

Flex Article

A Primer for Building Flex Applications

Kevin Hoyt

Platform Evangelist

Adobe

You're ready to make the Macromedia Flex plunge. You have installed the trial edition on your favorite application server. You have your favorite IDE ready (or maybe you decided to leverage Flex Builder (code-named Brady) and now you are ready to start coding your first Flex project. Hold on! Wait a second! Before you get started there are some things you should know about Flex that will make your life much easier.

Flex is a new product and you might think that best practices and patterns are still forming around the technology. In reality, the Flex programming model builds on traditional software engineering methodologies, best practices, and design patterns. Better understanding how Macromedia architected Flex can dramatically soften your learning curve.

In this article I cover some helpful hints that may not be immediately apparent when you start down the Flex development road. The first step is to understand the class library and how it manifests itself in the Macromedia Flex Markup Language (MXML). After that, I explain how you can dynamically create user interfaces (UI). At the end of this article, I will take a moment to stand on my soapbox and encourage you to think outside the box in terms of UI design—in hopes of improving your application performance at the same time.

Requirements

To complete this tutorial you will need to install the following software:

Macromedia Flex

Try (www.adobe.com/go/devcenter_flex_try) Buy (www.adobe.com/go/devcenter_flex_buy)

Prerequisite knowledge:

- You have experimented with Macromedia Flex
- You are familiar with object-oriented programming
- You are familiar with general design patterns

Understanding the Flex Class Library

There are three parts to understanding the Flex class library:

- Understanding what you are doing when you type `<mx:Button />` or any other tag
- Understanding that the Flex class library parallels the meaning of all those tags
- Being able to read (and find) the ActionScript documentation

First, I'll dig into what it means when you declare a user interface (UI) component using a tag.

Everything Is a Class

The Button class is a great starting point. Take a moment and ask yourself: How do I declare a button in MXML? If you are like most developers, you probably read the documentation and start with something like `<mx:Button`

```
label="Hello" />. There is nothing wrong with that, but what did you do? You instantiated a Button object from the class library, and at the same time, you set the label property for the object to a value of "Hello".
```

Everything in Flex is a class in the API. Ultimately, when you declare something using a tag, you create an instance of that class. There are two great ways to understand this at a deeper level. The first way is to set Flex up to save the ActionScript it generates that represents your application. The second way is to read specific parts of the documentation (which I will touch on momentarily).

You can enable this option in the flex-config.xml file by changing the property `keep-generated-as` from "false" to "true". When you enable this option and run your application, Flex will store the ActionScript file that represents your application in the same directory as the original MXML source file. If you open the generated ActionScript file and read it, you can see how Flex instantiates the button (and a lot more). This is not only an excellent way to understand your application at a much lower level but it also gives you an excellent way to debug your code.

ActionScript Class Documentation

Most beginning Flex developers are all too familiar with the *Developing Flex Applications* documentation, but did you know that there is another set of documentation that explains the class library? The *Flex ActionScript and MXML API Reference* is a great resource that should feel familiar for those of you familiar with JavaDoc.

In the *Flex ActionScript and MXML API Reference*, take a closer look at the Button class in the All Classes pane. Here, the docs list classes you can use in your Flex applications in alphabetical order. Find the Button class and click the provided link. The corresponding documentation appears in the main pane.

The first listing on this page is the class hierarchy. The Button class extends SimpleButton, which extends UIComponent, which extends UIObject. Each time a subclass is created, it adds more specifics to the behavior of the parent class. In this case you eventually end up with a full-fledged button and you can still leverage the classes that reside further up the tree for your own component needs. Try the following code example by creating an MXML file and running the application.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:SimpleButton />
</mx:Application>
```

This little application is not particularly useful, but now you should understand what it is that you are instantiating. If you scroll further down through the documentation, you will find a plethora of information about properties you may have never known existed. Keep in mind that the class at the bottom of the inheritance chain gets all the functionality of the previous classes. For example, you may have used the "width" property on your buttons before, and now you know that the property is actually implemented all the way back up the tree at UIObject.

If you expand your view and look at a potentially more complex UI component, you will find some interesting correlations between the classes and their MXML declarations. Most developers try the DataGrid control in one of their first projects (more on that later). A typical DataGrid declaration might look something like the following code listing:

```
<mx:DataGrid>
  <mx:columns>
    <mx:Array>
      <mx:DataGridColumn headerText="Name" />
      <mx:DataGridColumn headerText="Price" />
    </mx:Array>
  </mx:columns>
</mx:DataGrid>
```

Now that you know this code simply instantiates a class, what does the MXML do? If you reference the API documentation I mentioned above, you can see that “columns” is a property on the DataGrid class, and that it takes an Array of DataGridColumn objects. In turn, the DataGridColumn class has a “headerText” property that is displayed at the top of the column. Bringing this concept full circle, can you think of another way to declare a Button class with a label?

```
<mx:Button>
  <mx:label>Hello</mx:label>
  <mx:toolTip>Now I get it!</mx:toolTip>
</mx:Button>
```

To make this a little more interesting I added the `toolTip` property as well (found on `UIObject`). You can use this technique in a number of scenarios; more importantly, is that you now understand how the tags relate to the class library. Now you can read the MXML API and understand how to instantiate those classes, as necessary, in your Flex applications. It may seem that you are done, but you can take this quite a bit further.

Tip: The default layout of a Panel may look like it is a VBox container, but it is actually a Box with the `direction` property set to `vertical`. VBox and HBox layout containers are convenience classes that extend from Box. It turns out then that if you want to lay out controls inside of a Panel in a horizontal direction, you can set the `direction` property to `horizontal`. No need to nest another container!

Revisiting MXML Components

How does all this discussion on classes relate to making custom components? If MXML tags simply declare classes and set properties, shouldn't you then be able to create a component subclass using MXML? You bet you can, and the good news is that if you have played with Flex at all, it is likely that you have already done this to some degree, probably using an `<mx:HBox>` container or `<mx:VBox>` container! It is easy enough to understand how to take a box and make a subclass of that box by adding components to it, but what if you wanted to create a Label control subclass?

Now that you understand the class structure and how you declare those classes in MXML, you can extend even the most basic component without having to resort to a pure ActionScript class. In this example, I wanted to create a Label subclass that would handle formatting of currency automatically. In short, I wanted to move the `CurrencyFormatter` that I might otherwise use in my main application, inside of the Label component itself.

In a new MXML file, start with the standard XML declaration. After the XML declaration, establish an instance of the Label class. You are creating a component, so don't forget to specify the appropriate namespace. Once you have created an instance of the Label class, your code should look like the following code snippet:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Label xmlns:mx="http://www.macromedia.com/2003/mxml">

</mx:Label>
```

Since this Label subclass handles currency, the `text` property probably doesn't make complete sense; add a property to the class that accepts a numeric value into a property called **amount**. Remember that Number is just another class, and that you can declare classes in MXML. You will likely also want to set the initial value of the new Label subclass to a numeric value (such as 0).

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Label xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Number id="amount">0</mx:Number>
</mx:Label>
```

Lastly, format the text property that the label displays by default. There are formatter classes available in Flex, so it makes sense to use an instance of one of those formatters inside the new Label class. Add an instance of the `CurrencyFormatter` class as a property in your new subclass and set the values, formatting it as you prefer. Then, declare the text property and use the `CurrencyFormatter.format()` method to fine tune your display.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Label xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Number id="amount">0</mx:Number>
  <mx:CurrencyFormatter id="currency" precision="2" rounding="nearest" />
  <mx:text>{currency.format( amount )}</mx:text>
</mx:Label>
```

Congratulations, you have just created an MXML subclass for a simple control! Save your new MXML subclass as `CurrencyLabel.mxml` and reuse it. This is a very simple example, and probably one that's not altogether useful, but it further illustrates that what you do in tags is what you might otherwise do using ActionScript. The principles of object-oriented programming really start to shine through and you can start finding ways to apply common design patterns to your applications.

Next steps: Think about how you might apply validation, and raise and handle runtime errors if your application passed an alphabetic value to your new `CurrencyLabel` class.

Dynamically Creating User Interface Components

With a foothold firmly established in understanding what you are doing declaratively with MXML and how it relates to the Flex class library, you can start digging for some other really interesting features you might have otherwise overlooked. Now look at the `View.createChild()` method. This method requires:

- The name of a class or symbol to instantiate
- The name of the resulting object in the application
- Any initialization properties you wish to pass to the newly created child

Dynamic Creation How-To

Because the View class is the base class for almost every container, everything from the Application class to the HBox class to the Canvas class supports this method through inheritance. With this in mind, you can create an entire UI programmatically through ActionScript. In the following code example, you will add a Label, TextInput, and Button component to your application through this dynamic technique to create a simple "Hello World" example.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application initialize="initApp()"
  xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script>
  <![CDATA[
import mx.controls.Label;
import mx.controls.TextInput;
import mx.controls.Button;
import mx.containers.Panel;

public var btnSubmit:Button = null;
public var txtFirst:TextInput = null;

public function initApp( Void ):Void {
  var pnlHello:Panel = null;
  var init:Object = null;

  init = new Object();
  init.verticalAlign = "middle";
  init.direction = "horizontal";
  init.title = "Hello World";
  pnlHello = Panel( createChild( Panel, "pnlHello", init ) );

  init = new Object();
  init.text = "First name:";
  pnlHello.createChild( Label, "lblFirst", init );

  init = new Object();
  init.text = "Macromedia";
  txtFirst = TextInput( pnlHello.createChild( TextInput, "txtFirst", init ) );

  init = new Object();
  init.label = "Submit";
  btnSubmit = Button( pnlHello.createChild( Button, "btnSubmit", init ) );
  btnSubmit.addEventListener( "click", this );
}

public function handleEvent( event:Object ):Void {
  if( event.type == "click" ) {
    if( event.target == btnSubmit ) {
      alert( "Hello " + txtFirst.text + "!", "Alert" );
    }
  }
}
]]>
</mx:Script>
</mx:Application>
```

There are a few interesting points about this code example:

- There are no MXML tags beyond the required `<mx:Application>` tag, and the `<mx:Script>` tag for the block of script you provide.
- You create every part of the UI dynamically from script. By doing this, remember that without the tags, the appropriate packages are not included, and thus you must use an import statement to ensure that the compiler is aware of the objects you will create in script.

Here, I have chosen an `initApp()` method that creates the UI, which is called during the initialize event of the application. For all practical purposes, the end user could drive the creation of the UI. I will talk more about this soon. Notice that I have also declared object variables for the UI components I wish to use outside of the `initApp()` method.

Once I started creating the UI objects, I used an internal variable of type `Object` to hold my initialization properties. You may also use the classic ActionScript approach, using brackets (such as `{text:"Macromedia"}`) inline.

The example may also introduce you to casting in ActionScript which I do to leverage the benefits of strict data typing. By default, `View.createChild()` actually returns a `MovieClip` object. This general data type may cause confusion or errors down the road, so while casting isn't necessarily required, I encourage you to use it for code maintainability. Also notice that when creating the `Label` object, I ignore the return altogether. Since I won't programmatically manipulate the `Label` object during the application, there's no need to store a named reference.

Lastly, notice how event handling differs when building dynamic UIs through ActionScript. In this example, the submit button must raise its click event so that the application can display an alert dialog box. Rather than use the init object, you leverage the `UIEventDispatcher.addEventListener()` method. When triggered, the `EventDispatcher` calls an `handleEvent()` method, which you must implement.

The `handleEvent()` method takes an object as an argument which you can use to understand which component raised the event. The object argument has two properties: one specifies the event type and the other the event target (the component that fired the event). Since all components used in this programmatic fashion might call the `handleEvent()` method, you need to examine the type and target properties to isolate events and take appropriate action.

Tip: When implementing the Model View Controller (MVC) composite pattern, it can be useful to leverage the event object by adding additional data properties. For example, if your MXML component contains a `<mx:DataGrid>` that triggers a change event, you will likely pass the `selectedItem` from the grid to any listeners. You might be inclined to have the component parent access the property directly, but this creates a tight coupling between UI elements. It is cleaner to add the `selectedItem` to the event object and let the parent leverage that property.

When to Use Dynamic Creation

One key flaw to note in this example: It requires far more code and can be substantially less maintainable. Clearly, you do not want to leverage this technique for generic UI requirements, but it can prove exceptionally useful in a number of scenarios. Generally, you can use this method to add components to your application at runtime when you might otherwise not know how many you will need.

An example where this type of dynamic creation is useful might be a product comparison matrix. For instance, let's say that your application lists products that are similar in nature. You would like to create functionality so that the user can drag products to an area on the screen that presents more information about those products for side-by-side comparison. Once the user has dropped a product on the comparison area, you can now create an instance of the special comparison view (MXML component) and pass it the required data.

There is also the opposite method, `View.destroyChild()`, that you can use to remove controls from the user interface.

Thinking Outside the Proverbial Box

If there is one thing I have seen time and again, it is that developers stick with what they know. Web application developers know tables, grids, and page refreshes. Client/server developers know dialog box windows and multiple document interfaces (MDI). In the case of Flex this can be very problematic. Flex applications are not web applications, and they are not client/server either. Flex is its own hybrid, and requires that developers approach problems from a fresh perspective.

The HALO look and feel used throughout Flex is a solid way to start addressing this hybrid environment. You can learn more about HALO from *Mike Sundermeyer's article* (www.adobe.com/devnet/flex/articles/halo.html) . Mike makes a point that Flex applications “aren't OS-native applications; they are Internet applications that run in all different browsers and platforms.”

This brings you as far as a component foundation, but what about overall UI considerations?

Take the scenario of allowing the user to zoom in on an image. From a web application standpoint, you know that file size limits what you can do. You don't want to deliver the high-resolution image, but scaling of a smaller image can result in pixilated image. From a client/server standpoint, you might be tempted to use the client's power to read the high-resolution image outright and manipulate the image bits. Neither is ideal for a Flex application, but you can find a happy, hybrid, median.

In a Flex application, it is the server that holds most of the processing power, but the client is intelligent. A solution to this problem, then, might be to have the server process the image based on event-driven requirements and deliver the resulting scaled and cropped image back to the client. In a Flex application, the client is responsible for letting the server know of its exact needs: the available viewing space and the pixel coordinates the user has specified for a zoom operation. This hybrid approach can result in the ability to have a client (Flex application) view a multi-megabyte high-resolution image and yet deliver only a few kilobytes of data at any one point in time.

The trick is to think outside the box. When developing Flex applications, you might be tempted to stick to what you know. In some cases this may mean believing that Flex cannot deliver the required functionality, and yet in others it may mean expecting too much from the Flex application during runtime. Solutions you may have historically dismissed as challenging or impossible can now come into existence because of Flex. For more examples of thinking outside the box, check out *my blog*.

About the author

Kevin Hoyt is a platform evangelist with Adobe Systems, Inc. Passionate about engaging user experiences as he is, you'll most often find him meeting with customers, speaking at conferences, presenting online seminars, or just enjoying the chance to share ideas and brainstorm with other developers. When not on the road, Kevin enjoys spending time with his family, photography, and general aviation.