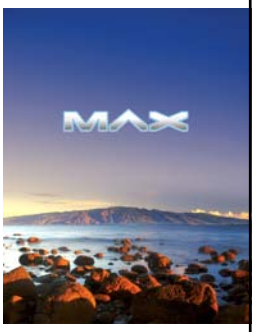
MAX 2006
Beyond Boundaries

David Gassner

Building Applications with Flex Data Services

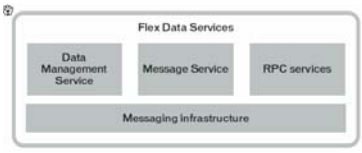
Bardo Technical Services



Overview

- What is Flex Data Services 2?
- Build Applications using:
  - Remote Object Services
  - Flex Message Service
  - Data Management Service

What is Flex Data Services?



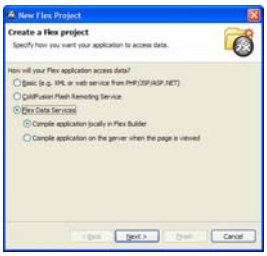
- A J2EE application that allows you to access and synchronize data across 1 or more applications built with the Flex Framework
- Runs on industry standard J2EE application servers*
- Serves as middleware between Flex application and server resources such as databases and LDAP servers

* Includes JRun, Websphere, WebLogic, JBoss, and others

Flex Data Services - Supported Services

- **Flex Message Service** for real-time chat and collaboration applications
- **Flex Data Management Service** for real-time data synchronization between a Flex client application and a web application's data tier
- **Flex Proxy Service** to route requests to remote web and HTTP services from Flex applications without needing a cross-domain policy file on the remote server
- **Remote Object Service** management to make server-side custom Java classes and ColdFusion Components accessible to Flex 2 client applications

Creating a Flex Project (Step 1)

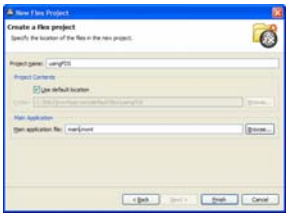


- To create a Flex project for use with Flex Data Services:
  - Select File → New → Flex Project.
  - In the **New Flex Project** dialog box, select a project that uses Flex Data Services.
  - Select one of these 2 options:
    - Compile application locally in Flex Builder
    - Compile application on the server when the page is viewed.

Creating a Flex Project (Step 2)

- Select a Flex server, including the context root

Creating a Flex Project (Step 3)



- Set the project name and main application file

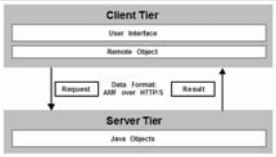
Exercise 1: Create a Project for use with FDS

- Follow the steps in the Exercises handout

Calling Java Classes as Remote Object Services

- One of the RPC services supported by Flex 2
- Other RPC services include:
  - HTTPService
  - WebService
- Allows you to call public methods of server-side Java classes as though they were local objects in a Flex client application

Remote Object Service Architecture



- Accessing remote object services is similar to that of the other RPC services:
  - Calls are made asynchronously
  - Results are processed using data binding expressions or event handlers

Action Message Format

- The communication format used with remote object services
- Binary, rather than text-based as in SOAP or HTTPService
- Significantly smaller and faster, uses less bandwidth than other communication protocols
- Supports encoding of strongly typed Transfer Object classes

Java Class Placement

- Java classes must be made accessible to the Flex Data Services classpath
- Most common strategy:
  - Classes that are compiled individually are placed in **`WEB-INF/classes`**
  - Classes that are compiled into .jar files are placed in **`WEB-INF/lib`**

Creating a Remote Object Destination

```
<destination id="sandwichService">
    <properties>
        <source>rorpc.SandwichService</source>
        <scope>application</scope>
    </properties>
</destination>

<destination id="photoService">
    <properties>
        <source>fstop.PhotoService</source>
        <scope>application</scope>
    </properties>
</destination>

<destination id="orderService">
    <properties>
        <source>fstop.OrderService</source>
        <scope>application</scope>
    </properties>
</destination>
```

- FDS Configuration files are stored in **`WEB-INF/flex`**
- Create messaging destinations in **`remoting-config.xml`**
- Each destination has a unique **`id`**
- Set the **`channel`** as either:
  - **`my-amf`**
  - **`my-secure-amf`**
- Set the scope as one of:
  - **`application`**
  - **`session`**
  - **`request`**

Using <mx:RemoteObject>

- Call the Java class from the Flex application using **`<mx:RemoteObject>`**

```
<mx:RemoteObject
  id="myService"
  destination="sandwichService"
  result="resultHandler(event)"
  fault="faultHandler(event)"/>
```

- RemoteObject class can also be instantiated in ActionScript

```
import mx.rpc.remoting.RemoteObject;
var myService:RemoteObject = new RemoteObject();
myService.destination = "sandwichService";
myService.addEventListener("result", resultHandler);
myService.addEventListener("fault", faultHandler);
```


Calling a Remote Method

- Call remote method using same syntax as in Java:

```
<mx:Button label="Get Data"
  click="myService.getAllSandwiches()"/>
```

- Pass arguments in same position as defined in Java class:

```
<mx:Button label="Get Data"
  click="myService.getSandwichesByType('Hot')"/>
```


Binding to remote method results

- To bind to results upon method return, use this syntax:

```
serviceid.methodName.lastResult
```

- For instance:

```
<mx:RemoteObject
  id="myService"
  destination="sandwichService"/>

<mx:Button label="Get Data"
  click="myService.getAllSandwiches()"/>

<mx:DataGrid
  dataProvider="{myService.getAllSandwiches.lastResult}"/>
```


Exercise 2: Call a remote object service

- Follow the instructions in the exercises handout

Handling RemoteObject events

- Two primary events are used to capture results of remote method calls:
  - **`result`**
    - occurs upon successful completion of a remote method call
    - returns instance of **`mx.rpc.events.ResultEvent`**
    - event's **`result`** property is reference to returned data
  - **`fault`**:
    - occurs upon failure of remote method call
    - returns instance of **`mx.rpc.events.FaultEvent`**
    - event's **`fault`** property contains information about the failure

Handling the result event

- Create persistent variable to store data, and a custom event handler method to capture and save reference to result

```
[Bindable]
private var sandwichData:ArrayCollection;
private function myResultHandler(event:ResultEvent):void {
  sandwichData = event.result as ArrayCollection;
}
```

- Add an event listener to the remote object definition

```
result=myResultHandler(event);
```


Handling multiple method results

- If calling multiple methods, use **<mx:method>** compiler tag to dispatch events to multiple ActionScript methods:

```
<mx:RemoteObject id="myService"
  destination="roDestination"
  fault="faultHandler(event)">
  <mx:method name="getAllSandwiches"
      result="getAllHandler(event)"/>
  <mx:method name="getSandwichByID"
      result="getOneHandler(event)"/>
</mx:RemoteObject>
```

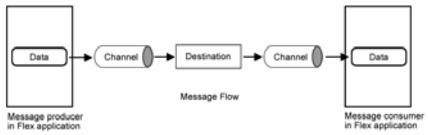

Exercise 3: Handling RemoteObject Events

- Follow the instructions in the exercises handout

The Flex Message Service

- The foundation of Flex Data Services
- Uses client- and server-side components, allowing you to send and received messages of many kinds in a multi-user application
- Use **<mx:Producer>** and **<mx:Consumer>** client-side components to send and receive messages
- Define a server-side message destination through which messages will be routed

Publish-Subscribe Messaging



- The foundation of Flex Data Services
- Uses client- and server-side components, allowing you to send and received messages of many kinds in a multi-user application.
- Use **<mx:Producer>** and **<mx:Consumer>** components to send and receive messages.
- Define a server-side message destination through which messages will be routed.

Creating a Destination

```
<destination id="restaurantDestination">
  <channels>
    <channel ref="my-polling-amf"/>
  </channels>
</destination>

<destination id="chatDestination">
  <channels>
    <channel ref="my-rtmp"/>
  </channels>
</destination>

<destination id="photoDestination">
  <channels>
    <channel ref="my-rtmp"/>
  </channels>
</destination>
```

- FDS Configuration files are stored in WEB-INF/flex directory
- Create messaging destinations in messaging-config.xml file
- Each destination has a unique id
- Set the channel as either:
  - **my-rtmp**
  - **my-polling-amf**

Using <mx:Producer>

- The Producer component sends the message to the server-side destination
- Declare **<mx:Producer>** in the Flex application
- Set its destination as named in the configuration file

```
<mx:Producer id="producer"
  destination="chatDestination"/>
```


Sending the Message

- To send a message:
  - Create a new instance of the AsyncMessage class
  - Add the object's **body** property to send the primary message
  - Add **headers** properties for additional information
  - Send the message

```
import mx.messaging.messages.AsyncMessage;
var message:AsyncMessage = new AsyncMessage();
message.headers.user = user;
message.body = msg.text;
producer.send(message);
```


Using <mx:Consumer>

- The Consumer component receives the message from the server-side destination
- Declare **<mx:Consumer>** in the Flex application
- Set its destination as named in the configuration file

```
<mx:Consumer id="consumer"
  destination="chatDestination"/>
```

- Call its **subscribe()** method to receive messages:

```
consumer.subscribe();
```


Receiving Messages

- When a message is received, the Consumer dispatches a **message** event
- The event object is an instance of **MessageEvent**
- The event object's **message** property is an instance of **AsyncMessage** containing the message contents
- Create an event handler method:

```
private function messageHandler(event:MessageEvent):void {
 mx.controls.Alert.show(
  event.message.body, "Message received from " +
  event.message.headers.user);
}
```

- Register the Consumer to listen for the event:

```
message="messageHandler(event)"
```

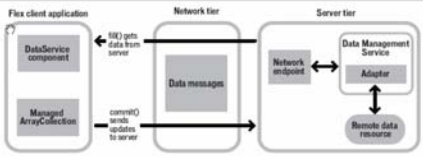

Exercise 4: Create a Messaging Application

- Follow the steps in the Exercises handout

Data Management Service

- Provides top-level functionality for distributed data in Flex applications
- Synchronizes data between client and server, with instant update of data on multiple Flex clients
- Based on Flex Message Service
- Messages are sent over one of these protocols:
  - **RTMP** for server-pushed communications
  - **AMF** with client polling for client-pulled communications

Data Management Service Architecture



- Data moves from server to client as "managed" data
- Client uses ArrayCollection to store data
- Server uses Java-based code to exchange data with server-side data store

Creating a Data Management Service Destination

- FDS Configuration files are stored in **`WEB-INF/flex`** directory
- Create destinations in **`data-management-config.xml`**
- Each destination has a unique **`id`**
- Set the **`channel`** as either:
  - **`my-rtmp`**
  - **`my-polling-amf`**

Configuring the Destination

```
<destination id="sandwich">
  <adapter ref="java-dao" />
  <properties>
    <source>dms.SandwichAssembler</source>
    <scope>application</scope>
    <metadata>
        <identity property="sandwichId"/>
    </metadata>
    <server>
        <fill-method>
            <name>loadSandwiches</name>
        </fill-method>
        <sync-method>
            <name>syncSandwiches</name>
        </sync-method>
    </server>
  </properties>
</destination>
```

Java class

Fill method definition

Sync method definition

Java Assembler Code

```
package dms;
import statements . . .
public class SandwichAssembler
{
  public List loadSandwiches()
  {
   SandwichDAO dao = new SandwichDAO();
   return dao.getSandwiches();
  }
  public List syncSandwiches(List changes)
  {
   . . . code to manage data changes . . .
  }
}
```


Java Sync Method Code

```
public List syncSandwiches(List changes) {
   Iterator iterator = changes.iterator();
   ChangeObject co;
   while (iterator.hasNext())
   {
    co = (ChangeObject) iterator.next();
    if (co.isCreate()) {
            co = doCreate(co);
    }
    else if (co.isUpdate()) {
            doUpdate(co);
    }
    else if (co.isDelete()) {
            doDelete(co);
    }
   }
   return changes;
}
```


Using <mx:DataService>

- Declare an ArrayCollection for client-side data access:

  `<mx:ArrayCollection id="acSandwiches"/>`

- Client uses <mx:DataService> component to link to server-side destination

  `<mx:DataService id="dsSandwiches" destination="sandwich"/>`

- Fill data from server destination with fill() method:

  `dsSandwiches.fill( acSandwiches );`

Exercise 5: Filling an ArrayCollection

- Follow the instructions in the exercises handout

Explicitly Handling Data Changes

- DataService **`autoCommit`** property is **`true`** by default
- For full application functionality, set **`autoCommit`** to **`false`** and handle commit/revert functionality explicitly
- To commit pending changes:

```
dsSandwiches.commit()
```

- To revert pending changes to existing data:

```
dsSandwiches.revert()
```

- To check whether changes are pending, use Boolean **`commitRequired`** property

Debugging Message Traffic

- **`<mx:TraceTarget>`** enables tracing of client-server messaging traffic
- Add tag to code:
  - **`<mx:TraceTarget>`**
- Run in debug mode
- Look at Flex Builder's Console view

Exercise 6: Explicit Data Changes and Debugging

- Follow the instructions in the exercises handout

More Information

- Read these sections in **Developing Flex Applications**
  - Remote Object Services
    - Understanding RPC Components
    - Using RPC Components
    - Configuring RPC Services
  - Flex Message Service
    - Understanding Flex Messaging
    - Using Flex Messaging
    - Configuring the Message Service
  - Flex Data Service
    - Understanding the Flex Data Management Service
    - Distributing Data in Flex Applications
    - Configuring the Data Management Service

Useful DevNet Articles

- These articles can be found at the Flex Developer Center at: **`http://www.adobe.com/devnet/flex`**
  - ***Using RPC services in Flex Data Services 2***, by Peter Farland
  - ***Introduction to the Flex Message Service***, by Sean Neville
  - ***Refactoring Flex applications from RPC-style data management to Data Management Services***, by James Ward
  - ***Architecting RIAs with Flex Data Management Services***, by Jeff Vroom

Speaker Contact Information

- For session code or questions, contact David Gassner at Bardo Technical Services
- Web: `http://www.bardotech.com`
- Email: `david@bardotech.com`

Better by Adobe™