

# Flex for Flash Developers

(updates and corrections will be published at <http://weblogs.macromedia.com/sho/>)

## Introduction

What is Flex? Why did we make it?

One of the main reasons we made Flex is that application developers told us that the Flash authoring tool was too confusing for them. To help these people out, we created a tag-based language and a rich class library that includes all the basic building blocks you need to get an application up and running. The simplest Flex application is just a few lines of code.

So here's a question: if Flex is for people who were confused by Flash, why am I giving a talk about Flex for Flash developers? Even if you are Flash expert, you may still want to get your head around Flex if you fall into one of the following categories:

- You want a more structured way to build applications.
- You don't feel like reinventing the wheel (layout management code, anyone?).
- You need to work with people who are not used to the Flash authoring paradigm.
- Your boss wants you to use Flex.

One final thing: teaching Flex at even a beginner level is something that requires a minimum of several days. Meanwhile, I hate going to talks that are too high level and don't end up getting to the meat of things. Because of this, I split the difference and spend some time on a brief overview of the whole thing, followed by a few deep dives into specific topics. There are a lot of aspects of Flex that I don't get a chance to cover (such as data handling), but there is lots of great material on the Adobe website if you are interested.

## Overview (aka what is Flex?)

If you know anything about Flex 1.0, you should be aware that with version 2.0, you no longer have to buy a server to build Flex applications. In exchange, we have developed some pretty sophisticated data handling services that now form the core of the server product. If you need these new services, buy the server. If you don't, just pick up the tool and off you go.

So if Flex isn't the server, what is Flex?

Flex is...

- A way for people to create Flash applications strictly through text files (tags and script)
- A set of ActionScript classes that form the building blocks for every Flex application.
- A compiler, debugger, and other tools that help make all this possible.

Flex is similar to HTML in that it is a tag-based way to create user interfaces. Unlike HTML, it was designed from the ground up for the needs of application development. That probably sounds like marketing talk, but it's true. The creators of HTML never thought that people would be creating things that look like tab controls by drawing a picture of a tab control, slicing it up, and knitting pages together using hyperlinks.

Flex (and Flex Builder) is also similar to Flash, in that you launch a tool, create a project, knit together ActionScript classes and program logic, hit "compile" and create a SWF that you can put on any web server.

Flex is also like traditional desktop UI frameworks, such as MFC, NextStep, etc. There is a lot you can do in Flex that feels more like traditional OO desktop development.

## Step 1 – hello, world!

Here's the obligatory hello world app. You probably won't need much explanation to understand what it does.

### Step1.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<Application xmlns="http://www.macromedia.com/2005/mxml">
  <Panel>
    <Label text="Hello, world!" />
    <Button label="goodbye" />
  </Panel>
</Application>
```

As you can see, this is a well formed XML document. From here on out, I'll omit the XML declaration and namespace declaration for the sake of readability.

So what's going on? <Panel> is a tag that creates an area of the screen that looks like a small window. Unlike windows, you can't move these things around, and they are not supposed to overlap. By default, panels, application, and every other container has a default way to lay out its children. In this case, the items inside the panel are laid out vertically. Finally, you will notice that there are no sizes on anything. Labels and buttons, like every other control, know how to size themselves. Panels, like other containers, know how to size themselves based on what is inside them. The application takes up the whole browser by default, and centers its contents at the top of the browser window.

Under the hood, there is an ActionScript class that corresponds to each tag. You can use these classes from ActionScript, and you can create your own tags by writing code. In fact, every time you create an MXML file, you are actually creating an ActionScript class. The code above is basically equivalent to the following:

### Step1.as

```
Class Step1 extends Application
{
```

```
override public function initialize(): void
{
    var panel : Panel = new Panel();

    var label : Label = new Label();
    label.text = "Hello, world!";
    panel.addChild(label);

    var button : Button = new Button();
    button.label = "goodbye";
    panel.addChild(button);

    this.addChild(panel);
}
```

Of course, the real code is much hairier because it has to deal with all the complexities of real world applications. To see the real code, use the developer version of Flash Enterprise Services and watch the files that it generates!

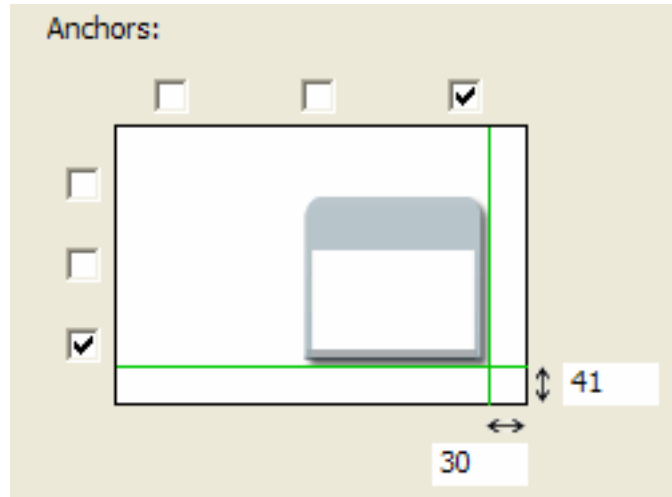
## Step 2 – Flexible layouts

Ok... enough with the under the hood analysis. Let's go back to the MXML version and play around with it. One nice thing about Flex and Flex Builder is that you can use the IDE to create flexible layouts.

You can drag and drop controls within the IDE to make a layout that looks like this:



And you can use the anchor widget to tell the button and text field how they should move as the window stretches.



The code looks like this:

### Step2.mxml

```
<Application>
  <Panel layout="absolute" width="80%" height="80%">
    <TextArea text="Hello, world!"
      top="10" bottom="71" left="10" right="30"/>
    <Button label="goodbye" right="30" bottom="41"/>
  </Panel>
</Application>
```

### Step 3 – Styling through CSS

What do we do if we want to change the way this looks? One good way to do this is through CSS. The basic CSS styles for things like font styles all work, and we have added many new properties to deal with everything from panel header height to the amount of blur that should occur behind a modal dialog box!

### Step3.mxml

```
<Application>
  <Style>
    <![CDATA[
      TextArea {
        font-size: 36px;
        font-weight: bold;
      }
    ]]>
  </Style>

  <Panel layout="absolute" width="80%" height="80%">
    <TextArea text="Hello, world!"
      top="10" bottom="71" left="10" right="30"/>
    <Button label="goodbye" right="30" bottom="41"/>
  </Panel>
</Application>
```

This is a declaration that says that all TextArea tags should have this style applied. You can also create styles for named classes. Other types of selectors (id selectors, contextual selectors, etc.) are not supported.

This CSS can also be put into an external file for easier maintenance. Plus, you don't have to worry about marking it as CDATA anymore.

## Step 4 – Adding behavior

Now, let's make the application do something. Well, there's a button labeled goodbye that looks pretty tempting. Let's write some code to make the panel disappear.

### Step4a.mxml

```
<Application>
  <Style source="style.css" />
  <Panel id="myPanel" layout="absolute" width="80%" height="80%">
    <TextArea text="Hello, world!"
      top="10" bottom="71" left="10" right="30"/>
    <Button label="goodbye" right="30" bottom="41"
      click="myPanel.visible=false"/>
  </Panel>
</Application>
```

Notice how we had to give the panel an id in order to be able to refer to it.

In general, it's not a good idea to write a lot of code directly in the event handler, so let's isolate this code into a function call.

### Step4b.mxml

```
<Application>
  <Style source="style.css" />
  <Script>
    <![CDATA[
      public function goodbye() : void {
        myPanel.visible = false;
      }
    ]]>
  </Script>

  <Panel id="myPanel" layout="absolute" width="80%" height="80%">
    <TextArea text="Hello, world!"
      top="10" bottom="71" left="10" right="30"/>
    <Button label="goodbye" right="30" bottom="41"
      click="goodbye()"/>
  </Panel>
</Application>
```

Another best practice is to try to keep ActionScript and MXML separate. In order to separate out the code as much as possible, it is a good idea to attach the event handler for the button from ActionScript instead of MXML.

### Step4c.xml

```
<Application initialize="doInit()">
  <Style source="style.css" />
  <Script source="Step4c_code.as" />

  <Panel id="myPanel" layout="absolute" width="80%" height="80%">
    <TextArea text="Hello, world!"
      top="10" bottom="71" left="10" right="30"/>
    <Button id="goodbyeButton" label="goodbye"
      right="30" bottom="41" />
  </Panel>
</Application>
```

### Step4b\_code.as

```
public function doInit() : void
{
    goodbyeButton.addEventListener("click", goodbye);
}

public function goodbye(event: Event) : void
{
    myPanel.visible = false;
}
```

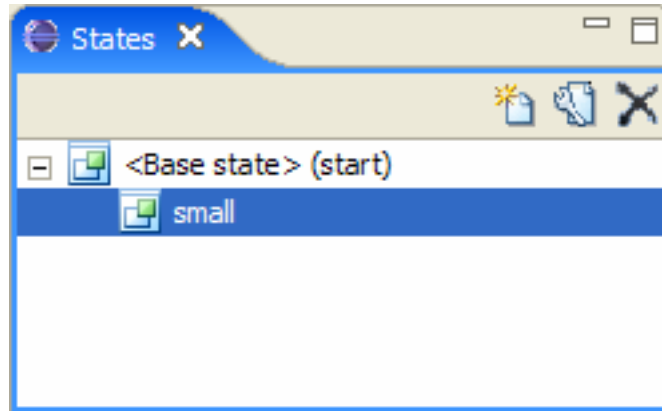
## Step 5 – Adding states

Next, let's add states to this application. But before doing that, here's a little context.

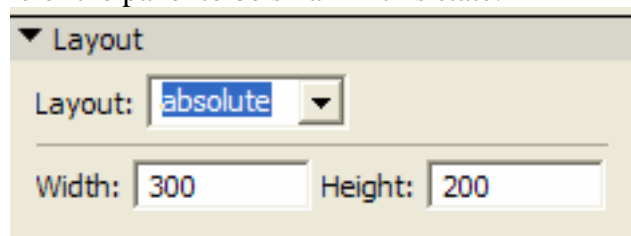
In HTML, you represent different parts of your application by using different HTML pages. In Flash, there is only one file, the Flash movie. In some ways, this is great. As you navigate to different parts of your application, you can make the application change in a fluid way without a page refresh. The downside is that you lose the sense of each "place" being a different "place", which can be hard during development time (everything tends to be done through ActionScript).

In MXML, this is accomplished using states. Each state represents a configuration of the MXML file. In each state, you can create new controls, remove controls, or set properties on existing controls. From the Flash authoring perspective, states sort of like keyframes and sort of like screens.

It is easiest to work with states from within the IDE. Just create the states you need, click on them, and work in design view as normal. Let's create a state called "small":



Now, let's set the size of the panel to be small in this state:



We also add a button labeled “small” to make the panel smaller. The code should now look like this:

### Step5.mxml

```
<Application initialize="doInit()">
  <states>
    <State name="small">
      <SetProperty target="{myPanel}"
        property="height" value="200"/>
      <SetProperty target="{myPanel}"
        property="width" value="300"/>
    </State>
  </states>

  <Style source="style.css"/>
  <Script source="Step5_code.as"/>

  <Panel id="myPanel" layout="absolute" width="80%" height="80%">
    <TextArea text="Hello, world!"
      top="10" bottom="71" left="10" right="30"/>
    <Button id="goodbyeButton" label="goodbye"
      right="30" bottom="41"/>
    <Button id="smallButton" label="small" left="10" bottom="41"/>
  </Panel>
</Application>
```

We finish this off by adding a handler for the new button to tell the application to enter the “small” state.

### Step5\_code.as

```

public function doInit() : void
{
    goodbyeButton.addEventListener("click", goodbye);
    smallButton.addEventListener("click", makeSmall);
}

public function goodbye(event: Event) : void
{
    myPanel.visible = false;
}

public function makeSmall(event: Event) : void
{
    currentState = "small";
}

```

## Step 6 – Adding effects

Let's add some effects to finish this application off. Wouldn't it be nice if the panel smoothly resized and faded out when it went away?

### Step6.mxml

```

<Application initialize="doInit()">
    <states>
        <State name="small">
            <SetProperty target="{myPanel}"
                property="height" value="200"/>
            <SetProperty target="{myPanel}"
                property="width" value="300"/>
        </State>
    </states>

    <Style source="style.css"/>
    <Script source="Step5_code.as"/>

    <Panel id="myPanel" layout="absolute" width="80%" height="80%"
        hideEffect="Fade" resizeEffect="Resize">
        <TextArea text="Hello, world!"
            top="10" bottom="71" left="10" right="30"/>
        <Button id="goodbyeButton" label="goodbye"
            right="30" bottom="41"/>
        <Button id="smallButton" label="small" left="10" bottom="41"/>
    </Panel>
</Application>

```

This is only the tip of the iceberg when it comes to Flex, but this should give a general idea of the concepts that you'll need to know as we do some deeper dives on specific topics.

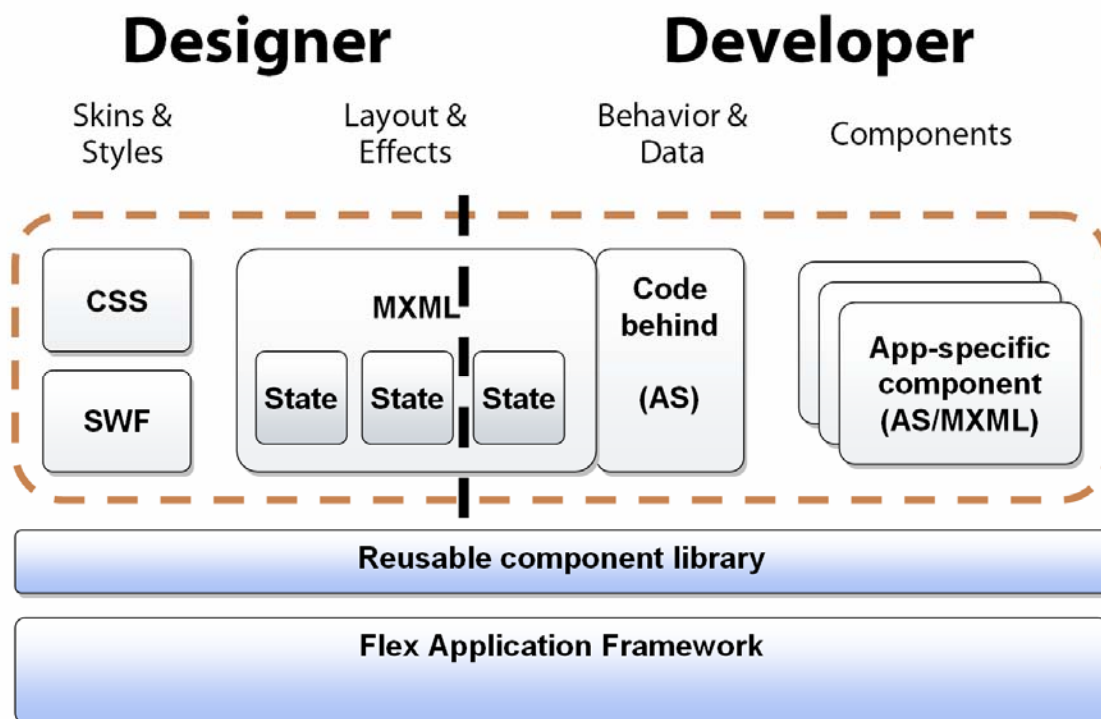
## Designer / developer workflow

While Flex does make it easier for traditional application developers to create Flash applications, you still need to structure your application cleanly so that designers and developers can work together effectively.

When it comes to design tasks and development tasks, certain file types are clearly in one camp or the other. CSS files are used for styling. SWFs are used for skinning individual components. Designers can change these files without affecting the developer at all.

Meanwhile, program logic and component development are primarily done in ActionScript. While many designers delve into ActionScript from time to time, ActionScript files are obviously more in the developer camp than the design camp.

The area with the biggest overlap is MXML. Design tasks and development tasks can both be done in MXML, which can cause issues.



To avoid tangled code in which designers and developers end up screwing each other up, try following these guidelines:

- Separate out code into “code behind” pages .
  - Keep ActionScript code separate from MXML using `<Script source="xxx">` or view helper classes.
  - Add event handlers from ActionScript, not MXML.
  - Try not to write fancy data binding code in MXML.

- Do not create visual components programmatically unless absolutely necessary.
- Do as much styling in CSS as possible.
- Use states when possible to express changes to user interface.

## Skinning and Styling

Because Flex applications tend to be more structured than Flash applications, you need to use specific mechanisms built into Flex for visual styling. Instead of first drawing a button and then making it into a button, you do the reverse: create a button, and then later attach a SWF or a bitmap image that specifies how it looks.

Each Flex component has specific pieces that can be replaced by image or SWF assets. For example, buttons have up, down and over skins. More complicated components, such as scrollbars, may have smaller pieces that can be skinned separately, such as the up arrow, down arrow, etc.

When skinning, say, a button, you can choose look that applies to all buttons in the entire application, or you can create a named style that you can apply to one or more buttons as you see fit. This is done through the CSS mechanism.

These skins may need to be stretched in order to display properly. If the skin is a SWF, you can use scale 9 within Flash authoring. If the skin is a GIF or a PNG, you can set the scale 9 coordinates using CSS syntax.

CSS can also be used to style virtually every aspect of the standard Flex controls. A great way to explore these styles is to visit:

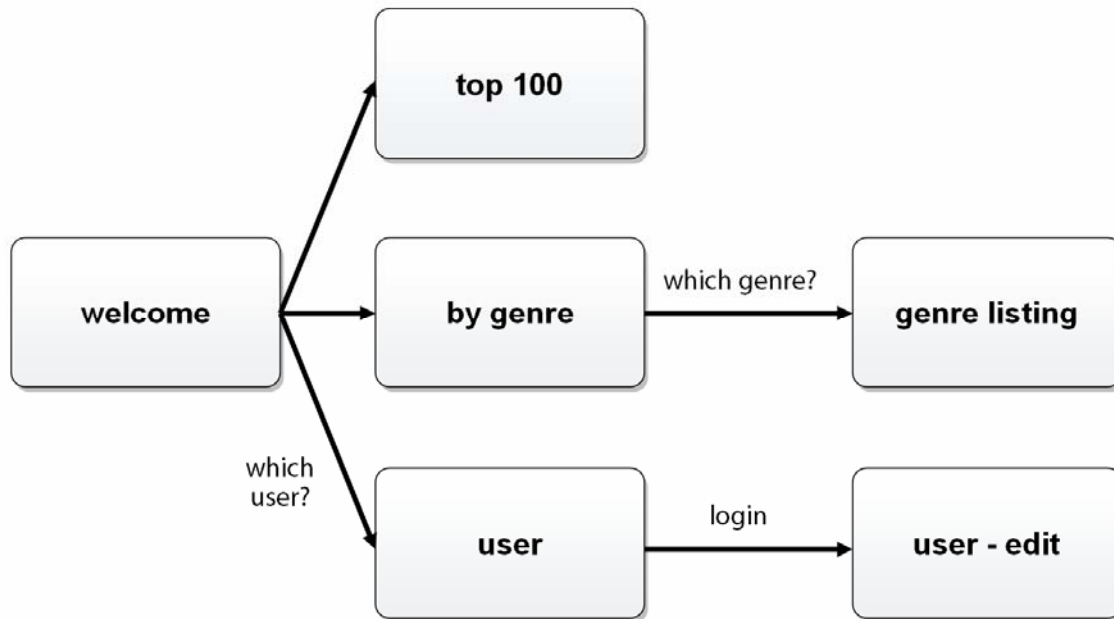
<http://weblogs.macromedia.com/mc/archives/FlexStyleExplorer.html>.

## Working with states

As we saw earlier, states are a way to create different configurations of your HTML application that are something like “pages” in an HTML application. Let’s take a closer look at how this works.

Let’s say we are creating a music browsing application. The application has a welcome screen, a listing of top 100 songs, a listing of songs by genre, and a listing of the songs that a particular user has. Furthermore, if you log in, you will be able to edit your list of songs.

The first thing to do is to figure out what states you need by creating a state diagram that represents how the user moves between our states. Ours might look something like this:



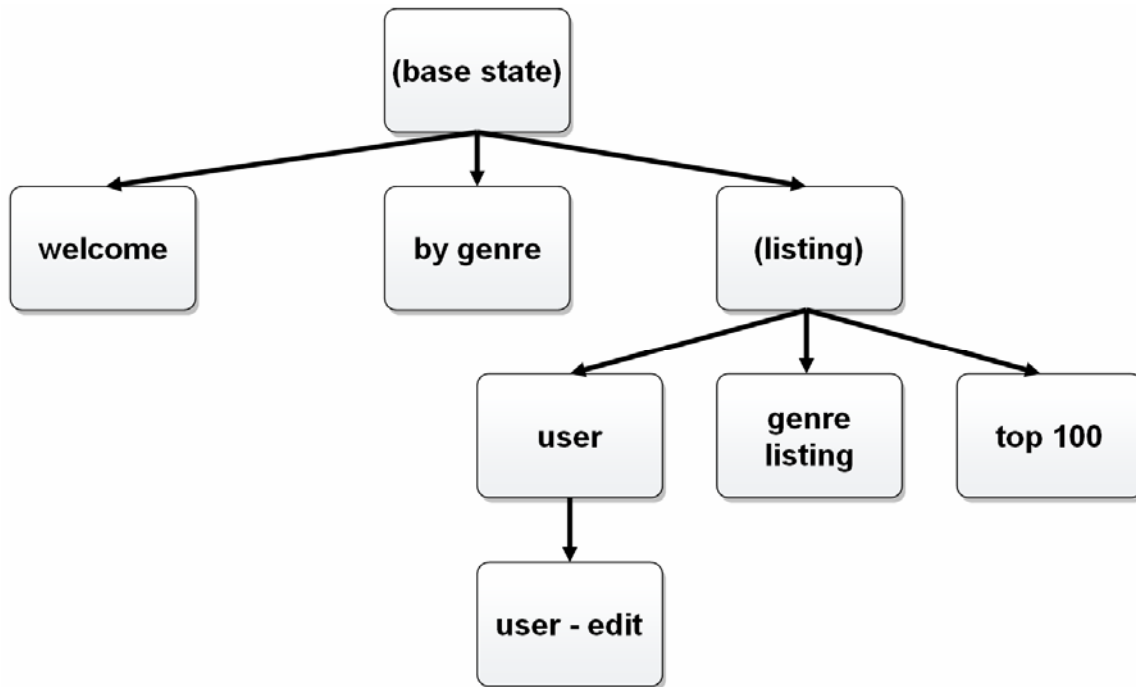
The user will enter the application in the welcome state, and depending on the user's action, the application will enter various states.

In some cases, extra data will be required. When going to the user state, we will need to know which user to display. When going to the "edit" page, the user will be required to log in.

This state diagram has been simplified in order to make it easier to explain. In a real world example, you probably want a way to log in directly from the welcome screen. You will also need a transition that represents what happens when the login fails.

Each one of these states will look different. However, the states will all be related to one another in one way or another. For starters, every state will share the common look and feel, including things like navigation, corporate logo, etc. Meanwhile, let's say that the top 100 listing and the genre listing are very similar to each other, the only difference being the data being displayed.

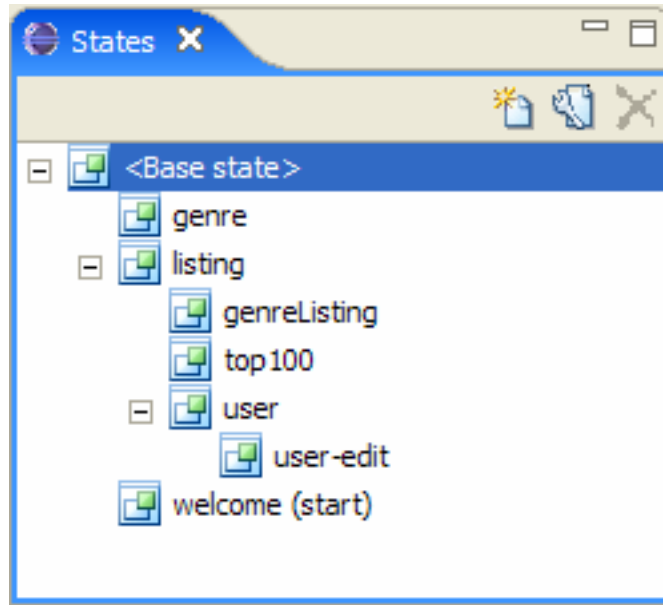
We can represent these visual relationships using a state inheritance diagram:



Note that the arrows here don't connect in the same way as the state transition diagram earlier. These two diagrams represent very different concepts. Note also that we had to introduce new states to represent these relationships. These states are called abstract states, and I put parentheses around them so you can tell which ones they are.

The user never enters an abstract state. Their only purpose is for you to be able to place things in them that are shared by all the states that are based on them.

Next, you will need to go into Flex Builder and create states that correspond to the picture above:



Everything that you do in a state will also show up in the states that are based on that state. For example, if I change the background color in the listing state, four states (“genreListing”, “top100”, “user”, and “user-edit”) will have the new background color. If I add a button in the “user” state, the “user-edit” state will also have that new button, but no others.

Because of this, you can see how important it is to map out your states before you create them.

## Effects and transitions

There are three main ways in which effects run within Flex:

1. In response to an “effect trigger”, such as `showEffect` or `hideEffect`.
2. When you change from one state to another.
3. When you explicitly call an effect from within `ActionScript`.

The first method is probably the most foreign to you if you are coming from the Flash world. Let’s say you want to fade out an image. Instead of clicking on the image and fading it using the timeline or programmatically fading it by calling an effect from `ActionScript`, you do the following:

- Attach a fade effect to the `hideEffect` trigger for the image.
- Hide the image by saying `myImage.visible = false`.

The second method is somewhat like the first method, in the sense that it’s just a case of an effect running when something happens (in this case when the state changes). However, the way I think of it is a bit different. Think of the state transition diagram above that describes the ways in which users can move through various states. Transitions specify the effects that should run as the user moves through these states.

The third method is probably the most familiar to you if you are already a Flash user, which is that effects can be run programmatically whenever you want.

No matter which method you use to get effects to fire, you can create your effects in one of two ways.

1. Create you effect as an ActionScript class that subclasses one of the built-in effect classes.
2. Use `<Parallel>` and `<Sequence>` to combine multiple effects into a bigger effect.

You can think of parallel and sequential sets of events as being very similar to a timeline. In fact, you should be able to create a Flex application that displays parallel and sequential effect combinations using a timeline-like UI! However, unlike timelines in Flash, you can attach these effects to different targets at runtime using data binding. You can also change the parameters of the effects at runtime.

For example, let's say you wanted to take whichever panel the user had clicked on and fade out the contents for 500ms, followed by a move and a resize in which you want the panel to occupy the "main panel" area of the screen during the next 500ms. Your effect might look something like this:

```
<Parallel target="{currentPanel}">
  <Dissolve duration="500" alphaFrom="1" alphaTo="0" />
  <Dissolve duration="500" startDelay="500"
    alphaFrom="0" alphaTo="0" />
  <Resize duration="500" startDelay="500"
    widthTo="{main.width}" heightTo="{main.height}" />
  <Move duration="500" startDelay="500"
    xTo="{main.x}" yTo="{main.y}" />
</Parallel>
```

There is one final gotcha that you should be aware of when using transitions and states. Because states and effects are both mechanisms for changing the properties of things that are on the screen, they can sometimes compete with one another.

Let's say you have a state change in which a panel's visibility gets set to false. Now, let's imagine that you want to fade this panel out. If the state change happens before the effect runs, you will never see the effect! Meanwhile, if the state change were always to happen after the effect ran, there would be other cases that wouldn't work properly.

The way around this is to choreograph the pieces of the state change directly within your effect using `<SetPropertyAction>`, `<AddChildAction>` and `<RemoveChildAction>`. In this case, you would write code that looked like this:

```
<Transition fromState="from" toState="to">
  <Sequence target="{startPanel}">
    <Dissolve duration="500" alphaFrom="1" alphaTo="0" />
    <SetPropertyAction property="visible" value="false" />
  </ Sequence >
</Transition>
```

If you are a true geek, you might still be asking yourself how this solves the problem. As it turns out, when moving from the “from” state to the “to” state, the state mechanism is smart enough to be able to inspect the effect and notice that the visible property is being set and makes sure that the right thing happens.

## Components

After your first week with Flex, you will find yourself wanting to break your application into pieces. This is done through components.

Flex components can be defined in ActionScript or in MXML (remember.. it’s all the same thing under the hood!). For visual components, it is often easiest to define components in MXML.

One way to think of components is that it is a way of creating new tags by composing other tags together. Let’s look at a quick example:

### MyApp.mxml

```
<Application>
  <Script>
    function initStartPanel() {...}
    function initDetailPanel() {...}
  </Script>
  <HBox>
    <Panel id="startPanel" creationComplete="initStartPanel()">
      ...
    </Panel>
    <Panel id="detailPanel" creationComplete="initDetailPanel()">
      ...
    </Panel>
  </HBox>
</Application>
```

In the above application, I have an HBox with two panels. Each of these panels has implementation code and may end up getting pretty messy once my application is complete.

I can separate these out into components by simply creating three files that look like this:

### MyApp.mxml

```
<Application>
  <HBox>
    <StartPanel id="startPanel" />
    <DetailPanel id="detailPanel" />
  </HBox>
</Application>
```

### StartPanel.mxml

```
<Panel creationComplete="initStartPanel()">
```

```
<Script>
    function initStartPanel() {...}
</Script>
...
</Panel>
```

### **DetailPanel.mxml**

```
<Panel creationComplete="initDetailPanel()">
    <Script>
        function initDetailPanel () {...}
    </Script>
    ...
</Panel>
```

Notice that in this new world, the implementation for the two panels is cleanly separated out into their own files.

## **Conclusions**

Well, this is a lot to cover. Hopefully, you've learned some concepts that will make it easier for you to get started as you learn about Flex. And hopefully you've learned some tips that will make it easier for you to work with other developers on Flex projects.

Happy building!