

4D For Flex: Rich and Easy Web Data

Presented by: **Tim Kaufman - 4D, Inc.**

INTRODUCTION

This is an introductory look at both Adobe's Flex platform, and the new addition to 4D's Web 2.0 Pack, 4D For Flex. It is a basic overview of some of the most important features. Adobe's Flex Builder software is very helpful to have, but most of the examples are in just code, not design view.

RICH INTERNET APPLICATIONS (RIA)

Rich Internet applications are web applications that have some of the function and feel of desktop applications. Until the advent of RIA techniques, web applications would rely on the server for most of the processing, wherein a server request is made, and a complete HTML page is returned in response, which must be reloaded in its entirety. This can involve excessive overhead, for example when just one small part of a web page needs to refresh the data that it presents.

RIAs take advantage of the processing power of scripting languages made available in the browser, the transfer of data via a lightweight but self-contained form such as XML, and asynchronous communication between client and server (different requests being handled concurrently). The browser can make requests of the server independently of how the user is interacting with the page. Data can move between client and server without a full page request. A richer user experience can be provided with features such as drag and drop and interactive controls.

Ajax vs. Flex

These are the two best-known front-end technology alternatives for developing RIAs. Ajax has the advantage of using mostly already familiar standards, such as HTTP, HTML, JavaScript, and XML. This leverages existing developer knowledge. It can have issues with browser compatibility though, since the browser provides the runtime environment on the client side.

Adobe Flex is a programming framework for applications running on the Flash Player platform. Because Flash is a plug-in provided by one company (Adobe), there are fewer problems with compatibility. Because it provides its own custom user interface components, a richer interface can be developed more easily than with HTML/JavaScript. But it does require learning less familiar languages, MXML and ActionScript; and it depends on the client having an up-to-date version of the Flash Player plug-in.

Flex Platform and Tools

The easiest way to develop Flex applications is with the Adobe Flex Builder software, which is based on the open source Eclipse, but must be purchased from Adobe. Flex Builder provides a complete integrated development environment. However, the Flex SDK (Software Development Kit) can be downloaded for free, and in conjunction with a good text editor and the command line, used to build the same Flex applications, albeit without the convenient tools and views of Flex Builder.

ADOBE FLEX BASICS

Flex was implemented as an ActionScript class library. The library provides the Flex developer with components (containers and controls), manager classes, data-service classes, and classes for all other features. The developer writes applications by using the MXML and ActionScript programming languages with the classes provided by the library.

You can create an MXML application in one file or in many files. Normally you would have a main file that has the `<mx:Application>` tag. The main file can reference additional files coded in MXML, ActionScript, or a mix of the two.

The MXML Language

MXML is an XML-based markup language that can generate an interface layout with some simple declarative statements. In this way it is something like using HTML, but with a richer set of layout elements. MXML can also be used to define non-visual aspects of an application, such as access to server-side data sources and data bindings between user-interface components and data sources.

Since an MXML file is ordinary XML, you can easily work with it in a number of text editors and other programming environments. Here for example is a simple MXML file to put a button (which does nothing) on the screen.

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

  <mx:Button id="button1" label="Hello Button" />
</mx:Application>
```

The first line of the MXML file is the XML declaration. This line has to be the first line in each MXML file.

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

The next line is the `<mx:Application>` tag, which defines the Application container that is always the root tag of a Flex application. It is the all-encompassing container, and the application's entry point. For this reason, even though you may have multiple MXML files or components, there can be only one Application container for the application. The other MXML files will have some other container as the root, such as Canvas or Hbox, or perhaps not use containers at all.

The `xmlns:mx` attribute is defining `mx` as a namespace. A namespace is a grouping for Flex library elements, to help avoid naming collisions. The `mx` prefix maps each of the components in the `mx` namespace to its fully qualified class name. It is the base namespace for the Flex libraries. You can define your own custom namespace for your libraries; for instance, here is a "fourD" namespace declaration:

```
xmlns:fourD="http://www.4d.com/2007/mxml"
```

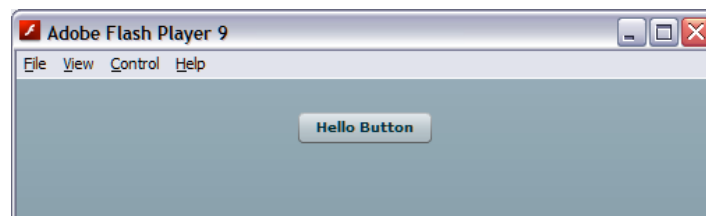
Compiling and Running

After you write a Flex application, you must compile it using the Flex compiler. If you are using Flex Builder, create a new Flex Project, create an MXML file with this code, and click "run".

If you don't have Flex Builder, the Flex compiler is a small executable file called `mxmcl` in the `bin` folder of the Flex SDK. The `PATH` environment variable can be set to include the path of `mxmcl`, or you can simply spell out the path when compiling. On a Windows system with Flex Builder 3 installed, this is how the compile command was made in the command prompt:

```
> c:\Program Files\Adobe\Flex Builder 3\sdk\3.0.0\bin\mxmcl Button1.mxml
```

The result of compilation is a Flash file called `Button1.swf`. Double-clicking it will run it in the Flash Player:



If the "Generate HTML wrapper file" compiler option is selected, Flex Builder will automatically create an HTML wrapper page (called "Button1.html" in this case) with a reference to the SWF

file. The wrapper uses JavaScript to check for the presence of an appropriate version of the Flash Player plugin, and embeds the Flex application's SWF file in the web page. It also allows the enabling of browser-oriented functionality like deep linking and back button support.

While MXML uses the familiar declarative style of XML, what may not be so apparent is the object-oriented aspect of the Button declaration from above:

```
<mx:Button id="button1" label="Hello Button" />
```

This simple bit of code is instantiating an object of the `mx:Button` class. Further, the attributes such as “id” and “label” are properties of the object. Why this is will become clear in a moment.

The ActionScript Language

ActionScript is a programming language executed by the ActionScript Virtual Machine (AVM) built into the Flash Player, and is similar to JavaScript. In fact, both languages are conforming to ECMA standards. ActionScript 3.0 is based on ECMAScript 4, as is JavaScript 2.0, so when JavaScript 2.0 is more widely adopted, they will be virtually the same. ActionScript 3.0 has strong variable typing, object-oriented classes, and packages and namespaces.

Although we speak of MXML and ActionScript as if they are two separate entities, actually MXML files are converted behind the scenes into ActionScript classes. MXML tags correspond to ActionScript classes or class properties. When the Flex application is built, Flex parses the MXML tags and generates the corresponding ActionScript classes. Then the ActionScript classes are compiled into SWF binary bytecode that is saved in an SWF file.

So MXML turns out to be a nice convenience language, which makes it unnecessary to code out everything in ActionScript, which can be quite lengthy and tedious for just routine items like user interface components. Take the button we coded in MXML above. The compiler converts that to ActionScript for us; but we could code it directly in ActionScript.

Note that ActionScript code here is placed inside the `<mx:Script>` tags in an MXML file, but we could have put it in a separate file with a `.as` extension. Also such code in an MXML file needs to be inside CDATA tags, i.e. `<![CDATA[...]]>`, so the MXML parser will not try to parse the script as XML. This is similar to how JavaScript in an HTML file was often surrounded by HTML comments so older browser would not try to interpret script as HTML. The executed result of this example displays the identical button the MXML version displayed.

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

  creationComplete="creationCompleteHandler();"
  width="300" height="80"
>

  <mx:Script>
    <![CDATA[
      import mx.controls.Button;
      import mx.events.FlexEvent;

      private var button1:Button;

      private function creationCompleteHandler():void
      {
        // Create a Button instance and set its label
        button1 = new Button();
        button1.label = "Hello Button";

        // Get notified once button component has been
        // created and processed for layout
      }
    ]]>
  </mx:Script>
</mx:Application>
```

```

        myButton.addEventListener (FlexEvent.CREATION_COMPLETE,
                                   buttonCreationCompleteHandler);

        // Add the Button instance to the DisplayList
        addChild (myButton);
    }

    private function buttonCreationCompleteHandler (
        evt:FlexEvent ):void
    {
        // Center the button
        myButton.x = parent.width/2 - myButton.width/2;
        myButton.y = parent.height/2 - myButton.height/2;
    }

    ]]>
</mx:Script>
</mx:Application>

```

As you can see, we had to handle a lot of details that were taken care of for us in MXML, most notably handling events. But the important thing to remember here is that every MXML tag attribute corresponds to some property, style or event listener of the ActionScript object.

So while you could get the impression that MXML is simply for defining interface objects, and ActionScript for event handling and application logic, actually both languages describe the same objects via different syntax. MXML has the convenience of far fewer lines of code, but ActionScript is required for more complex programming tasks.

So ActionScript is a strongly typed language with syntax similar to JavaScript, and many of the object-oriented and other advanced features of languages like Java or C#.

Flex COMPONENTS: Containers and Controls

A container is a component that defines a particular block of the Flash Player's drawing screen, and can contain controls and other nested containers, which are then referred to as child components. The different kinds of containers provide different kinds of layout flow and style. Controls are the components users interact with in the interface, such as Button, TextInput, and ComboBox.

Here is an example which combines a few containers and components:

```

<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="horizontal"
    horizontalGap="25">
    <mx:HBox width="596" height="441" backgroundColor="#FAA4A4">
        <mx:VBox height="100%" width="269" backgroundColor="#6DDBDA">
            <mx:Panel width="250" height="196" layout="horizontal"
title="One">
                <mx:Label text="Label"/>
                <mx:TextInput text="TextInput"/>
            </mx:Panel>
            <mx:Panel width="250" height="200" layout="absolute"
title="Two">
                <mx:TextArea height="128" text="TextArea" x="19"
y="10"/>
            </mx:Panel>
        </mx:VBox>
        <mx:VBox height="100%" width="270" horizontalAlign="left"
backgroundColor="#B9E396">
            <mx:Panel width="250" height="195" layout="vertical"
title="Three">
                <mx:CheckBox label="Checkbox1"/>
                <mx:CheckBox label="Checkbox2"/>
                <mx:CheckBox label="Checkbox3"/>
            </mx:Panel>
        </mx:VBox>
    </mx:HBox>
</mx:Application>

```

```

        <mx:LinkButton label="LinkButton"/>
    </mx:Panel>
    <mx:Panel width="250" height="200" layout="vertical"
        title="Four">
        <mx:RadioButton label="Radio2"/>
        <mx:RadioButton label="Radio1"/>
        <mx:RadioButton label="Radio3"/>
        <mx:Button label="Button"/>
    </mx:Panel>
</mx:VBox>
</mx:HBox>
</mx:Application>

```

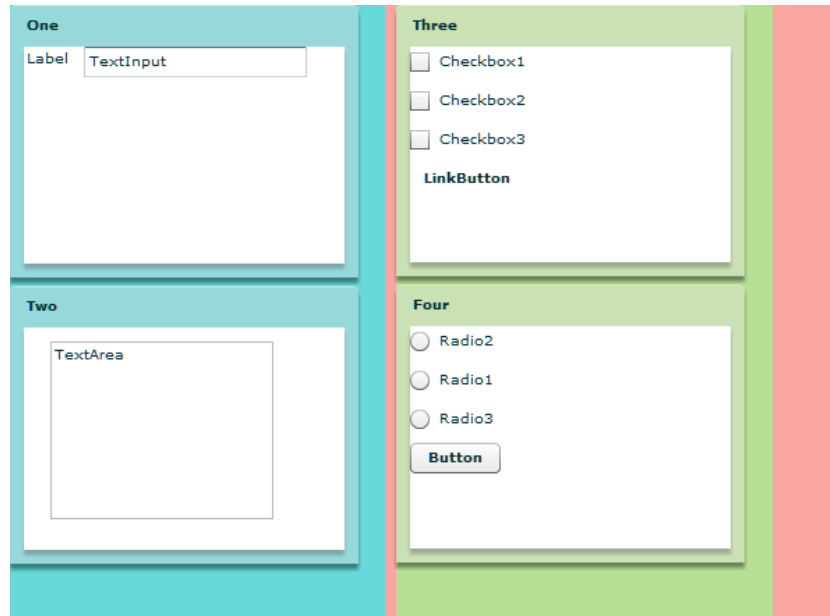
Layout

The containers control the flow of their contained items. There are generally three layout choices:

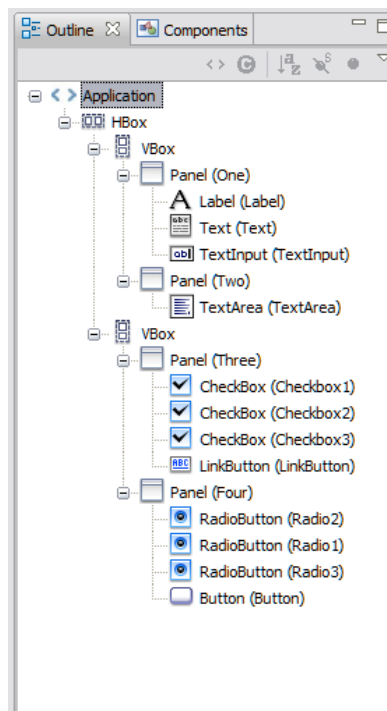
- Vertical - each child component stacks vertically from top to the bottom.
- Horizontal - Each child component stacks horizontally from left to right.
- Absolute – Layout is not automatic and requires you to specify the location of each child component. The Application and Panel containers can have absolute positioning if you specify it as the “layout” attribute. Canvas always uses absolute positioning.

Note that the only way to get Flex components to overlap is to use absolute layout.

The outermost container in the example is an HBox, which has a default horizontal flow. It contains two VBoxes, which therefore stack left to right, rather than top to bottom. But the items contained in the VBoxes will stack vertically. Each VBox contains two Panels, which do just that. A Panel has a title and a container area. Each Panel here contains some Control items, such as a TextInput or Button. Panel One was explicitly assigned a horizontal layout, the other Panels vertical.



If you are using Flex Builder, the Design View allows you to place and manipulate these interface components visually. Also, it has an Outline view that visually indicates the hierarchy of containment:



Handling Events

As Ajax does with its asynchronous processing, so Flex depends on events to determine timing and life cycle in an application. When a component dispatches an event, objects that are registered as handlers for that event are notified. You define event handlers in ActionScript to do the processing. You register event handlers for the particular events either in the MXML declaration for the component or in ActionScript. A

common event is *creationComplete*, which is an event the Application container dispatches when initialization of the Flex component is complete, similar to the *onload* event of an HTML document body.

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

The *init()* function would then be defined in ActionScript to do whatever initial setup is desired. Here is an example with an event handler ActionScript function we create called *clickHandler()*, which must receive an object of type *MouseEvent* as an argument. The *clickHandler()* function is registered to handle the click event of the button in MXML by assigning it to the button's *click* property:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  width="318" height="190"
  horizontalAlign="center" verticalAlign="middle"
  viewSourceURL="src/HandlingEventsEventHandler/index.html"
>
  <mx:Script>
    <![CDATA[
      import flash.events.MouseEvent;

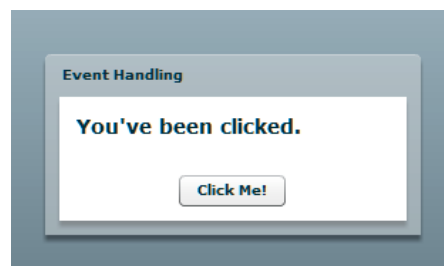
      private function clickHandler ( event:MouseEvent ):void
      {
        myLabel.text = "You've been clicked.";
      }
    ]]>
  </mx:Script>

  <mx:Panel
    title="Event Handling" horizontalAlign="center"
    paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10"
    height="127" width="264">

    <mx:Label id="myLabel" width="224" fontWeight="bold" fontSize="14"
      height="39"/>
    <mx:Button id="myButton" label="Click here"
      click="clickHandler(event);" />

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

Clicking on the button results in the dispatch of the event to the *clickHandler* function, which executes and populates the label control with text.



As with HTML and JavaScript, the event handler can be defined this way, or inline. Inline event handling would involve writing the event handling script directly in the *click* attribute of the Button tag. However, since this does not separate the MXML and ActionScript, this method usually suffers from poor readability

unless only one or two script statements are required. In this case, the <Script> tag and its contents are unnecessary, and the meat is all in the Button tag:

```
<mx:Label id="myLabel" width="224" fontWeight="bold" fontSize="14"
height="39"/>
<mx:Button id="myButton" label="Click here"
click="myLabel.text = 'You\'ve been clicked.';" />
```

Again analogous to HTML and JavaScript, it is a good habit to use single rather than double quotes in the inline ActionScript code, so the two sets of quotes will contrast and not confuse the compiler. Additionally note there was an apostrophe in the text which needed to be escaped (You\'ve) to avoid interpretation as a single-quote.

MXML Objects

While arrays and other data structure objects are obviously available in the object-oriented ActionScript language, some hierarchical data objects can be quickly created in MXML, using its native XML structure. Here is an array of simple objects:

```
<mx:Array>
  <mx:Object label="white" />
  <mx:Object label="red" />
  <mx:Object label="green" />
  <mx:Object label="blue" />
  <mx:Object label="yellow" />
  <mx:Object label="purple" />
  <mx:Object label="black" />
</mx:Array>
```

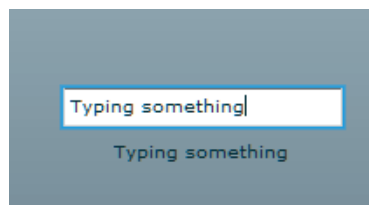
More properties can be added to an Object via XML attributes. This is a hard-coded example, but the values can be populated dynamically using data binding techniques.

Data Binding

A data binding copies the value of a property in one object to a property in another object. You can bind the properties of Flex components, Flex data models, and Flex data services. The data is copied from the source property to the destination property. The simplest method of specifying a data binding is with the curly brackets notation, as in this example:

```
<mx:TextInput id="Username"></mx:TextInput>
<mx:Label text="{ Username.text}"></mx:Label>
```

The curly brackets referencing the source property are placed in the destination property. This causes the label to update its text property as the text is being typed into the text input:



An ActionScript expression that returns a value can be placed inside the curly brackets.

```
<mx:TextInput id="Age" width="104"></mx:TextInput>
<mx:Label text="Your dog is { (Number(Age.text) as Number) * 7 } in human
years.">
</mx:Label>
```



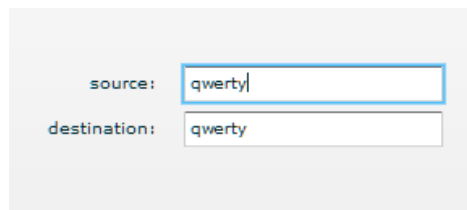

An alternative to the curly braces is the `<mx:Binding>` tag, which has explicit “source” and “destination” attributes. One benefit of this can be the separation of data binding code from user interface code. Also it is possible to specify multiple bindings of different source properties to the same destination property. Here is an example of the Binding tag:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- http://blog.flexexamples.com/2007/10/01/data-binding-in-flex/ -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical"
    verticalAlign="middle"
    backgroundColor="white">

    <mx:Binding source="textSrc.text"
        destination="textDst.text" />

    <mx:Form>
        <mx:FormItem label="source:">
            <mx:TextInput id="textSrc" />
        </mx:FormItem>
        <mx:FormItem label="destination:">
            <mx:TextInput id="textDst"
                width="{textSrc.width}" />
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

The Binding tag defines one text input as the source for the data change of the other text input (the typing input is being done in the source field):



The [Bindable] tag

The source of a data binding cannot be one of your own variables or class objects unless you declare it as bindable. You do this by putting the `[Bindable]` tag before its definition, like this:

```
[Bindable(event="xyzChanged")]
public var xyz;
```

The Bindable tag causes a source property to be broadcast, i.e. a change in its value will automatically be copied to any destination source set up bind to it, for instance with curly brace notation.

Specifying an event name means you are responsible for dispatching this event on change of the source property. If you do not name an event here, the *propertyChange* event is automatically created, and Flex dispatches the event whenever the property changes to trigger any data bindings. This example declares

two Number variables as bindable. Inline code for the Button click changes the values of the variables. Then the data binding for the text controls causes them to be notified of the updated values of the variables:

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

    <mx:Script>
        <![CDATA[

            [Bindable]
            public var maxNum:Number = 100;

            [Bindable]
            public var minNum:Number = 0;

        ]]>
    </mx:Script>
    <mx:Panel width="136" height="141" layout="vertical"
        horizontalAlign="center" title="Bindable">
        <mx:Text text="{minNum}"/>
        <mx:Text text="{maxNum}"/>
        <mx:Button click="maxNum=50; minNum=10;" label="Click here"/>
    </mx:Panel>
</mx:Application>
```



Before clicking



After clicking

Changing States

A common need for a more desktop-like experience in an application is for the interface to change based on the user's progress. When viewing records from a database for example, the view might change from a list form to a detail form. A *state* in Flex is a collection of changes for a particular view, including the addition or deletion of components, and changes to their properties and/or behaviors.

In any Flex application there is always at least a *base* state. This is the layout of your application outside of any state component declarations. Other states can be defined which are based on the base state by default, but can be based on another state. This is an example of a very simple state change:

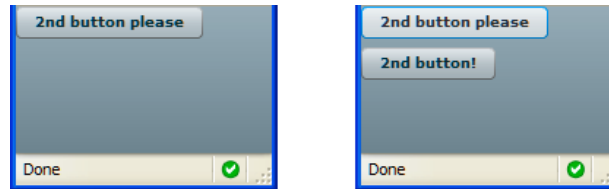
```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute">

    <mx:states>
        <mx:State name="newButton" basedOn="">
            <mx:AddChild relativeTo="{myBox}">
                <mx:Button id="button2" label="2nd button!" />
            </mx:AddChild>
        </mx:State>
    </mx:states>

    <mx:VBox id="myBox">
        <mx:Button label="2nd button please" click="currentState='newButton'"
    />
    </mx:VBox>
</mx:Application>
```

The base state is simply the “myBox” VBox container, and a button. Then within the `<mx:states>` tag, additional states are defined (here only one). The “newButton” state is based on the base state, so actually the “basedOn” attribute is not even necessary. The state of the layout is changed via the `<mx:AddChild>` tag, which adds a component, in this case a button. The `<mx:RemoveChild>` tag is available to remove a component from the view. The “relativeTo” attribute here indicates the child will be placed inside the “myBox” parent container; used in conjunction with the “position” attribute”, other positioning is possible, such as “before” or “lastChild”.

The event triggering the state change is the click of the base state’s button, which is set to change the “currentState” property, specifically to the “newButton” state. Here are the before and after views:



Before clicking

After clicking

A slightly more useful example is to switch from a simple search page to an advanced search page when the user clicks the link button:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  verticalAlign="top" horizontalAlign="center"
  width="400" height="240">

  <mx:states>

    <mx:State name="AdvSearch" basedOn="">
      <mx:AddChild relativeTo="{searchForm}" position="lastChild">
        <mx:FormItem label="This exact phrase:">
          <mx:TextInput/>
        </mx:FormItem>
      </mx:AddChild>

      <mx:AddChild relativeTo="{searchForm}" position="lastChild">
        <mx:FormItem label="Not these words:">
          <mx:TextInput/>
        </mx:FormItem>
      </mx:AddChild>

      <mx:SetProperty target="{searchPanel}"
        name="title" value="Advanced Search"/>

      <mx:SetProperty target="{simpleButton}"
        name="label" value="Search All Terms"/>

      <mx:RemoveChild target="{advLink}"/>

      <mx:AddChild relativeTo="{spl}" position="before">
        <mx:LinkButton label="To Simple Search"
          click="currentState=''" />
      </mx:AddChild>
    </mx:State>

  </mx:states>

  <mx:Panel
    title="Simple Search" id="searchPanel">
```

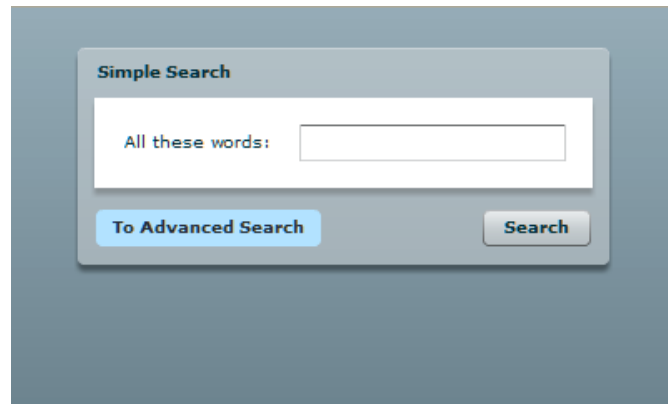
```
<mx:Form id="searchForm">
  <mx:FormItem label="All these words:">
    <mx:TextInput/>
  </mx:FormItem>
</mx:Form>

<mx:ControlBar>
  <mx:LinkButton
    label="To Advanced Search" id="advLink"
    click="currentState='AdvSearch'"/>

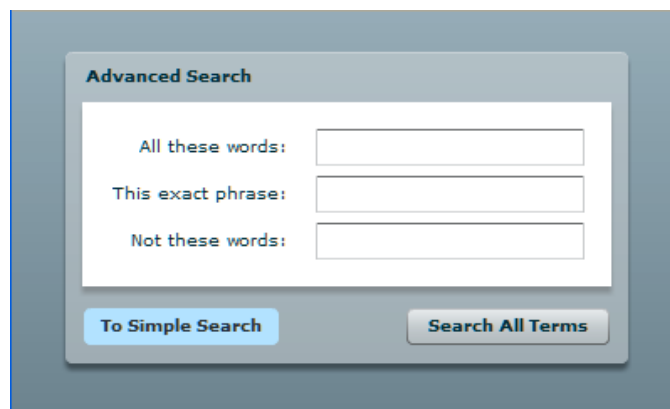
    <mx:Spacer width="100%" id="sp1"/>
    <mx:Button label="Search" id="simpleButton"/>
  </mx:ControlBar>
</mx:Panel>
</mx:Application>
```

The base state is a single form item with a text input. The advanced search state, “AdvSearch”, uses the `<mx:AddChild>` tag to add two more text inputs to the form. Then the “AdvSearch” state changes a couple of properties with the `<mx:SetProperty>` tag: the “title” property of the Panel container, and the “label” property of the button, so their text will reflect the current context.

Finally there is the matter of the LinkButton control, used to navigate between the two states. Rather than changing every single property, the `<mx:RemoveChild>` tag was used to remove the original LinkButton altogether, and `<mx:AddChild>` used to create a different one in its place. The position specified is right before the Spacer element (in the ControlBar). This results in the following:



Clicking “To Advanced Search” changes the state to...



Clicking “To Simple Search” reverts to the original state

4D FOR FLEX

Like with other server-side Web technologies, it is possible to connect to 4D’s Web server using objects like Flex’s `HttpService` or `WebService`. This would require a lot of server side coding, and the exchange of a lot of data, most likely through XML. Then there must be parallel coding effort on client and server.

But 4D For Flex takes advantage of the SQL Server found in 4D v11 SQL, and makes a direct connection to it with a proprietary SQL protocol, without an ODBC driver. Unlike HTTP it is a persistent connection, no server-side code is required, and data access is done with straight SQL queries.

4D For Flex Components

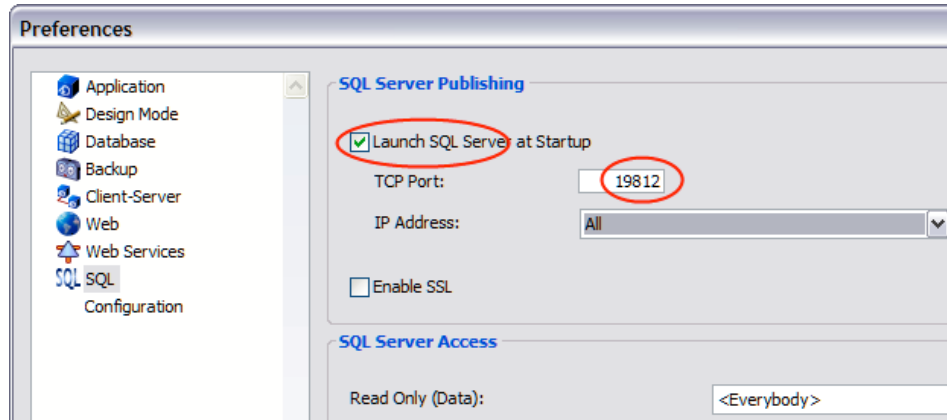
4D For Flex is the gateway between the 4D database and standard Flex controls and components. It consists of three major components:

- `Flex4D_SQL`: This is the minimum required for connecting with 4D. It includes the `SQLService` MXML component, which provides an easy-to-use query API for getting data from 4D.

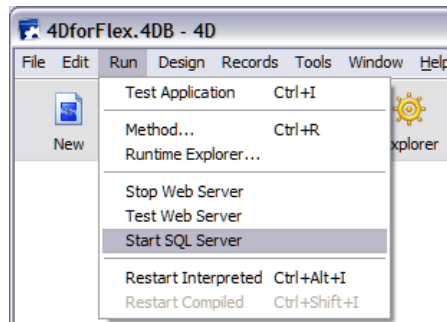
- Flex4D_Controls: Optional package which provides a custom DataGrid object, and set of navigation buttons for making user record selection easier.
- Flex4D_Components: Optional package which provides objects such as ImageRenderer, ConnectionDialog, Language selector.

Configuring 4D's SQL Server for Flex

The minimum version of 4D is 4D v11 SQL, release 2. The SQL server of 4D is what responds to front-end 4D for Flex queries, so it has to be running. The standard port is 19812. Either set the server to run on startup in the 4D Preferences dialog:



Or start the SQL server from the Run menu:



Flash Security Policy

In order to ensure the Flash Player (the runtime for compiled Flex code) can connect to the server, a cross-domain policy file is used. This specifies ports, domains, or IP numbers that have access to certain resources, specifically the SQL socket connection in our case. In the Preferences folder of the 4D database, there needs to be an SQL/Flash subfolder, containing a policy file called "socketpolicy.xml". The socketpolicy.xml file should at a minimum contain an entry like this:

```
<cross-domain-policy>
  <allow-access-from domain="*" to-ports="19812"/>
</cross-domain-policy>
```

This uses the * wildcard to allow entry for all domains to the SQL socket connection on port 19812.

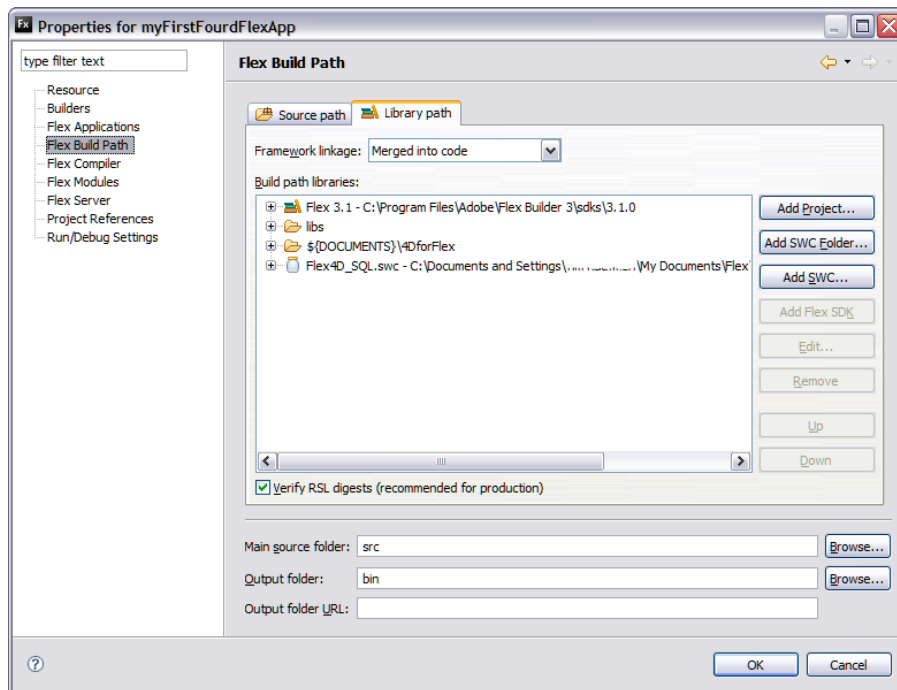
Including 4D For Flex in Your Flex Builder Project

The three major 4D for Flex packages are contained in SWC files, which are archive files for Flex components. The three 4D for Flex archive files are:

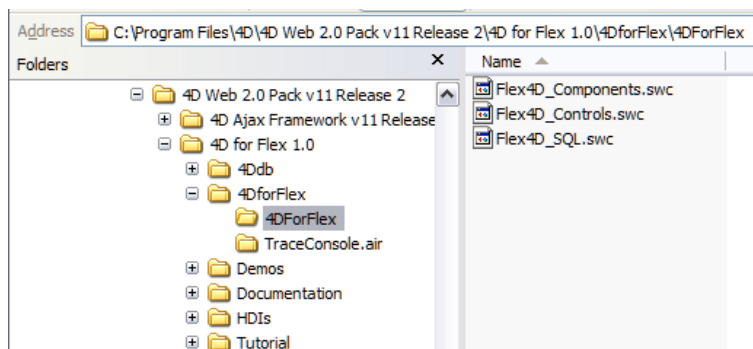
- Flex4D_SQL.swc

- Flex4D_Controls.swc
- Flex4D_Components.swc

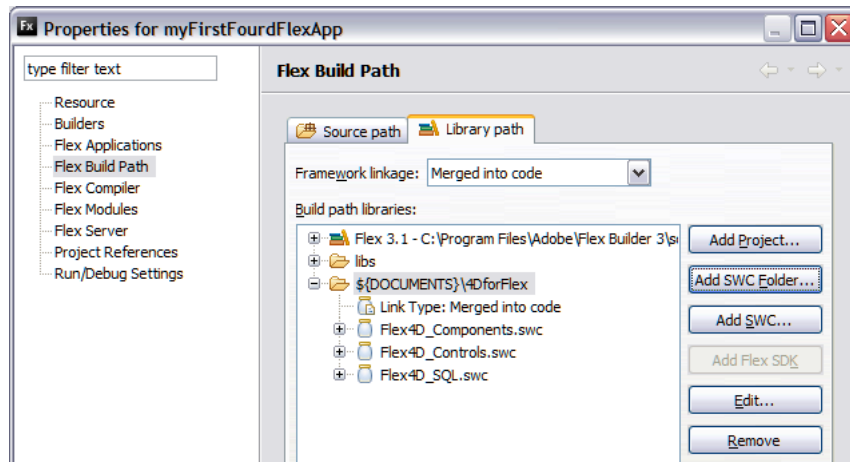
To include one or more of these components in the Flex Builder project, open the Project menu and select Properties. Select the “Flex Build Path” page, and the “Library path” tab.



(Also note that the “Output folder”, rather than pointing to the bin folder of the Flex project folder, can point to the Webfolder of the 4D database to cause the HTML and SWF files to be placed there.) Click on the “Add SWC...” button, and navigate to the 4D for Flex components in the Web 2.0 Pack installation files and select the desired component:

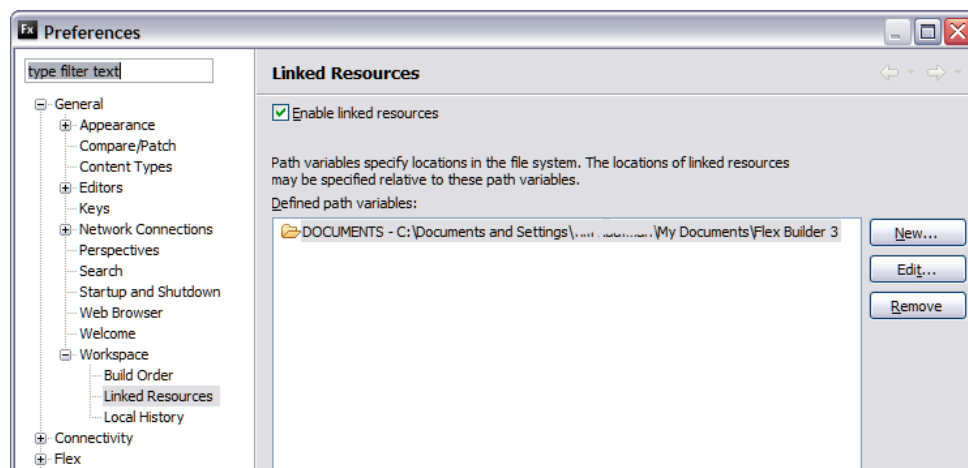


Note that it is also possible to point to the folder containing the 4D for Flex components, and they will all be included in the project, if you click on the “Add SWC Folder...” button and select the 4DforFlex folder:



The `${Documents}` variable is a convenience for specifying the path to your workspace or a central directory for the components, to which all your Flex projects can point. Here a 4DforFlex subfolder containing all three components was created in the `${Documents}` path, and was added as an SWC Folder to the project.

The `${Documents}` variable can be set under the Window menu, Preferences dialog in Flex Builder. Navigate through the hierarchy to General/Workspace/Linked Resources. Create a DOCUMENTS variable, and check “Enable linked resources”.



The SQLService Class

This is the basic class for connecting to the 4D SQL Server and executing queries. It can be declared in MXML, and has an API for communicating with 4D via SQL statements. For all the following sample code, assume that we have a single MXML file, and a single ActionScript file, named:

- My4dFlex.mxml
- My4dFlex.as

The SQLService object is simply declared in MXML, here right after the opening Application tag:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:n4d="http://www.4d.com/2007/mxml"
  creationComplete="init()"
  layout="vertical">
```



```

borderColor="#FFFFFF"
horizontalAlign="left"
verticalAlign="top">

<mx:Script source="RIA_pp5.as"/>

<!-- 4D for Flex SQLDataService -->
<n4d:SQLService
    id = "fourDSQLService"
    host = "10.96.0.26"
    userName = "Administrator"
    autoConnect="false"

    result = "resultHandler(event)"
    fault = "faultHandler(event)"
    connect="connectHandler(event)"

    disconnect="disconnectHandler(event)"/>
</mx:Application>

```

First note we have declared a namespace, “n4d”, for the 4D for Flex library components; this namespace can be called anything (as long as it doesn’t clash with another namespace). The Application components creationComplete event causes the init() function to execute, which will be defined in the ActionScript file.

This SQLService object will try to connect to the host that served the SWF file on the default port, since host and port were not specified. A login as Administrator with no password will happen. Handler functions, which need to be defined in the ActionScript file, are registered to several events. Result for when a query completes; fault for an error; connect for when a connection is started; disconnect for when the connection is closed.

Here is the start of the init() function in My4dFlex.as:

```

private function init():void
{
    fourDSQLService.connect();
}

```

The connection is attempted with the 4D SQL Server. The connectHandler() function will be called if this is successful:

```

private function connectHandler(event:Event):void
{
    var sql:String = "SELECT SpDate, Description, Total FROM Expense;"

    SQLService.execute(sql);
}

```

Once the SQLService.connect() function has been called, SQL statements can be executed, so this may be done right in the connectHandler() function.

The function registered to the SQLService’s result event (*resultHandler()*) is called after a successful query, so it is a good place to establish an SQLResultSet object:

```

import fourD.sql.SQLResultSet;

. . . .

private function resultHandler(event:ResultEvent):void
{
    var resultSet:SQLResultSet;
    myResultSet = event.result as SQLResultSet;
    someVariable.text = String(myResultSet.nbRecords) + " record(s) found";
}

```

The SQLResultSet Class

SQLResultSet is a data structure of the current selection from the 4D database. Here the nbRecords property was put to use, but other properties include the fields property, an ArrayCollection of the fields (in the current selection) as objects. (An ArrayCollection is a Flex wrapper class for the Array class that allows more convenient collection-based manipulation of the elements of the array.)

SQLResultSet also provides some convenient methods (member functions), such as *getItemAt(index:int)*, *insertRecord(record:Object)*, *updateRecordAt(index:int, fieldIndex:int)*, and *sortBy(fieldIndex:int)*, for accessing and manipulating data in the current selection.

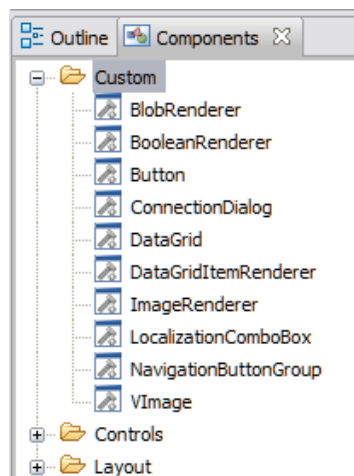
Import Statements

Note that the fully qualified name is fourD.sql.SQLResultSet; i.e. the SQLResultSet class belongs to the fourD.sql package. A package is an ActionScript directory of class files which share a namespace. To enable referencing this class simply as SQLResultSet in this ActionScript file, it is necessary to provide an import statement at the top of the file:

```
import fourD.sql.SQLResultSet;
```

The DataGrid Class

The 4D for Flex DataGrid is an inherited subclass of the standard Flex DataGrid, which is a list-based control to display large data sets in multiple columns. The 4D DataGrid has additional default properties, such as automatically populating header names with field names, and displaying fields in the order specified in the select statement. This class is part of the Flex4D_Controls.swc component, which needs to be added to the Flex build path. After this is done the 4D for Flex custom components can be seen in Flex Builder's Components view:



Here is an example of declaring the DataGrid in MXML:

```
<fourD:DataGrid
    id="expenseGrid"
    width="100%"
    height="95%"
    doubleClickEnabled="true"

    dataProvider="{myResultSet}"

    change="updateBalance()"
/>
```

```

<fourD:columns>
  <mx:DataGridColumn
    dataField="SpDate"
  />
  <mx:DataGridColumn
    dataField="Description"
  />
  <mx:DataGridColumn
    dataField="Total"
  />
</fourD:columns>
</fourD:DataGrid>

```

The most important thing to notice is the “dataProvider” attribute. Data binding is used to make the data source of this DataGrid object – guess who? – the `SQLResultSet` object, `resultSet`, from the previous section. Running the application with the `SQLResultSet` and `DataGrid` both defined properly results in a live Grid on the page, showing the current selection:

SpDate	Description	Total
01/01/2005	New Server for testing	\$3,299.00
05/12/2005	Hard Disk Raid Array2	\$4,999.00
11/15/2005	new printer	\$499.00
02/28/2006	Toner Cartridges	\$999.00
07/18/2006	Company Party	\$1,350.00
01/07/2007	New Seconed Printer	\$499.00
02/04/2007	Early iPhone Development Kit	\$2,999.00
06/20/2007	iPhone	\$499.00
06/18/2008	Colorescreen Radius HDI megacool	\$456.23

The `DataGrid` class provides some events to take advantage of, in addition to many properties and methods. Here a function called `updateBalance()` is registered to the *change* event of the Grid, to run whenever data is changed.

“Columns” is an Array property of the `DataGrid`, which can contain `DataGridColumn` objects, should you choose to define one or more in your MXML. The `dataField` property of the `DataGridColumn` is the name of the field in the data provider associated with the column, either hard-coded to field name or programmatically.

Alternatively, the `fields` property of the `ResultSet` object can be used to get the field names, rather than naming them explicitly, assuming you know the correct ordinal numbers of the fields you want:

```

<fourD:columns>
  <mx:DataGridColumn
    dataField="{resultSet.fields.getItemAt(0).name}"
  />
  <mx:DataGridColumn
    dataField="{resultSet.fields.getItemAt(1).name}"
  />
  <mx:DataGridColumn
    dataField="{resultSet.fields.getItemAt(2).name}"
  />
</fourD:columns>

```

Data Manipulation

The `SQLResultSet` object provides these methods for changing records in the current result set:

- `deleteRecordAt(index:int):void`
- `insertRecord(record:Object):void`
- `updateRecordAt(index:int, fieldIndex:int, value:?):void`

If the `autoSubmitChanges` property for the `SQLResultSet` object is not set to “True”, you must call:

```
myResultSet.submitChanges()
```

for the update to take place in the database. It may look like the change happened in the `DataGrid` on your page, but if `submitChanges()` was not called, you may find a page refresh makes you lose the update you thought you had performed.

Delete Records

To use `deleteRecordAt()`, you must know the index, or position, of the record in the `SQLResultSet`. If your result set is bound to a `DataGrid` object, you are often interested in performing the operation on the selected row. Since they are bound, you can provide the value of the `Grid`’s `selectedIndex` property to respond to the user’s row selection:

```
myResultSet.deleteRecordAt(myGrid.selectedIndex);  
myResultSet.submitChanges();  
  
myResultSet.refreshQuery();
```

For good measure, `refreshQuery()` was called so your `DataGrid` will update its display to reflect the change in records.

Insert Records

The new challenge here is `insertRecord()` expects to be passed an argument of type `Object`. Why is this? Think of the `Object` as something like an `Array`, except it can hold items of different data types. Create the `Object`, fill it with text, number and date properties to match the fields of the table in question, and call `insertRecord(Object)`.

```
var myObject:Object = new Object();  
  
myObject["Title"] = txtTitle.text;  
myObject["Location"] = txtLocation.text;  
myObject["Status"] = txtStatus.text;  
  
myResultSet.insertRecord(taskObject);  
myResultSet.submitChanges();  
  
myResultSet.refreshQuery();
```

Here the values from which to create the new record come from text inputs of a form on the page, referenced by their id property (“txtTitle”, etc.). The `myObject` object is used like an associative array, using field names instead of index numbers. Alternatively, the fields array object could have been used instead of hard-coding the field names:

```
myObject[myResultSet.fields[0].name] = txtTitle.text;  
myObject[myResultSet.fields[1].name] = txtLocation.text;  
myObject[myResultSet.fields[2].name] = txtStatus.text;
```

Update Records

The `updateRecordAt()` method will change the value of one field in a record, so it must be called repeatedly to update multiple fields. Here we update the value in the first field, “Title”:

```
myResultSet.updateRecordAt(myGrid.selectedIndex, 0, "Gazebo");
myResultSet.submitChanges();

myResultSet.refreshQuery();
```

However, assume we have the same “myObject” object from the above insertion operation, with data for three fields. We could use a loop to update all three fields in the record:

```
for(var i:int=0; i < 3; i++)
{
    myResultSet.updateRecordAt(myGrid.selectedIndex, i,
                               myObject[myResultSet.fields[i].name]);
}
myResultSet.submitChanges();

myResultSet.refreshQuery();
```

Working inside out, `myResultSet.fields[i]` is the field object for the current loop index; `myResultSet.fields[i].name` is the name of the field, i.e. “Title” for the first field; and `myObject[myResultSet.fields[i].name]` is the value stored in `myObject`, with the field name (i.e. “Title”) being used to dereference it.