# Introduction

# Separated UI and Services development

# Setup

## Installing Flex Builder 3

1. Browse to the URL http://www.adobe.com/go/tryflex.
2. Download Flex Builder 3 Professional for Windows

*Note: You must register with a free Adobe user id to download software.*

1. Install Flex Builder 3 with all default options.

If prompted for a serial number, leave it blank if you don’t have one available.

Flex Builder will install as a 60-day evaluation.

## Installing LiveCycle Data Services ES

1. Browse to the URL http://www.adobe.com/go/trylivecycle\_dataservices.
2. Download LiveCycle Data Services ES for Windows and save the installation file, **lcds26-win.exe**, to any location on your disk.
3. Install LiveCycle Data Services ES by running **lcds26-win.exe**.
4. At the **Serial Number** screen, leave the serial number blank to use the **Single CPU License** version of Live Cycle Data Services ES.
5. At the **Installation Location** screen, accept the default installation path of ***c:\lcds***.
6. At the **Installer Options** screen, select ***LiveCycle Data Services with Tomcat***.
7. Complete the installation.

## Viewing the LCDS Examples (Very Useful!)

1. From the Windows Start menu, select All Programs > Adobe > LiveCycle Data Services 2.6.1 ES > Start Samples Database
2. From the Windows Start menu, select All Programs > Adobe > LiveCycle Data Services 2.6.1 ES > Start LiveCycle Data Services Server to start the Tomcat server.
3. Navigate to <http://localhost:8400/lcds-samples>

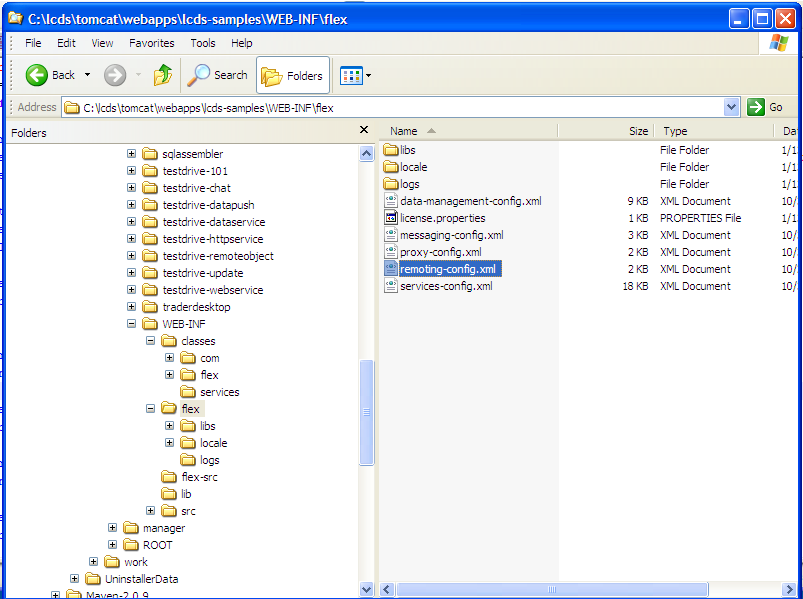
## (1 point - A priority) How to write a generic LCDS enabled Java service

There are three steps to using a Java based RPC service. (1) Creating the Java service itself, in this case from a POJO project in eclipse, (2) configuring the remoting endpoint and (3) Creating the Flex project to consume the service.

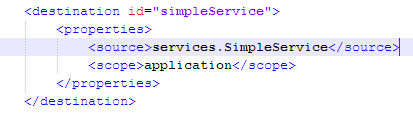
For this example I’ll be leveraging the /lcds sample application that ships with LCDS. If you have done the default LCDS installation you can see this application running by navigating to [http://localhost:8400/lcds](http://localhost:8080/lcds)

1. Create a Java project (the Java Service)
   1. Using Eclipse 3.4, click new🡪Project🡪Java called POJOProject
   2. Create package “services”
   3. Create a new Java class “SimpleService.java”
   4. Code is attached for SimpleService.java
   5. Compile the SimpleService to SimpleService.class
   6. Copy the services directory containing your class file to C:\lcds\tomcat\webapps\lcds\WEB-INF\classes
   7. You could also use ANT to package up the Java service into a JAR file and deploy it under C:\lcds\tomcat\webapps\lcds\WEB-INF\libs. This would be a more appropriate solution for a more complex service.
   8. The LCDS-Samples application available at <http://localhost:8400/lcds-samples> contain other great examples, as does Christophe Coenraets “Test Drive” application http://coenraets.org/blog/2009/05/new-update-to-the-spring-blazeds-integration-test-drive/
2. In order to create a remoting endpoint capable of being consumed by flex, we need to make the following change
   1. Destinations in the remoting-config.xml file specify remote services that can be called from flex, therefore we need to add the destination to C:\lcds\tomcat\webapps\lcds\WEB-INF\flex\remoting-config, as follows (after the </default-channels>

Note that C:\lcds\tomcat\webapps\lcds is a template you can use for creating your own LCDS applications

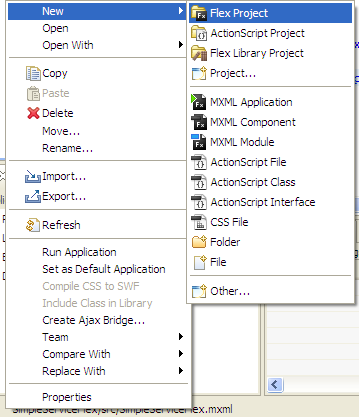


remoting-config.xml

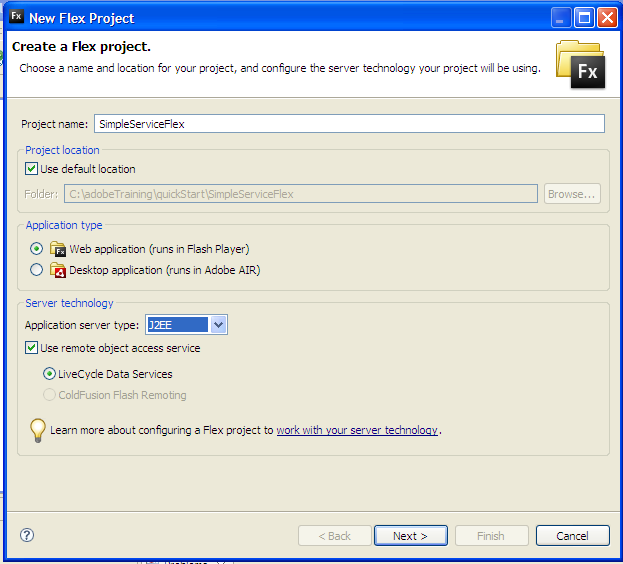


* 1. Once the java remote service is defined in remoting-config.xml we can proceed to creating the flex code that can call these services.

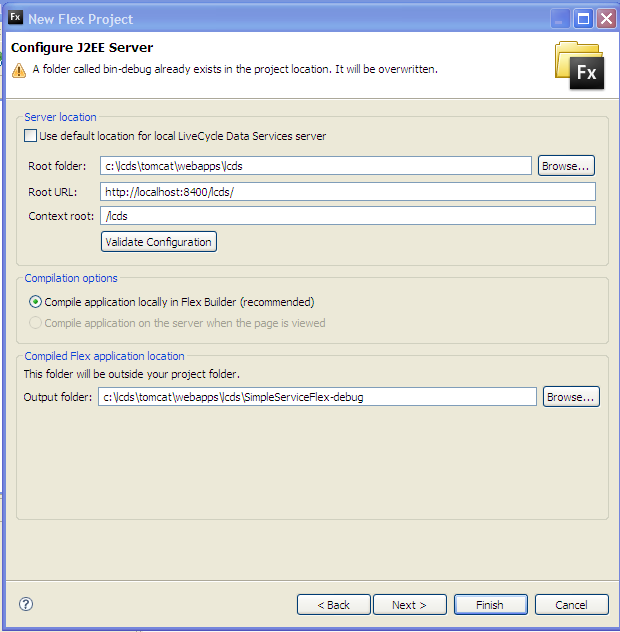
1. Create the Flex Application
   1. Click New🡪Flex Project within Flex Builder



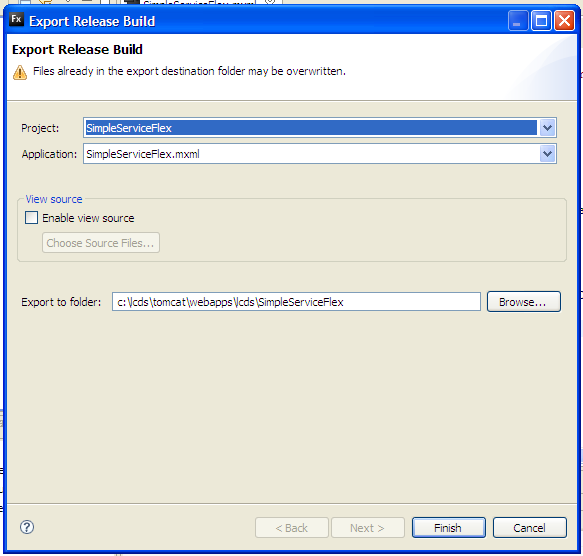
* 1. Name the Project and choose “J2EE” s the application server type



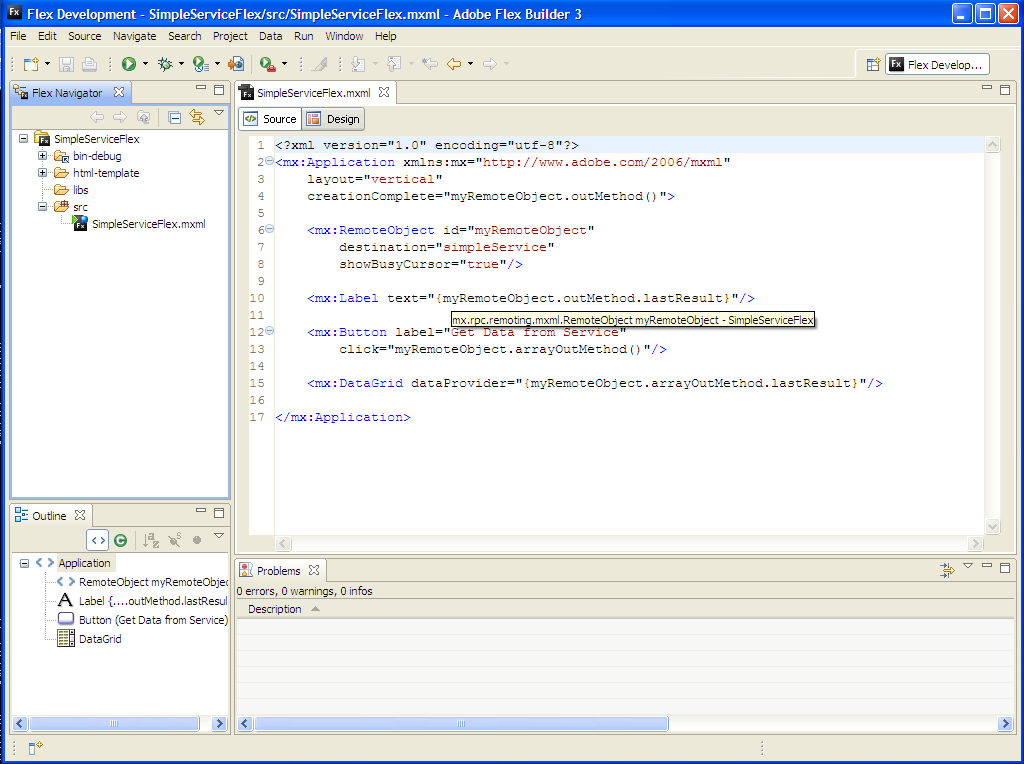
* 1. Point the application you’re creating to the LCDS application on the default Tomcat Server



* 1. The MXML for SimpleServiceFlex is attached
  2. You can also export a “release build” to the Tomcat server by clicking “project🡪export release build”



* 1. The view of the application



* 1. Run the Application



## (2 points - A priority) How to write a Flex / AIR UI that can connect to n different services

Step by step to consume an RPC service from the client. The following is a setep by step explanation at creating the files that are included, and deployed above.

1. Define mx:RemoteObject consuming the created Java service

With the destination defined in the remoting-config.xml file, we can call the Java class’s methods from our Flex application using the <Mx:RemoteObject> tag

|  |  |
| --- | --- |
| **Property** | **Purpose** |
| id | Instance name of the object |
| destination | Service ID destintation, matching the one defined in remoting-config.xml |
| result | The event handler for the result |
| fault | The event handler for the fault |

<mx:RemoteObject

id="roService" destination="roDestination"

1. To invoke the service simply call the Java class’ public methods in either system events or user events
   1. Example 1 – User Event – the click of a button

<mx:Button label="get Data"

click="roService.getData()" />

* 1. Example 2 – System Event – creationComplete of the application

<mx:Application

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

layout="absolute"

creationComplete="roService.getData();">

1. To directly access data returned in a binding expression, use the lastResult property of the method call object. In the following example, the result is bound to a DataGrid control.

<mx:DataGrid dataProvider="{roService.getData.lastResult}"/>

1. The ResultObject’s result event occurs on successful completion of the remote call. Object and event listeners must be declared to handle the event. Note that in these examples, “event” is an instance of mx.rpc.events.ResultEvent. This Object’s result property refers to the returned data and must be cast to the right datatype since the result object can be of any type.
   1. In MXML

<mx:RemoteObject

id="roService" destination="roDestination"

**result="myResultHandler(event)"**/>

* 1. In ActionScript

**import** mx.rpc.events.ResultEvent;

**import** mx.rpc.remoting.RemoteObject;

**private** **var** myRemoteObject:RemoteObject = **new** RemoteObject(**"roService"**);

**private** **function** init():**void**{

myRemoteObject.addEventListener(ResultEvent.RESULT, resultHandler);

}

* 1. The Result Handler

[**Bindable**]

**private** **var** data:ArrayCollection;

**private** **function** resultHandler(event:ResultEvent):**void** {

data=event.result **as** ArrayCollection;

}

In (c.) above, we’re casting the result to an ArrayCollection (this would be the cast when the Java class returns a List of objects)

1. A fault event occurs when an error is triggered (on the client or server). “fault” is an instance of mx.rpc.events.FaultEvent, which has the following properties

|  |  |
| --- | --- |
| **Property** | **Purpose** |
| faultCode:String | Simple code describing the fault |
| faultDetail:String | Extra details (if any) on the fault |
| faultString | The text description of the fault |
| rootCause | The cause of the fault |

As with the result event, handlers must be created to handle the fault

* 1. In MXML

<mx:RemoteObject

id="roService" destination="roDestination"

**fault="myfaultHandler(event)"**/>

* 1. In ActionScript

**private** **function** init():**void**{

myRemoteObject.addEventListener(ResultEvent.FAULT, faultHandler);

}

* 1. The Result Handler

[**Bindable**]

**import** mx.rpc.events.FaultEvent;

**import** mx.controls.Alert;

**private** **function** faultHandler(event:FaultEvent):**void** {

Alert.show (event.fault.faultString,event.fault.faultCode);

}

1. Client and Server Debugging Techniques
   1. Flex Builder Debugger

Flex builder has a built in debugger which may be used to examine the resulting data from a remote call. To activate the debugger to examine the result object, place a breakpoint on the closing brace of the result event handler, and debug the application.

Navigate in the variables view to event🡪result🡪 to see the returned data

* 1. Flex Builder Network Monitor

<TBD>

* 1. Charles

<TBD>

Note: Best practices about architecting the app won’t be considered.

Note: Dynamic end-points won’t be used here. Flex project will be compiled using compiler arguments pointing to LCDS. This way we can introduce the problem we want to solve in the next point

dson