

***Getting Started
with
Real-Time Flex and AIR
via
LiveCycle Data Services***

MAX 2008 MegaLab

Ben Forta

1: Setup And Configuration

Before you proceed, please note the following: Throughout this document, the following styles are used to simplify instructions. **Menu items** and **field prompts** are formatted in **bold**. Text you need to enter is always in **monospace**. And any special notes are in boxes like this.

This hands-on lab requires that you have the following installed:

- Flex Builder 3 (trial edition is acceptable)
- AIR runtime
- MegaLab image (ZIP file) which contains Apache Tomcat, LCDS 2.6, ColdFusion 8 Developer Edition, and sample databases

The MegaLab image ZIP file should be expanded in the root of your hard drive. This will create a folder named **lcds** containing other files and folders. Once the ZIP file has been expanded, you will need to do the following to start the servers:

1. Open the **/lcds/sampledb** folder.
2. Start the sample database - Windows users run **startdb.bat**, Mac users run **startdb.sh**.
3. Open the **/lcds/tomcat/bin** folder.
4. Start Tomcat - Windows users run **startup.bat**, Mac users run **startup.sh**.

The above steps will open two windows, and both will (eventually) show messages indicating that the servers have successfully loaded. This may take several minutes, depending on your computer speed and resources. To verify that the servers have loaded properly, open a browser and go to the following URL:

http://localhost:8400/

You should see a screen welcoming you to Adobe LiveCycle Data Services 2.6. If you see this message, then you should be all set for the lab.

If you do not see the welcome screen, or if the server startup scripts fail to run or generate errors, please see the instructor or lab assistants before the session begins.

To shutdown LCDS, use **shutdown.bat** or **shutdown.sh** in the same **/lcds/tomcat/bin** folder.

2: Lab 1 – Retrieving Data Via HTTP Request

This lab demonstrates the simplest form of Flex data integration, `<mx:HTTPService>` which provides the ability to invoke any HTTP URL.

Create New Project

To create a new project, perform the following steps:

1. If you have not yet opened Flex Builder 3, please do so now.
2. Close any open projects in Flex Builder.
3. Open the **File** menu and select **New**, and then **Flex Project** to display the **New Flex Project** dialog.
4. Fill in the following fields:
 - **Project name** = `GetDataHttp`
 - **Application type** = **Web application**
 - **Application server type** = **J2EE**
 - **Use remote object access service** = checked
 - **LiveCycle Data Services** = checked
5. Click the **Next** button.
6. On the next dialog screen, fill in the following fields:
 - **Use default location for local LiveCycle Data Services folder** = unchecked
 - **Root folder** = the path to the `l cds- samples` folder (usually `C:\l cds\tomcat\webapps\l cds- samples` or `/l cds/tomcat/webapps/l cds- samples`)
 - **Root URL** = `http://localhost:8400/l cds- samples/`
 - **Context root** = `/l cds- samples`
7. Click the **Validate Configuration** button and make sure that the top of the dialog box indicates that the folder and URL are valid.
8. Click the **Next** button.
9. Fill in the following fields:
 - **Main Application File** = `Main.mxml`
10. Click the **Finish** button to create and open the new project.

Remember the steps used here, they will be repeated for each project you'll create. However, the exact steps will not be repeated, and just the unique values for the specific project will be provided. You can refer back to these steps as needed.

Enter Your Code

The newly created **Main.mxml** file will contain some default code. Replace it with the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HTTPService id="srv"
        url="/1cds-samples/testdrive-httpservice/catalog.jsp" />

    <mx:DataGrid width="100%" height="100%"
        dataProvider="{srv.lastResult.catalog.product}" />

    <mx:Button label="Get Data" click="srv.send()" />

</mx:Application>
```

Test Your Code

Your application should now be tested:

1. Once you have entered the code, save your work (you can press **Ctrl-S**, or select **Save** from the **File** menu).
2. To run your application, click the Run button (it has a green circle with a white right facing arrow in it). If you are prompted for the type of application, select **Flex**.
3. Your application will be displayed in your web browser. Once it has been displayed, click the **Get Data** button to retrieve and display the data via HTTP call.

Use A Destination Instead Of A Hardcoded URL

The application currently includes a hardcoded URL for the HTTP service. While this is fine for simple testing, the practice should be avoided in real applications. URLs can be stored in external XML files along with a destination name, and then referred to within your code by the destination name. Do the following:

1. Destinations are stored in `/l cds/tomcat/webapps/l cds- sampl es/WEB- INF/fl ex/proxy- confi g. xml`. Feel free to open this file, you'll see that it defines a destination named `catalog` that maps to the URL used in the `<mx: HTTPService>` tag.
2. Go back to Flex Builder, and make sure file `Main.mxml` is open for editing.
3. Locate the line of code containing the `<mx: HTTPService>` tag, and edit it so that it looks like the following:

```
<mx:HTTPService id="srv" destination="catalog" />
```

4. Save the changes, and run the application again to ensure that it executes correctly.

3: Lab 2 – Retrieving Data Via Web Service

This lab demonstrates the use of `<mx:WebService>` which provides the ability to invoke SOAP based web services from within Flex (and AIR) applications.

Create New Project

Create a new project for this lab:

1. Close any existing Flex Builder projects.
1. Create a new Flex Builder project (using the same steps as used previously) specifying the following:
 - **Project name** = GetDataWs
 - **Application type** = Web application
 - **Application server type** = J2EE
 - **Use remote object access service** = checked
 - **LiveCycle Data Services** = checked
 - **Use default location for local LiveCycle Data Services folder** = unchecked
 - **Root folder** = the path to the l cds- samples
 - **Root URL** = `http://localhost:8400/l cds- samples/`
 - **Context root** = `/l cds- samples`
 - **Main Application File** = `Main.mxml`

Enter Your Code

The newly created `Main.mxml` file will contain some default code. Replace it with the following code:

```
<?xml version="1.0" encoding="utf-8"?>

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

    <mx:WebService id="srv"

        wsdl="http://localhost:8400/cfusion/service/artists.cfc?wsdl"

        showBusyCursor="true"/>

    <mx:DataGrid width="100%" height="100%"

        dataProvider="{srv.list.lastResult}" />

</mx:Application>
```

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

```
</mx:Application>
```

In this example, a hard coded WSDL URL was used. However, a destination could have been defined in **proxy-config.xml**, as notes previously.

Test Your Code

You can now save any code changes and test your code. You should be able to click the **Get Data** button to retrieve data from ColdFusion.

4: Lab 3 – Retrieving Data Via Flash Remoting

This lab demonstrates the use of `<mx:RemoteObject>` which is used to access data via Flash Remoting from within Flex (and AIR) applications.

Create New Project

Create a new project for this lab:

2. Close any existing Flex Builder projects.
3. Create a new Flex Builder project (using the same steps as used previously) specifying the following:
 - **Project name** = GetDataRemoting
 - **Application type** = Web application
 - **Application server type** = J2EE
 - **Use remote object access service** = checked
 - **LiveCycle Data Services** = checked
 - **Use default location for local LiveCycle Data Services folder** = unchecked
 - **Root folder** = the path to the `l cds- samples`
 - **Root URL** = `http://localhost:8400/l cds- samples/`
 - **Context root** = `/l cds- samples`
 - **Main Application File** = `Main.mxml`

Enter Your Code

The newly created `Main.mxml` file will contain some default code. Replace it with the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundColor="#FFFFFF">
    <mx:RemoteObject id="srv" destination="product"/>
    <mx:DataGrid width="100%" height="100%"
        dataProvider="{srv.getProductsWithLastResult}" />
</mx:Application>
```

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

```
</mx:Application>
```

In this example, a **destination** was specified, instead of a hardcoded path. This destination is defined in **remoting-config.xml**.

Test Your Code

You can now save any code changes and test your code. Click the **Get Data** button to retrieve data via Flash Remoting.

5: Lab 4 – Consuming A Data Feed

This lab demonstrates the use of `<mx: Consumer>` which is used to subscribe to, and consume from, a LiveCycle Data Services data feed.

Create New Project

Create a new project for this lab:

1. Close any existing Flex Builder projects.
2. Create a new Flex Builder project (using the same steps as used previously) specifying the following:
 - **Project name** = ConsumeFeed
 - **Application type** = Web application
 - **Application server type** = J2EE
 - **Use remote object access service** = checked
 - **LiveCycle Data Services** = checked
 - **Use default location for local LiveCycle Data Services folder** = unchecked
 - **Root folder** = the path to the `l cds- samples`
 - **Root URL** = `http://localhost:8400/l cds- samples/`
 - **Context root** = `/l cds- samples`
 - **Main Application File** = `Main.mxml`

Enter Your Code

The newly created `Main.mxml` file will contain some default code. Replace it with the following code:

```
<?xml version="1.0" encoding="utf-8"?>

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

    <mx: Script>
    <![CDATA[
        import mx.messaging.messages.IMessage;

        // Incoming message handler
```

```

        private function messageHandler(message: IMessage): void
        {
            lblValue.text = message.body.toString();
        }
    ]]>
</mx:Script>

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

<mx:HBox>
    <mx:Button label="Subscribe to 'feed' destination"
        click="consumer.subscribe()"
        enabled="{!consumer.subscribed}"/>
    <mx:Button label="Unsubscribe from 'feed' destination"
        click="consumer.unsubscribe()"
        enabled="{consumer.subscribed}"/>
</mx:HBox>

<mx:Label id="lblValue" fontWeight="bold"/>

</mx:Application>

```

The specified **destination**, (here named **feed**), is defined in **messaging-config.xml**.

Test Your Code

You should now try out your application:

1. Save your application.
2. The LCDS feed (named **feed**) publishes randomly generated numbers. To start the feed, open your browser and go to the following URL:

<http://localhost:8400/lcds-samples/testdrive-datapush/startfeed.jsp>

3. Now run your Flex application.
4. Click the **Subscribe to 'feed' destination** button to subscribe to the LCDS feed, you will see the generated numbers displayed as they are generated and pushed from the server.
5. Click the **Unsubscribe from 'feed' destination** button to unsubscribe from the LCDS feed. You can subscribe and unsubscribe as desired.
6. To stop generating numbers for the feed, open your browser and go to the following URL:

<http://localhost:8400/lcds-samples/testdrive-datapush/stopfeed.jsp>

7. You can subscribe to an unsubscribe from the feed regardless of whether or not it is publishing data. If subscribed while no data is being published, the Flex application will simply remain connected to the feed waiting for data until it is unsubscribed.

6: Lab 5 – Using Data Feeds With Charting

This lab continues demonstrating the use of `<mx: Consumer>`, this time in conjunction with Flex charting components.

Enter Your Code

For this lab you may create a new project, or simply update the previous project. Either way, enter the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<mx: Application xmlns:mx="http://www.adobe.com/2006/mxml"
                 backgroundColor="#FFFFFF">

    <mx: Script>
    <![CDATA[

        import mx.messaging.messages.IMessage;
        import mx.collections.ArrayCollection;

        [Bindable]
        // Variable to store received feed values
        public var chartData:ArrayCollection = new ArrayCollection();

        // Incoming message handler
        private function messageHandler(message:IMessage):void
        {
            // Received data from feed, add to array
            chartData.addItem(message.body);
        }

        // Subscribe/unsubscribe
        private function toggleSubscribe():void
        {
```

```

        if (btnSubscribe.selected)
            consumer.subscribe();
        else
            consumer.unsubscribe();
    }
]]>
</mx:Script>

<!-- Consume feed -->
<mx:Consumer id="consumer"
    destination="feed"
    message="messageHandler(event.message)"/>

<!-- Define chart -->
<mx:LineChart width="100%" height="100%"
    dataProvider="{chartData}"
    showDataTips="true">
    <mx:series>
        <mx:LineSeries displayName="Data" />
    </mx:series>
</mx:LineChart>

<!-- Control bar -->
<mx:ControlBar width="100%">
    <mx:Button id="btnSubscribe"
        toggle="true"
        label="Subscribe to feed"
        click="toggleSubscribe()" />
    <mx:Spacer width="100%" />
    <mx:Label text="{chartData.length} values received" />
    <mx:Button label="Reset Data"

```

```
click="chartData.removeAll()" />  
</mx:ControlBar>
```

```
</mx:Application>
```

Test Your Code

You should now try out your application:

1. Save your application.
2. Once again, start the feed using the following URL:

<http://localhost:8400/lcds-samples/testdrive-datapush/startfeed.jsp>

3. Now run your Flex application.
4. Use the **Subscribe to feed destination** to toggle feed subscription.
5. As before, to stop generating numbers for the feed, use the following URL:

<http://localhost:8400/lcds-samples/testdrive-datapush/stopfeed.jsp>

7: Lab 6 – Producing And Consuming Feeds (Chat)

This lab further demonstrates the use of `<mx: Consumer>` and also introduces `<mx: Producer>` which is used to publish to a LiveCycle Data Services data feed.

Create New Project

Create a new project for this lab:

1. Close any existing Flex Builder projects.
2. Create a new Flex Builder project (using the same steps as used previously) specifying the following:
 - **Project name** = Chat
 - **Application type** = Web application
 - **Application server type** = J2EE
 - **Use remote object access service** = checked
 - **LiveCycle Data Services** = checked
 - **Use default location for local LiveCycle Data Services folder** = unchecked
 - **Root folder** = the path to the `l cds- sampl es`
 - **Root URL** = `http://localhost:8400/l cds- sampl es/`
 - **Context root** = `/l cds- sampl es`
 - **Main Application File** = `Main.mxml`

Enter Your Code

The newly created `Main.mxml` file will contain some default code. Replace it with the following code:

```
<?xml version="1.0" encoding="utf-8"?>

<mx: Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="initApp()">

    <mx: Script>
    <![CDATA[
        import mx.messaging.messages.AsyncMessage;
        import mx.messaging.messages.IMessage;
```

```

// On startup
private function initApp():void
{
    // Subscribe to chat
    consumer.subscribe();
}

// Send chat message
private function send():void
{
    var message:IMessage = new AsyncMessage();
    message.body.chatMessage = msg.text;
    producer.send(message);
    msg.text = "";
}

// Process received chat message
private function messageHandler(message:IMessage):void
{
    log.text +=
        message.body.chatMessage + "\n";
}
]]>
</mx:Script>

<!-- Need to produce and consume -->
<mx:Producer id="producer"
    destination="chat" />
<mx:Consumer id="consumer"
    destination="chat"
    message="messageHandler(event.message)" />

```

```
<!-- Chat window -->
<mx:Panel title="Chat" width="100%" height="100%">
  <mx:TextArea id="log" width="100%" height="100%" />
  <mx:Control Bar>
    <mx:TextInput id="msg" width="100%"
      enter="send() " />
    <mx:Button label="Send"
      click="send() " />
  </mx:Control Bar>
</mx:Panel >

</mx:Appl i cat i on>
```

In this example, a feed destination named **chat** is used by both **<mx:Producer>** and **<mx:Consumer>**. This destination is defined in **messaging-config.xml**.

Test Your Code

You should now try out your application:

1. Save your application.
2. Run the application twice (you need to be running two or more copies to be able to chat). Any text typed in one chat window is echoed in all other chat windows.

8: Lab 6a – Producing And Consuming Complex Data (Optional)

This optional lab adds functionality to the chat application created in Lab 6, displaying a popup window to prompt for a login, and passing multiple items in the message body.

Enter Your Code

This lab updates the project created in the previous lab. It requires the creation of a new MXML component in your project. The new component, named **Login.mxml**, is the popup login window, and is based on **<mx:TitleWindow>**. To add this component:

1. Click on **src** in the **Chat** project
2. Open the **File** menu and select **New** and then **MXML Component**
3. Here is the code for **Login.mxml**
4. In the **New MXML Component** dialog, specify the following:
 - **Filename** = **Login**
 - **Based on** = **TitleWindow**
 - **Width** = **300**
 - **Height** = **150**
5. Click **Finish**

Then update the new **Login.mxml** component so that it looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:TitleWindow xmlns:mx="http://www.adobe.com/2006/mxml"
width="300" height="150"
title="Login"
creationComplete="PopUpManager.centerPopUp(this)">
<mx:Script>
    <![CDATA[
        import mx.managers.PopUpManager;

        private function processLogin():void
```

```

        {
            // Save the handle
            parentApplication.nickname=nickname.text;
            // Close popup window
            PopupManager.removePopup(this);
        }
    ]]>
</mx:Script>

<!-- Login form -->
<mx:Form width="100%" height="100%">
    <mx:FormItem label="Name: " width="100%">
        <mx:TextInput id="nickname" width="100%"
            enter="processLogin();" />
    </mx:FormItem>
    <mx:FormItem width="100%">
        <mx:Button label="Login"
            click="processLogin();" />
    </mx:FormItem>
</mx:Form>

</mx:TitleWindow>

```

To use this code you will also need to update the `<mx:Script>` block in `Main.mxml`. Here is the updated code (to make editing the code easier, changed code has been highlighted below):

```

<mx:Script>
<![CDATA[
    import mx.messaging.messages.AsyncMessage;
    import mx.messaging.messages.IMessage;
    import mx.managers.PopupManager;
    import mx.core.IFlexDisplayObject;

```

```

// User handle
[Bindable]
public var nickname:String="Guest";

// On startup
private function initApp():void
{
    // Subscribe to chat
    consumer.subscribe();
    // Create login window
    var loginWindow:IFlexDisplayObject =
        PopUpManager.createPopUp(this, Login, true);
}

// Send chat message
private function send():void
{
    var message:IMessage = new AsyncMessage();
    message.body.sender = nickname;
    message.body.chatMessage = msg.text;
    producer.send(message);
    msg.text = "";
}

// Process received chat message
private function messageHandler(message:IMessage):void
{
    log.text +=
        message.body.sender + ": " +
        message.body.chatMessage + "\n";
}

```

```
    }  
  ]]>  
</mx:Script>
```

And finally, in file **Main.mxml**, locate the line with the **<mx:Panel>** tag. You'll want to update this so that it shows the name of the logged in user. Update the title attribute so that the tag looks like this:

```
<mx:Panel title="Chat [{nickname}]"  
          width="100%" height="100%">
```

Test Your Code

You should now try out your application:

1. Save your changes.
2. Run the application twice (you need to be running two or more copies to be able to chat). This time each window will prompt you to login. Any text typed in one chat window is echoed in all other chat windows, and the displayed text will correctly identify the user who sent it.

9: Lab 7 – Using Data Services

This lab further introduces `<mx: DataService>` which is used to connect to LiveCycle Data Services, seamlessly synchronizing data on both the client and the server.

Create New Project

Create a new project for this lab:

3. Close any existing Flex Builder projects.
4. Create a new Flex Builder project (using the same steps as used previously) specifying the following:
 - **Project name** = `DataService`
 - **Application type** = **Web application**
 - **Application server type** = **J2EE**
 - **Use remote object access service** = checked
 - **LiveCycle Data Services** = checked
 - **Use default location for local LiveCycle Data Services folder** = unchecked
 - **Root folder** = the path to the `l cds- samples`
 - **Root URL** = `http://localhost:8400/l cds- samples/`
 - **Context root** = `/l cds- samples`
 - **Main Application File** = `Main.mxml`

Enter Your Code

The newly created `Main.mxml` file will contain some default code. Replace it with the following code:

```
<?xml version="1.0" encoding="utf-8"?>

<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns="*"
    backgroundColor="#FFFFFF"
    creationComplete="initApp()" >

    <mx:Script>
    <![CDATA[

        public function initApp():void
```

```

        {
            ds.fill(products);
        }
    ]]>
</mx:Script>

<!-- Define ArrayCollection to store data -->
<mx:ArrayCollection id="products"/>

<!-- Define DataService connection -->
<mx:DataService id="ds" destination="inventory"/>

<!-- Force Product class to be seen -->
<Product/>

<mx:DataGrid dataProvider="{products}"
    width="100%" height="100%">
    <mx:columns>
        <mx:DataGridColumn dataField="name"
            headerText="Name"/>
        <mx:DataGridColumn dataField="category"
            headerText="Category"/>
        <mx:DataGridColumn dataField="price"
            headerText="Price"/>
        <mx:DataGridColumn dataField="image"
            headerText="Image"/>
        <mx:DataGridColumn dataField="description"
            headerText="Description"/>
    </mx:columns>
</mx:DataGrid>

</mx:Application>

```

This application refers to a class (or component) named **Product**. That class does not yet exist, so if you try to compile the application now the compiler will generate an error. To create the **Product** class:

6. Click on **src** in the **DataService** project
7. Open the **File** menu and select **New** and then **ActionScript Class**
8. Here is the code for **Product.as**
9. In the **New ActionScript Class** dialog, specify the following:
 - o **Name = Product**
10. Click **Finish**

Then update the new **Product.as** file so that it looks like this:

```
package
{
    [Managed]
    [RemoteClass(alias="flex.samples.product.Product")]
    public class Product
    {
        public function Product()
        {
        }

        public var productId: int;
        public var name: String;
        public var description: String;
        public var image: String;
        public var category: String;
        public var price: Number;
        public var qtyInStock: int;
    }
}
```

In this example, a data services destination named **inventory** is used by `<mx:DataService>`. This destination is defined in **data-management-config.xml**.

Test Your Code

You can now try out your application:

1. Save your changes.
2. Run the application. Upon startup the application will connect to LiveCycle Data Services, and synchronize the local **products** array with the data stored in the **inventory** destination on the server.

10: Lab 8 – Using Data Services For Data Editing

This lab adds functionality to the data services application, allowing data to be edited, and ensuring that multiple clients are kept in synch when edits occur.

Enter Your Code

Select file **Main.mxml**, and locate the line containing the **<mx:DataGrid>** tag. Edit the tag so that it looks like the following:

```
<mx:DataGrid dataProvider="{products}"
    editable="true"
    width="100%" height="100%">
```

Test Your Code

You can now try out your application:

1. Save your changes.
2. Run the application twice (in two browser windows). Upon startup the application will connect to LiveCycle Data Services, and synchronize the local **products** array with the data stored in the **inventory** destination on the server. You'll be able to click on any cell in the data grid to make edits, and any edits made in one grid will be immediately reflected in the other.

11: Lab 9 – Controlling Commits And Handling Conflicts

This lab adds functionality to the data services application, giving you the ability to explicitly control when commits occur, and demonstrating how to trap the occurrence of conflicts.

Enter Your Code

Select file **Main.mxml**, and locate the line containing the **<mx:DataService>** tag. Edit the tag so that it looks like the following:

```
<mx:DataService id="ds"
    destination="inventory"
    autoCommit="false"
    conflict="conflictHandler(event)" />
```

Now that commits will no longer happen automatically, you'll need a way to save changes. Add the following **<mx:Button>** below the data grid (after **</mx:DataGrid>** and before **</mx:Application>**):

```
<mx:Button label="Save Changes"
    click="ds.commit()" />
```

And finally, update the **<mx:Script>** block so that it looks like the following (everything but the **initApp()** function is new):

```
<mx:Script>
<![CDATA[
    import mx.controls.Alert;
    import mx.data.Conflict;
    import mx.data.Conflicts;
    import mx.data.events.DataConflictEvent;

    // On startup
    public function initApp():void
    {
        ds.fill(products);
    }
}
]]>
```

```

    }

    // When a conflict occurs
    public function conflictHandler(event: DataConflictEvent): void
    {
        var conflicts: Conflicts = ds.conflicts;
        var conflict: Conflict;

        // Loop through conflicts
        for (var c:int=0; c<conflicts.length; c++)
        {
            // Get conflict
            conflict = Conflict(conflicts.getItemAt(c));
            // Notify user
            Alert.show("A conflict has occurred. Using server value.",
                       "Conflict");
            // Accept server value
            conflict.acceptServer();
        }
    }
}]]>
</mx:Script>

```

For the sake of simplicity, when a conflict occurs (the value on the server is not the same as the one currently on the client), this example accepts the server value (by calling the **acceptServer()** method). In your own code you may want to present users with the conflicting values allowing them to decide which they want, and then accepting the server value or the client value (by calling the **acceptClient()** method) for each conflict.

Test Your Code

You can now try out your application, making conflicting changes, and then clicking the **Save Changes** button.

12: Lab 10 – Building AIR Applications

This lab adds demonstrates how to turn a Flex Data Services application into an AIR application.

Create New Project

Create a new project for this lab:

1. Close any existing Flex Builder projects.
2. Create a new Flex Builder project (using the same steps as used previously) specifying the following:
 - **Project name** = AIRDataService
 - **Application type** = Desktop application
 - **Application server type** = J2EE
 - **Use remote object access service** = checked
 - **LiveCycle Data Services** = checked
 - **Use default location for local LiveCycle Data Services folder** = unchecked
 - **Root folder** = the path to the `l cds- sampl es`
 - **Root URL** = `http://localhost:8400/l cds- sampl es/`
 - **Context root** = `/l cds- sampl es`
 - **Main Application File** = `Main.mxml`

Enter Your Code

The newly created **Main.mxml** file will contain some default code. This application is the AIR version of the previous Flex application. So, do the following:

1. Open the **DataService** project, and copy all of the contents of **Main.mxml**, and paste into the **Main.mxml** in the new **AIRDataService** project.
2. AIR applications should be of type `<mx:WindowedApplication>`, so change the `<mx:Application>` tag at the top of **Main.mxml** to `<mx:WindowedApplication>` and change the `</mx:Application>` tag at the bottom of **Main.mxml** to `</mx:WindowedApplication>`.
3. Copy file **Product.as** from the **DataService** project to the **AIRDataService** project (make sure to place it in the **src** folder).

If you were to run compile and run the application now, the AIR app would run, but no data would be displayed. This is because the default configurations provided with LCDS are setup for applications to call back to the servers that served them. This is how the previous Flex example was able to connect to LCDS, but connecting to the server that served the Flex app. But AIR apps are not served by a server, and so the AIR app need to know exactly which server to connect to. To make this change, do the following:

1. Locate the **services-config.xml** file (in **/lcds/tomcat/webapps/lcds-samples/WEB-INF/flex** folder by default).
2. Open the file for editing.
3. Locate the **<channel-definition>** for **id="my-rtmp"**.
4. Change the endpoint so that it looks like this (replacing **{server.name}** with **local host**):

```
<endpoint url="rtmp://local host: 2037" class="flex.messaging.endpoints.RTMPEndpoint"/>
```

5. You'll need to force the app to be recompiled for this change to be recognized.

Test Your Code

You can now try out your application, which should function exactly like the Flex version created previously. You can also open both apps, the web Flash Player version as well as the AIR version. You should be able to make changes in either application, and the other application will correctly reflect any data changes.

13: Lab 11 – Offline Processing Support

This lab adds demonstrates how to take LCDS data offline when needed, and automatically synchronizing data upon reconnection.

Enter Your Code

Select file **Main.mxml**, and locate the line containing the **<mx:DataService>** tag. Edit the tag so that it looks like the following:

```
<mx:DataService id="ds"
    destination="inventory"
    autoCommit="false"
    conflict="conflictHandler(event)"
    fault="faultHandler(event)"
    autoSaveCache="true"
    cacheID="MAX2008Lab" />
```

Next, locate the **<mx:Script>** block, and add the following **import** (type it with the existing **import** statement):

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

And finally, add this function in the **<mx:Script>** block:

```
// Fault handler
public function faultHandler(event: FaultEvent): void
{
}
```

Test Your Code

You can now try out your application. When the AIR version retrieves data, it will cache it in the local SQLite database, ensuring that it will be available even when offline. You can test this ability by shutting down LiveCycle Data Services down. The AIR application will continue to function, and you'll even be able to edit data. And when the LCDS server is restarted, the changes will be synchronized back to the server.