>
<wsdl:input name="returnRecordsRequest">
<wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://flexapp" use="encoded"/>
</wsdl:input>
<wsdl:output name="returnRecordsResponse">
<wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://flexapp" use="encoded"/>
</wsdl:output>
<wsdl:fault name="CFCInvocationException">
<wsdlsoap:fault encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
name="CFCInvocationException" namespace="http://flexapp" use="encoded"/>
</wsdl:fault>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="returncfxmlService">
<wsdl:port binding="impl:returncfxml.cfcSoapBinding" name="returncfxml.cfc">
<wsdlsoap:address location="http://localhost:8500/flexapp/returnusers.cfc"/>
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

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 `echoArgs()` method of the service.

Note: *When you declare a WebService component in ActionScript, call the `WebService.loadWSDL()` method.*

```
<?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.wsdl="http://myserver:8500/flexapp/app1.cfc?wsdl";
            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>
```

Reserved Operation names

WebService operations are accessible by simply naming them after a service variable. However, naming conflicts can occur if an operation name happens to match a defined method on the service. You can use the following method in ActionScript on a WebService component to return the operation of the given name:

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

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 applications built with Flex, 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 an application built with Flex, sends to a web service.
`<message>`	Defines the data that a WebService 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 an application built with Flex.
`<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 RPC-oriented or document-oriented (document-literal) operations. Flex supports both operation styles.

When calling an RPC-oriented operation, an application sends a SOAP message that specifies an operation and its parameters. When calling a document-oriented operation, an 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 feature acts as an intermediary between applications and web services. It adds an endpoint's identity to whatever the endpoint passes to an application. If the endpoint sends session information, the application receives it. This feature requires no configuration, and 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.

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, use the following constructor:

```
SOAPHeader(qname:QName, content:Object)
```

To create the QName object in the first parameter of the `SOAPHeader()` method, 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" wsdl="http://myserver:8500/flexapp/app1.cfc?wsdl"
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 the 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" wsdl="http://myserver:8500/flexapp/app1.cfc?wsdl"
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 that you want to use a web service that requires you to pass security credentials. When 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 86.

Serializing web service data

Encoding ActionScript data

The following table shows the encoding mappings from ActionScript 3.0 types to XML schema complex types.

XML schema definition	Supported ActionScript 3.0 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 encoding mappings from ActionScript 3.0 types to XML schema built-in types.

XML schema type	Supported ActionScript 3.0 types	Notes
`xsd:anyType` `xsd:anySimpleType`	Object	Boolean -> `xsd:boolean` ByteArray -> `xsd:base64Binary` Date -> `xsd:dateTime` int -> `xsd:int` Number -> `xsd:double` String -> `xsd:string` uint -> `xsd:unsignedInt`
`xsd:base64Binary`	`flash.utils.ByteArray`	`mx.utils.Base64Encoder` is used (without line wrapping).
`xsd:boolean`	Boolean Number Object	Always encoded as `true` or `false`. `Number == 1` then `true`, otherwise `false`. `Object.toString()` == "true" or "1" then `true`, otherwise `false`.
`xsd:byte` `xsd:unsignedByte`	Number String	String first converted to Number.
`xsd:date`	Date Number String	Date UTC accessor methods are used. Number used to set Date.time. String assumed to be preformatted and encoded as is.
`xsd:dateTime`	Date Number String	Date UTC accessor methods are used. Number used to set Date.time. String assumed to be preformatted and encoded as is.
`xsd:decimal`	Number String	`Number.toString()` is used. 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.
`xsd:gMonthDay`	Date String	`Date.getUTCMonth()` and `Date.getUTCDate()` are used. String parsed for month and day portions.

XML schema type	Supported ActionScript 3.0 types	Notes
`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 mapping from ActionScript 3.0 types to SOAP-encoded types.

SOAPENC type	Supported ActionScript 3.0 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.

Decoding XML schema and SOAP to ActionScript 3.0

The following table shows the decoding mappings from XML schema built-in types to ActionScript 3.0 types.

XML schema type	Decoded ActionScript 3.0 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 time zone information is present, local time is assumed.
`xsd:dateTime`	Date	If no time zone information is present, local time is assumed.
`xsd:decimal`	Number	Content is created via `Number(content)` and is thus limited to the range of Number.
`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 3.0 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 time zone information is present, local time is assumed.
`xsi:nil`	null	

The following table shows the decoding mappings from SOAP-encoded types to ActionScript 3.0 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 decoding mappings from custom data types to ActionScript 3.0 data types.

Custom type	Decoded ActionScript 3.0 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 4:

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

The following XML Schema structures or structure attributes are ignored and are not supported in Flex 4:

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

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

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

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

<any
processContents="..."/>

<annotation>
```

Customizing web service type mapping

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

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

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

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

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

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

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 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 application:

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

When you invoke `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.

Using RemoteObject components

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

You can also use RemoteObject components with PHP and .NET objects with third-party software, such as the open source projects AMFPHP and SabreAMF, and Midnight Coders WebORB. For more information, see the following websites:

- Zend Framework http://framework.zend.com/
- AMFPHP http://amfphp.sourceforge.net/
- SabreAMF http://www.osflash.org/sabreamf
- Midnight Coders WebORB http://www.themidnightcoders.com/

RemoteObject components use the AMF protocol to send and receive data, while WebService and HTTPService components use the HTTP protocol. AMF is significantly faster than HTTP, however server-side coding and configuration is typically more complex.

Flash Builder for PHP is a development tool created in partnership with Zend Technologies that includes an integrated copy of Zend Studio. For more information, see the Adobe website.

As with HTTPService and WebService components, you can use a RemoteObject component to display the result of a database query in an application. You can also use the component to insert, update, and delete data in a database. When the result of the query has been returned to the application, you can display it in one or more user interface controls.

For API reference information about the RemoteObject component, see mx.rpc.remoting.mxml.RemoteObject.

Sample RemoteObject application

MXML code

The application in the following example uses a RemoteObject component to call a ColdFusion component. The ColdFusion component queries a MySQL database table called users. It returns the query result to the application where it is bound to the `dataProvider` property of a DataGrid control and displayed in the DataGrid control. The application also sends the user name and e-mail address of new users to the ColdFusion component, which performs an insert into the user database table.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:mx="library://ns.adobe.com/flex/mx" minWidth="955" minHeight="600">
    <fx:Declarations>
        <mx:RemoteObject
            id="userRequest"
            destination="ColdFusion"
            source="flexapp.returnusers">

            <mx:method name="returnRecords" result="returnHandler(event)"
                       fault="mx.controls.Alert.show(event.fault.faultString)"/>
            <mx:method name="insertRecord" result="insertHandler()"
                       fault="mx.controls.Alert.show(event.fault.faultString)"/>
        </mx:RemoteObject>
    </fx:Declarations>

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

            private function returnHandler(e:ResultEvent):void
```

```
            {
                dgUserRequest.dataProvider = e.result;
            }
            private function insertHandler():void
            {
                userRequest.returnRecords();
            }
            private function clickHandler():void
            {
                userRequest.insertRecord(username.text, emailaddress.text);
            }
        ]]>
    </fx:Script>

    <mx:Form x="22" y="10" width="300">
        <mx:FormItem>
            <s:Label text="Username" />
            <s:TextInput id="username"/>
        </mx:FormItem>
        <mx:FormItem>
            <s:Label text="Email Address" />
            <s:TextInput id="emailaddress"/>
        </mx:FormItem>
        <s:Button label="Submit" click="clickHandler()"/>
    </mx:Form>

    <mx:DataGrid id="dgUserRequest" x="22" y="200">
        <mx:columns>
            <mx:DataGridColumn headerText="User ID" dataField="userid"/>
            <mx:DataGridColumn headerText="User Name" dataField="username"/>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

In this application, the RemoteObject component's `destination` property is set to `Coldfusion` and the `source` property is set to the fully qualified name of the ColdFusion component.

In contrast, when working with LiveCycle Data Services or BlazeDS, you specify a fully qualified class name in the `source` property of a remoting service destination in a configuration file, which by default is the remoting-config.xml file. You specify the name of the destination in the RemoteObject component's `destination` property. The destination class also must have a no-args constructor. You can optionally configure a destination this way when working with ColdFusion instead of by using the `source` property on the RemoteObject component.

ColdFusion component

The application calls the following ColdFusion component. This ColdFusion code performs SQL database inserts and queries and returns query results to the application. The ColdFusion page uses the `cfquery` tag to insert data into the database and query the database, and it uses the `cfreturn` tag to format the query results as a ColdFusion query object.

```
<cfcomponent name="returnusers">
    <cffunction name="returnRecords" access="remote" returnType="query">

        <cfquery name="alluserinfo" datasource="flexcf">
                SELECT userid, username, emailaddress FROM users
        </cfquery>
        <cfreturn alluserinfo>
    </cffunction>
    <cffunction name="insertRecord" access="remote" returnType="void">

        <cfargument name="username" required="true" type="string">
        <cfargument name="emailaddress" required="true" type="string">
        <cfquery name="addempinfo" datasource="flexcf">
            INSERT INTO users (username, emailaddress) VALUES (
            <cfqueryparam value="#arguments.username#" cfsqltype="CF_SQL_VARCHAR"
maxlength="255">,
            <cfqueryparam value="#arguments.emailaddress#" cfsqltype="CF_SQL_VARCHAR"
maxlength="255"> )
        </cfquery>
        <cfreturn>
    </cffunction>
</cfcomponent>
```

Calling RemoteObject components in ActionScript

In the following ActionScript example, 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>
```

Accessing Java objects in the source path

The RemoteObject component lets you access stateless and stateful Java objects that are in the LiveCycle Data Services, BlazeDS, or ColdFusion 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 specify the fully qualified class name in the `source` property of a remoting service destination in the LiveCycle Data Services, BlazeDS, or ColdFusion 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 ColdFusion, you can optionally set the RemoteObject component's `destination` property to `Coldfusion` and the `source` property to the fully qualified name of a ColdFusion component or Java class.

When you configure a remoting service destination to access stateless objects (the request scope), Flex creates a different 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 a remote object destination to access stateful objects, Flex creates the object once on the server and maintains state between method calls. If storing the object in the application or session scope causes memory problems, use the request scope.

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 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 method names as your own method names:

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

Also, do not begin method names with an underscore (_) character.

RemoteObject methods (operations) are accessible by simply naming them after the service variable. However, naming conflicts can occur if an operation name happens to match a defined method on the service. You can use the following method in ActionScript on a RemoteObject component to return the operation of the given name:

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

Serializing between ActionScript and Java

LiveCycle Data Services and BlazeDS serialize data between ActionScript (AMF 3) and Java and ColdFusion data types in both directions. For information about ColdFusion data types, see the ColdFusion documentation set.

Converting data from ActionScript to Java

When method parameters send data from an application to a Java object, the data is automatically converted from an ActionScript data type to a Java data type. When LiveCycle Data Services or 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 match exactly 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 indexes 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 indexes are consecutive, with no gap, starting from 0 (zero). A *sparse array* is one in which there are gaps between the numeric indexes; the array is treated like an object and the numeric indexes 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 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[], 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 important only 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.

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.

LiveCycle Data Services and BlazeDS handle 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 that 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.

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[ ]).

LiveCycle Data Services and BlazeDS also convert 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)`, LiveCycle Data Services and BlazeDS convert 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, LiveCycle Data Services and BlazeDS pass an instance of java.util.TreeSet to parameters typed with the java.util.SortedSet interface.

LiveCycle Data Services and BlazeDS pass 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 to be 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.

Explicitly mapping ActionScript and Java objects

For Java objects that LiveCycle Data Services and BlazeDS do 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 0bject. Private properties, constants, static properties, 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.

LiveCycle Data Services and BlazeDS use the standard Java class, java.beans.Introspector, to get property descriptors for a JavaBean 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 runtime. 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. LiveCycle Data Services and BlazeDS also handle objects found within objects. LiveCycle Data Services implicitly handles the Java data types in the following table.

Java type	ActionScript type (AMF 3)
java.lang.String	String
java.lang.Boolean, boolean	Boolean
java.lang.Integer, int	int If value < 0xF0000000 || value > 0x0FFFFFFF, the value is promoted to Number due to AMF encoding requirements.
java.lang.Short, short	int If i < 0xF0000000 || i > 0x0FFFFFFF, the value is promoted to Number.

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

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 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`, instances 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

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

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

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

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

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

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

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

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

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

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

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

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

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

The example 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.

Explicit parameter passing and parameter binding

There are two distinct ways to call HTTPService, WebService, and RemoteObject components: explicit parameter passing and parameter binding. 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.

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

When you use parameter binding, you declare RemoteObject method parameter tags nested in an `<mx:arguments>` tag under an `<mx:method>` tag, HTTPService parameter tags nested in an `<mx:request>` tag, or WebService operation parameter tags nested in an `<mx:request>` tag under an `<mx:operation>` tag. You use the `send()` method to send the request.

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 by 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.

```
<?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 components

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, 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 by 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>
```

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.

```
<?xml version="1.0"?>
<!-- fds\rpc\ROParamBind1.mxml. -->
<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 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 Data providers and collections.

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 Data providers and collections.

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 so, 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>
```

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, populate those objects by using the object tree that Flex creates.

WebService and HTTPService components both return anonymous objects and arrays that are complex types. If `makeObjectsBindable` is `true`, which it is by default, objects are wrapped in mx.utils.ObjectProxy instances and arrays are wrapped in mx.collections.ArrayCollection instances.

Note: *ColdFusion is not case sensitive, so it internally uppercases all of its data. Keep this in mind when consuming a ColdFusion web service.*

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" wsdl="http://myserver:8500/flexapp/app1.cfc?wsdl"
        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 a data access component request has been invoked. This is useful if operations are queued and invoked at a later time.

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 property of type XML. You can access the `lastResult` property by using ECMAScript for XML (E4X) expressions. You do not include the root node of the XML structure in the dot notation when using an `E4X` XML object in a binding expression; this is different from the syntax for a `lastResult` property set to object for which you do include the root node of the XML structure in the dot notation. For example, when the lastResult property is set to `e4x`, you would use `{srv.lastResult.product}`; when the lastResult property is set to object, you would use `{srv.lastResult.products.product}`.

Using a result format of `e4x` is the preferred way to work directly 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.

Note: *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 an HTTPService component or WebService operation to `e4x`, you may have to handle namespace information contained in the XML that is returned. For a WebService component, namespace information is included 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 bold 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`, create a namespace object for the http://ws.invesbot.com/namespace. The following example shows an application that does this:

```
<?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 a namespace or namespaces are 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>
```

Handling web service results that contain .NET DataSets or DataTables

Web services written with the Microsoft .NET Framework can return special .NET DataSet or DataTable objects to the client. A .NET web service provides a basic WSDL document without information about the type of data that it manipulates. When the web service returns a DataSet or a DataTable, data type information is embedded in an XML Schema element in the SOAP message, which specifies how the rest of the message must 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 web service is automatically converted to an object with a Tables property, which contains a map of one or more DataTable objects. Each DataTable object from the Tables map contains two properties: Columns and Rows. The Rows property contains the data. The event.result object gets the following properties corresponding to DataSet and DataTable properties in .NET. Arrays of DataSets or DataTables have the same structures described here, but are nested in a top-level Array on the result object.

Property	Description
`result.Tables`	Map of table names to objects that contain table data.
`result.Tables["someTable"].Columns`	Array of column names in the order specified in the DataSet or DataTable schema for the table.
`result.Tables["someTable"].Rows`	Array of objects that represent the data of each table row. For example, {columnName1:value, columnName2:value, columnName3:value}.

The following MXML application populates a DataGrid control with DataTable data returned from a .NET web service.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns="*" xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical">
    <mx:WebService
        id="nwCL"
        wsdl="http://localhost/data/CustomerList.asmx?wsdl"
        result="onResult(event)"
        fault="onFault(event)" />
    <mx:Button label="Get Single DataTable" click="nwCL.getSingleDataTable()"/>
    <mx:Button label="Get MultiTable DataSet" click="nwCL.getMultiTableDataSet()"/>
    <mx:Panel id="dataPanel" width="100%" height="100%" title="Data Tables"/>

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

            private function onResult(event:ResultEvent):void {
                // A DataTable or DataSet returned from a .NET webservice is
                // automatically converted to an object with a "Tables" property,
                // which contains a map of one or more dataTables.
                if (event.result.Tables != null)
                {
                    // clean up panel from previous calls.
                    dataPanel.removeAllChildren();

                    for each (var table:Object in event.result.Tables)
                    {
                        displayTable(table);
                    }

                    // Alternatively, if a table's name is known beforehand,
                    // it can be accessed using this syntax:
                    var namedTable:Object = event.result.Tables.Customers;
```

```
                    //displayTable(namedTable);
                }
            }

            private function displayTable(tbl:Object):void {
                var dg:DataGrid = new DataGrid();
                dataPanel.addChild(dg);
                // Each table object from the "Tables" map contains two properties:
                // "Columns" and "Rows". "Rows" is where the data is, so we can set
                // that as the dataProvider for a DataGrid.
                dg.dataProvider = tbl.Rows;
            }

            private function onFault(event:FaultEvent):void {
                Alert.show(event.fault.toString());
            }
        ]]>
    </mx:Script>

</mx:Application>
```

The following example shows the .NET C# class that is the backend web service implementation called by the 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;
    }
}
```