

Powering Flex Applications with BlazeDS and LiveCycle Data Services

Tutorial

- Messaging
- Remoting
- Data Management

Christophe Coenraets
<http://coenraets.org>

November, 20th 2008

Exercise 1: Setting Up Your Environment

Step 1: Install the LCDS Turnkey Server

http://www.adobe.com/go/trylivecycle_dataservices

The free LiveCycle Data Services ES Single-CPU License lets you run an application in a commercial, production environment on a single machine with no more than one CPU. This version is ideal for use in small-scale production applications and proof-of-concept projects.

Step 2: Install the workshop files

Unzip flex-dataservices-tutorial.zip on your local file system.

Step 3: Start the Turnkey Server

The LCDS Turnkey Server is a ready-to-use version of Tomcat in which the LiveCycle Data Services have already been deployed along with sample applications.

Note: The goal of the turnkey server is to give developers an easy way to run samples out-of-the-box. In your real-life development or production environment, you would typically integrate LCDS in your own Web Application on your own application server.

Note: To allow you to run the samples “out-of-the-box” without setting up a database, the turnkey server includes an HSQLDB database. HSQLDB is a lightweight Java RDBMS that is particularly well suited to run samples. We use this sample database in this workshop.

To start the samples database:

1. Open a command prompt
2. Navigate to c:\lcds\sampledb
3. Execute the following command:

```
startdb
```

To start the LCDS Turnkey Server:

1. Open a command prompt
2. Navigate to c:\lcds\tomcat\bin
3. Execute the following command:

```
catalina run
```

To test your environment:

1. Open a browser and access the following URL:

<http://localhost:8400/lcds-samples/testdrive-remoteobject/index.html>

2. Make sure the application is working properly

Step 4: Create the Java Project

You will need a Java project to work on the server-side of the applications built in this workshop. There are several ways you can set up Eclipse to work on the Java classes of a Web application. You can use a simple Java project, work with the Web Tools Platform (WTP), or other plugins like MyEclipse. In this workshop, to avoid dependencies on a specific plugin, we use a simple Java project.

1. In the Eclipse menu, select **File>New>Project**
2. Select **“Java Project”** in the project type tree and click Next
3. On the “Create a Java Project” page of the wizard:
 - Specify **“lcds-server”** as the project name
 - Check “Create project from existing source”, and point to C:\lcds\tomcat\webapps\lcds\WEB-INF
 - Click Next
4. On the “Java Settings” page, make sure that the “Default output folder” is **lcds-server/classes**, and click Finish.

This project configuration allows you to store the source code for your Java classes in WEB-INF\src directory. These classes will automatically be compiled in WEB-INF\classes.

Exercise 2: Creating a Chat Application

The LCDS Message Service provides a publish/subscribe infrastructure that allows your Flex application to publish messages and subscribe to a set of messaging destinations, enabling the development of real-time data push and collaborative applications.

In this section, you build a simple chat application that demonstrates the LCDS Message Service.

Step 1: Create the Messaging Destination

A messaging destination represents a topic of real time conversation that interested parties can subscribe (listen) to, or contribute to by posting their own messages.

To define the chat destination for this application:

1. In the **lcds-server** project, open **messaging-config.xml** located in the flex folder.
2. Add a destination called chat defined as follows:

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

This channel configuration indicates that the client will first try to connect to the message service using the my-rtmp channel. If a connection to the server cannot be established using that channel, the client will fall back to the my-polling-amf channel. Channels themselves are configured in services-config.xml.

3. Restart the server

Step 2: Create the Flex Project

1. Select **File>New>Project...** in the Eclipse menu.
2. Expand Flex Builder, select **Flex Project** and click Next.
3. Specify **chat** as the project name.
4. Keep the **use default location** checkbox checked.
5. Select **Web Application** as the application type.
6. Select **J2EE** as the application server type.
7. Check **use remote object access service**.
8. Uncheck Create combined Java/Flex project using WTP.
9. Click **Next**.
10. Make sure the root folder for LiveCycle Data Services matches the root folder of your LCDS web application. The settings should look similar to this (you may need to adjust the exact folder based on your own settings):

Root Folder: C:\lcds\tomcat\webapps\lcds

Root URL: <http://localhost:8400/lcds/>

Context Root: /lcds

11. Click **Validate Configuration**, then **Finish**.

Step 3: Create the Client Application

Open the chat.mxml file located in the src folder of the newly created chat project, and implement the application as follows:

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

  <mx:Script>
    <![CDATA[

      import mx.messaging.events.MessageEvent;
      import mx.messaging.messages.AsyncMessage;

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

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

    ]]>
  </mx:Script>

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

  <mx:Panel title="Chat" width="100%" height="100%">
    <mx:TextArea id="log" width="100%" height="100%"/>
    <mx:ControlBar>
      <mx:TextInput id="msg" width="100%" enter="send()"/>
      <mx:Button label="Send" click="send()"/>
    </mx:ControlBar>
  </mx:Panel>

</mx:Application>
```

Code highlights:

- At the client-side, the LCDS Message Service API provides two classes, Producer and Consumer, that you use to respectively publish and subscribe to a destination.
- To subscribe to a destination, you use the subscribe() method of the Consumer class.
- When a message is published to a destination you subscribed to, the **message** event is triggered on the Consumer.

- In this example, messages are published by Flex clients. The LCDS message service also provides a Java API that allows a server-side component to publish messages to a LCDS destination. Another option to exchange messages between Flex and Java applications is to map destinations to JMS topics, essentially allowing a Flex client to publish and subscribe to JMS topics.

Step 4: Test the Application

1. Run the application
2. Open the same URL in another browser window to open a second instance of the chat application
3. Type a message in one of the chat clients and click "Send": the message should appear in the two chat clients

Extra Credit

1. Add a simple TextInput to the user interface to allow the user to provide his/her name.
2. Add the user name in the message published to the chat destination.
3. Display the name of the author of the message along with the message itself in the chat TextArea.

Exercise 3: Creating a Collaborative Mapping Application

Step 1: Create the Flex Project

1. Select **File>New>Project...** in the Eclipse menu.
2. Expand Flex Builder, select **Flex Project** and click Next.
3. Specify **flexmaps** as the project name.
4. Keep the **use default location** checkbox checked.
5. Select **Web Application** as the application type.
6. Select **J2EE** as the application server type.
7. Check **use remote object access service**.
8. Uncheck Create combined Java/Flex project using WTP.
9. Click **Next**.
10. Make sure the root folder for LiveCycle Data Services matches the root folder of your LCDS web application. The settings should look similar to this (you may need to adjust the exact folder based on your own settings):

Root Folder: C:\lcds\tomcat\webapps\lcds
Root URL: <http://localhost:8400/lcds/>
Context Root: /lcds

11. Click **Validate Configuration**, then **Finish**.

Step 2: Add the Yahoo Map Library to the Project

Copy YahooMaps.swc from flex-dataservices-tutorial\YahooMap\YahooMap.swc to the lib directory of your newly created flexmaps project in Eclipse.

Step 3: Experiment with the Basic FlexMaps Application

1. Copy flexmaps.mxml from flex-dataservices-tutorial\flexmaps to the src directory of your newly created flexmaps project in Eclipse.
2. Open flexmaps.mxml in Eclipse and examine the application.
3. Run the application.
4. Drag the map and notice that, when you release the mouse button, a dialog displays the new map coordinates.
5. Type a valid address in the text input field, and click Search. The map moves to that new location, and the “New Coordinates” dialog pops up.

Step 4: Enable Collaboration in the FlexMaps Application

1. Open **messaging-config.xml** located in the flex folder of the lcds-server project, and add a destination called flexmaps defined as follows:

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

2. In flexmaps.mxml, define Producer and Consumer components as follows:

```
<mx:Producer id="producer" destination="flexmaps"/>
<mx:Consumer id="consumer" destination="flexmaps"
  message="messageHandler(event)"/>
```

3. In the init() function, add a line of code to subscribe to the flexmaps destination:

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

4. Replace the implementation of the handleNewCoordinates function with the following code:

```
var message:AsyncMessage = new AsyncMessage();
message.body.latitude = _yahooMap.centerLatLon.lat;
message.body.longitude = _yahooMap.centerLatLon.lon;
producer.send(message);
```

5. Create a messageHandler function defined as follows:

```
private function messageHandler(event:MessageEvent):void
{
  var body:Object = event.message.body;
  _yahooMap.centerLatLon = new LatLon(body.latitude, body.longitude);
}
```

6. Add the appropriate import statements

7. Test the application:

- a. Run the application.
- b. Access the same URL in another browser window to open a second instance of the flexmaps application.
- c. Move the map in one browser and notice that the position of the map is synchronized in the other browser.
- d. You can also look for an address in one browser: the map will move to that address in the two browsers.

Extra Credit

1. Add a chat box to the application to allow users to chat while collaborating on a map.
2. Add a zoom widget to the map and synchronize the zoom level between clients
 - To add a zoom widget:
`_yahooMap.addZoomWidget();`
 - To listen to zoom changes:
`_yahooMap.addEventListener(YahooMapEvent.MAP_ZOOM, yourEventHandler);`
 - To get or set the zoom level, use the `_yahooMap.zoomLevel` property
3. Add a map type widget to the map and synchronize the map type between clients
 - To add a map type widget:
`_yahooMap.addTypeWidget();`
 - To listen to map type changes:
`_yahooMap.addEventListener(YahooMapEvent.MAP_TYPE_CHANGED,
yourEventHandler);`
 - To get or set the map type, use the `_yahooMap.mapType` property

Exercise 4: Creating a Simple Contact Management Application

Using the Remoting Service, you can directly invoke methods of Java objects deployed in your application server, and consume the return value. The return value can be a value of a primitive data type, an object, a collection of objects, an object graph, etc. Java objects returned by server-side methods are deserialized into either dynamic or typed ActionScript objects.

In this section, we build a simple contact management application that demonstrates the LCDS Remoting service.

Step 1: Copy and Examine the Java Classes

1. Copy **hsqldb.jar** from flex-dataservices-tutorial\hsqldb to the lib directory of the **lcds-server** project
2. Copy the **insync directory** and the **insync.properties** file from flex-dataservices-tutorial\contact\javasrc to the src directory of your **lcds-server** project
3. Open and examine the **Contact** class (in the insync.model package) and the **ContactDAO** class (in the insync.dao package)

The server-side of the insync application uses the simple Value Object and Data Access Object (DAO) patterns:

- Contacts are transferred between the client and the server using **Contact** value objects.
- The **ContactDAO** class provides the data access logic to retrieve and update contacts

Step 2: Create the Remoting Destination

A Remoting destination exposes a Java class that your Flex application can invoke remotely. The destination id is a logical name that your Flex application uses to refer to the remote class, avoiding a hardcoded reference to the fully qualified Java class name. This logical name is mapped to the Java class name as part of the destination configuration in remoting-config.xml.

To create a remoting destination for the ContactDAO class:

1. Open remoting-config.xml located in the flex folder of the lcds-server project.
2. Add a destination called contact defined as follows:

```
<destination id="contact">
  <properties>
    <source>insync.dao.ContactDAO</source>
  </properties>
</destination>
```

Step 3: Create the Flex Project

1. Select **File>New>Project...** in the Eclipse menu.
2. Expand Flex Builder, select **Flex Project** and click Next.
3. Specify **insync** as the project name.
4. Keep the **use default location** checkbox checked.
5. Select **Web Application** as the application type.
6. Select **J2EE** as the application server type.
7. Check **use remote object access service**.
8. Uncheck Create combined Java/Flex project using WTP.
9. Click **Next**.
10. Make sure the root folder for LiveCycle Data Services matches the root folder of your LCDS web application. The settings should look similar to this (you may need to adjust the exact folder based on your own settings):

Root Folder: C:\lcds\tomcat\webapps\lcds
Root URL: <http://localhost:8400/lcds/>
Context Root: /lcds

11. Click **Validate Configuration**, then **Finish**.

Step 4: Code the Application

1. Open insync.mxml located in the src folder of the insync project, and implement the application as follows:

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

    <mx:RemoteObject id="ro" destination="contact"/>

    <mx:ApplicationControlBar width="100%">
        <mx:TextInput id="searchStr"/>
        <mx:Button label="Search"
            click="ro.getContactsByName(searchStr.text)"/>
    </mx:ApplicationControlBar>

    <mx:DataGrid id="dg"
        dataProvider="{ro.getContactsByName.lastResult}"
        width="100%" height="100%"/>

</mx:Application>
```

2. Run and test the application
 - Type a few characters in the search `TextInput` field and click Search
 - To retrieve all the contacts, leave the `TextInput` field empty and click Search

Exercise 5: Using the RemoteObject Events

RemoteObject calls are asynchronous. You use the **result** and **fault** events of the RemoteObject component to handle results and errors.

1. To make the application more robust and better partitioned, modify the code as follows:

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

  <mx:Script>

    <![CDATA[

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

      [Bindable]
      private var contacts:ArrayCollection;

      private function resultHandler(event:ResultEvent):void
      {
        contacts = event.result as ArrayCollection;
      }

      private function faultHandler(event:FaultEvent):void
      {
        Alert.show(event.fault.faultDetail);
      }

    ]]>

  </mx:Script>

  <mx:RemoteObject id="ro" destination="contact"
    fault="faultHandler(event)">
    <mx:method name="getContactsByName"
      result="resultHandler(event)"/>
  </mx:RemoteObject>

  <mx:ApplicationControlBar width="100%">
    <mx:TextInput id="searchStr"/>
    <mx:Button label="Search"
      click="ro.getContactsByName(searchStr.text)"/>
  </mx:ApplicationControlBar>

  <mx:DataGrid id="dg" dataProvider="{contacts}"
    width="100%" height="100%"/>

</mx:Application>
```

2. Run and test the application

Exercise 6: Updating Contacts

Step 1: Create the Contact Value Object

In the application so far, the list of contacts returned by the `getContactsByName()` method is deserialized into dynamic objects. It is sometimes desirable to work with strongly typed objects. To work with typed objects in this application, let's create the ActionScript version of the Contact class defined in the lcms-server project:

1. Right-click the src folder in the insync project and select **New>ActionScript Class**. Specify **Contact** as the class name and click Finish.
2. Implement the Contact class as follows:

```
package
{
    [Bindable]
    [RemoteClass(alias="insync.model.Contact")]
    public class Contact
    {
        public var id:int;
        public var firstName:String;
        public var lastName:String;
        public var email:String;
        public var phone:String;
        public var address:String;
        public var city:String;
        public var state:String;
        public var zip:String;
    }
}
```

Notice that we use the `[RemoteClass(alias="insync.model.Contact")]` annotation to map the ActionScript version of the Contact class (`Contact.as`) to the Java version (`Contact.java`). As a result, Contact objects returned by the `getContactsByName()` method of `ContactDAO` are deserialized into instances of the ActionScript Contact class. Similarly, in the Contact form (see next step), the instance of the ActionScript Contact class passed as an argument to the `save` method of the `RemoteObject` is deserialized into an instance of the java version of the Contact class at the server-side.

Step 2: Add a Contact Form

1. Copy ContactForm.mxml from flex-dataservices-tutorial\insync\flexsrc to the src directory of the insync project in Eclipse.
2. Examine the ContactForm.mxml source code.
3. Modify the insync.mxml application to allow the user to update contacts in the Contact Form. Wrap the DataGrid in an HDividedBox, add and wire the ContactForm component as follows:

```
<mx:HDividedBox width="100%" height="100%">
  <mx:DataGrid id="dg"
    dataProvider="{contacts}" width="30%" height="100%">
    <mx:columns>
      <mx:DataGridColumn
        dataField="firstName" headerText="First Name"/>
      <mx:DataGridColumn
        dataField="lastName" headerText="Last Name"/>
    </mx:columns>
  </mx:DataGrid>
  <ContactForm contact="{dg.selectedItem as Contact}" width="70%"/>
</mx:HDividedBox>
```

Step 3: Test the Application

1. Run the application
2. Select a contact in the DataGrid, modify some data in the Contact Form, and click the Save button to save your changes.
3. Select a contact in the DataGrid, and click the Delete button to delete the contact.

Extra Credit

Implement a strategy to enable the Save button only when data has been modified

Exercise 7: Opening Multiple Contacts

1. In `insync.mxml`, replace the `ContactForm` component with a `TabNavigator` defined as follows:

```
<mx:TabNavigator id="tn" width="70%" height="100%" />
```

2. Add an `openContact` function defined as follows:

```
public function openContact(contact:Contact):void
{
    var form:ContactForm = new ContactForm();
    tn.addChild(form);
    form.contact = contact;
    tn.selectedChild = form;
}
```

3. Enable double-clicking in the `DataGrid` by adding the following attributes to the `DataGrid` definition:

```
doubleClickEnabled="true"
doubleClick="openContact(dg.selectedItem as Contact)"
```

4. In `ContactForm.mxml`, specify the `label` property of the `Canvas` as follows:

```
<mx:Canvas xmlns:mx="http://www.adobe.com/2006/mxml"
width="100%" height="100%"
label="{contact.firstName + ' ' + contact.lastName}">
```

5. In `ContactForm.mxml`, replace the implementation of the `deleteItem_resultHandler` function with the following implementation:

```
parent.removeChild(this);
```

6. In the `HBox` at the bottom of `ContactForm.mxml`, add a `Close` button defined as follows:

```
<mx:Button label="Close" click="parent.removeChild(this)" />
```

7. Test the application
 - a. Run the application.
 - b. Double-click a contact in the `DataGrid` to open a form and edit this contact in the `TabNavigator`
 - c. Double-click another contact to open a second contact in the `TabNavigator`.

Extra Credit

Add some logic to the `openContact` function to prevent the same contact from being opened multiple times in the `TabNavigator`. If the user tries to open a contact that is already open, simply activate the corresponding tab in the `TabNavigator`.

Exercise 8: Adding Contacts

1. Add a Button to the ApplicationControlBar in insync.mxml defined as follows:

```
<mx:Button label="New Contact" click="openContact(new Contact())"/>
```

2. In ContactForm.mxml, change the label of the Canvas as follows:

```
label="{contact.id>0?contact.firstName+' '+contact.lastName:'New Contact'}"
```

3. Replace the implementation of the save_resultHandler function with the following implementation:

```
contact.id = event.result.id;
```

4. Test the application

Exercise 9: Creating a Simple Data Management Application

Step 1: Examine the Assembler Class

Using the Data Management Service, changes made to the data at the client-side are automatically sent to a data service running in the application server. The data service then passes those changes to an object called the **assembler**. The role of the assembler is to pass the changes, in the appropriate format, to your existing business objects, or directly to your persistence layer. The assembler class is the only class that you have to write at the server-side (in addition to your existing business and persistence layer), and is typically a very simple class.

1. Make sure Eclipse is in the Java perspective
2. Open ContactAssembler.java in the insync.assembler package of the lcds-server project

Code highlights:

- ContactAssembler extends AbstractAssembler and implements a series of callback methods that the data service invokes when it gets changes from the client application.
- You use the assembler to plug in to your existing business objects or directly to your persistence layer.
- In this simple implementation, we invoke the method corresponding to the type of change in our ContactDAO class.

Step 2: Define the Data Management Destination

1. Open data-management-config.xml in the flex folder of the lcds-server project
2. Add the following destination:

```
<destination id="insync.contact">
  <adapter ref="java-dao"/>
  <properties>
    <source>insync.assembler.ContactAssembler</source>
    <scope>application</scope>
    <metadata>
      <identity property="id" undefined-value="0"/>
    </metadata>
  </properties>
  <channels>
    <channel ref="my-rtmp"/>
  </channels>
</destination>
```

This XML fragment maps the logical name for the destination (insync.contact) to an assembler class (insync.assembler.ContactAssembler).

3. Make sure you save data-management-config.xml, and restart the server

Step 3: Create the Flex Project

1. Select **File>New>Project...** in the Eclipse menu.
2. Expand Flex Builder, select **Flex Project** and click Next.
3. Specify **simplecontact** as the project name.
4. Keep the **use default location** checkbox checked.
5. Select **Web Application** as the application type.
6. Select **J2EE** as the application server type.
7. Check **use remote object access service**.
8. Uncheck Create combined Java/Flex project using WTP.
9. Click **Next**.
10. Make sure the root folder for LiveCycle Data Services matches the root folder of your LCDS web application. The settings should look similar to this (you may need to adjust the exact folder based on your own settings):

Root Folder: C:\lcds\tomcat\webapps\lcds

Root URL: <http://localhost:8400/lcds/>

Context Root: /lcds

11. Click **Validate Configuration**, then **Finish**.

Step 4: Define the Contact Value Object

1. Copy Contact.as from the insync project to the src folder of the simplecontact project
2. Open Contact.as **in the simplecontact project**
3. Replace the [Bindable] annotation with the [Managed] annotation

The [Managed] annotation indicates that the object should be automatically managed: the object will automatically trigger events when its properties are changed.

Step 5: Code the application

1. Open simplecontact.mxml and implement the application as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*"
    applicationComplete="ds.fill(contacts)">

    <mx:DataService id="ds" destination="insync.contact"/>
    <mx:ArrayCollection id="contacts"/>
    <Contact/>

    <mx:DataGrid id="dg" dataProvider="{contacts}" editable="true"
        width="100%" height="100%"/>

    <mx:Button label="Delete"
        click="contacts.removeItemAt(dg.selectedIndex)"/>

</mx:Application>
```

Code highlights:

- The DataService points to the "insync.contact" destination defined earlier in data-management-config.xml.
 - In the Application tag's applicationComplete event, the DataService's fill() method populates the "contacts" ArrayCollection. When you invoke fill() at the client-side, the assembler's fill() method is invoked at the server-side. This is where you specify how the data should actually be retrieved. See the fill() method in ContactAssembler.java.
 - The DataGrid is bound to the contacts array to display the list of contacts.
2. Test the application
 - a. Run the application
 - b. Change some data for a few contacts (do NOT change the value of the id column)
 - c. Reload the application and notice that your changes have been persisted
 3. Test the automatic client synchronization feature of the application
 - a. Open a second browser and access the same URL
 - b. Modify some data in one browser, and notice that the changes automatically appear in the second browser.

Note: Automatically pushing the changes made by one client to the other clients working on the same data is the default behavior of LCDS. We will see how this default behavior can be changed later in this workshop.

Step 6: Control when Changes are Sent to the Server

By default, LCDS sends a change message to the server immediately after you change a property of an object. This behavior is appropriate for some applications (for example, a collaborative form-filling application), but might not be desirable in other applications, because:

- It generates too much network traffic
- There may be dependencies between object attributes, and the server may not be able to persist partially filled objects

For these reasons, you often want to programmatically control when the changes are sent to the server.

1. To programmatically control when the changes are sent to the server:
 - a. In `simplecontact.mxml`, add `autoCommit="false"` to the `DataService` declaration
 - b. Add an “Apply Changes” button defined as follows:

```
<mx:Button label="Apply Changes" click="ds.commit()"
  enabled="{ds.commitRequired}"/>
```

2. Now that we have control over when the changes are sent to the server, we can also handle the creation of new contacts. Add a “New” Button defined as follows:

```
<mx:Button label="New" click="contacts.addItem(new Contact())"/>
```

Note: This logic for creating new contacts would have failed with `autoCommit` set to `true` (default) because the server-side component would have tried to insert an empty row and some columns of the contact table are defined as not supporting null values.

3. Test the application
 - a. Run the application
 - b. Change some data for a few contacts
 - c. Reload the application and notice that your changes have not been persisted
 - d. Change some data again and click the Apply Changes button
 - e. Reload the application and notice that your changes have now been persisted
4. Test the automatic client synchronization feature of the application
 - a. Open a second browser and access the same URL
 - b. Modify some data in one session, and notice that, with `autoCommit` set to `false`, the changes are not pushed to the second browser session until you click the “Apply Changes” button.

Exercise 10: Using Data Management in the inSync Application

1. In Contact.as, replace the [Bindable] annotation with the [Managed] annotation.
2. In insync.mxml, replace the RemoteObject with a DataService defined as follows:

```
<mx:DataService id="ds" destination="insync.contact"
    fault="faultHandler(event)" autoCommit="false"/>
```

3. Modify the declaration of the contacts variable as follows:

```
private var contacts:ArrayCollection = new ArrayCollection();
```

4. Modify the implementation of the Search button as follows:

```
<mx:Button label="Search"
    click="ds.fill(contacts, 'by-name', searchStr.text)"/>
```

5. In ContactForm.mxml, replace the RemoteObject with a DataService defined as follows:

```
<mx:DataService id="ds" destination="insync.contact"
    fault="faultHandler(event)" autoCommit="false"/>
```

6. In the save() function, replace ro.save() with the following code:

```
if (contact.id == 0)
{
    ds.createItem(contact);
}
ds.commit();
```

7. Replace the implementation of the deleteItem function with the following code:

```
ds.deleteItem(contact);
var token:AsyncToken = ds.commit();
token.addResponder(
    new AsyncResponder(deleteItem_resultHandler, faultHandler));
```

8. Add the appropriate import statements
9. Test the application

Exercise 11: Resolving Conflicts

Conflicts can occur when different clients are trying to update the same data at the same time. The number of conflicts in an application depends on the locking strategy (concurrency level) implemented in the application.

For example, open `ContactDAO.java` and examine the difference between the `update(Contact contact)` and the `update(Contact contact, Contact originalContact)` methods:

- In the `update(Contact contact)` method, the `WHERE` clause is built using the primary key only. As a result, your update will succeed even if the row has been changed by someone else since you read it.
- In the `update(Contact contact, Contact originalContact)` method, the `WHERE` clause is built using the primary key and by verifying that each column still has the same value it had when you read the row. Your update will fail if the row has been changed by someone else since you read it.

Note: Another way of implementing the same concurrency level without adding all the columns to the `WHERE` clause is to add a “version” column to the table. Every time the contact is updated, the contact’s version is incremented. Using this approach, you can make sure a row has not been updated since you read it by building the `WHERE` clause using the primary key and the version column only.

The level of concurrency you choose depends on your application. LCDS doesn’t force you to implement locking in any particular way, and doesn’t even attempt to identify when a conflict has occurred. You are free to define what represents a conflict in the context of your application. You tell LCDS when a conflict has occurred by throwing a `DataSyncException`. For example, open `ContactAssembler.java` and examine the `updateItem()` method. Notice that the method throws a `DataSyncException` when the update fails in the DAO (no contact matched the `WHERE` clause).

At the client-side, LCDS provides a sophisticated conflict resolution API. The `DataService`’s “conflict” event is triggered when a conflict occurs. The conflict resolution API provides several options to handle the conflict as appropriate in the event handler.

In this section, we will use the `update(Contact contact, Contact originalContact)` method of `ContactDAO` because it makes it easier to generate conflicts and thus experiment with conflict resolution.

1. Open ContactAssembler.java. In the updateItem() method, replace dao.update((Contact) newVersion) with dao.update((Contact) newVersion, (Contact) prevVersion). Recompile ContactAssembler.java and restart your application server.
2. In simplecontact.mxml, add a conflictHandler() function defined as follows:

```
private function conflictHandler(event:DataConflictEvent):void
{
    Alert.show("The current state of the contact has been reloaded",
               "Conflict");
    var conflicts:Conflicts = ds.conflicts;
    var conflict:Conflict;
    for (var i:int=0; i<conflicts.length; i++)
    {
        conflict = conflicts.getItemAt(i) as Conflict;
        if (!conflict.resolved)
        {
            conflict.acceptServer();
        }
    }
}
```

Note: acceptServer() represents one approach to resolve conflicts. Refer to the documentation to explore the other options.

3. Modify the DataService declaration as follows:

```
<mx:DataService id="ds" destination="insync.contact"
    autoCommit="false"
    autoSyncEnabled="false"
    conflict="conflictHandler(event)"/>
```

4. Add the appropriate import statements
5. Test the application
 - a. Run the application in two browsers
 - b. Modify the last name of a contact in one browser, and click “Apply Changes”
 - c. Modify the email address of the same contact in the other browser and click “Apply Changes”

Based on how we wrote the persistence logic at the server side, this is considered a conflict. The client application uses the conflict resolution API to revert to the current server value of the object.