Flex Mobile IN ACTION

SAMPLE CHAPTER

Jonathan Campos

MANNING

Flex Mobile in Action

By Jonathan Campos

Chapter 2

brief contents

2

Get going with Flex Mobile

This chapter covers

- Starting a Mobile Flex project
- Creating views
- Using the `ViewNavigator`
- Running your application
- Creating new device configurations
- Pulling data from Rotten Tomatoes AS3 API
- Customizing the `ActionBar`

It's time to dive headfirst into Flex Mobile development. Armed with knowledge of ActionScript, MXML, and Flex, you'll start making your application. In this book you'll be creating an application using the Rotten Tomatoes API. For those who don't know what Rotten Tomatoes is, it's a wonderful website that provides movie details and reviews available in a variety of browsing or search options.

> **Rotten Tomatoes API**
>
> To see a sample of the data you'll be using, feel free to visit the Rotten Tomatoes site at http://www.rottentomatoes.com/. To pull data from the Rotten Tomatoes API, you'll need to have an API key. Make sure to sign up for a developer's account to get an API key to the Rotten Tomatoes API at http://developer.rottentomatoes.com/.
>
> If you skipped chapter 1, full instructions on how to get a developer account are there.

Using the data provided by the Rotten Tomatoes API, you'll use the List/Details user experience paradigm. Even if you haven't heard of the List/Details term, I can promise you that you already know it; many mobile applications are designed using this pattern.

The idea is that you have a list of items, and after selecting a particular item in the list, you are then given details on the selected item. Figure 2.1 diagrams the List/Details user experience paradigm.

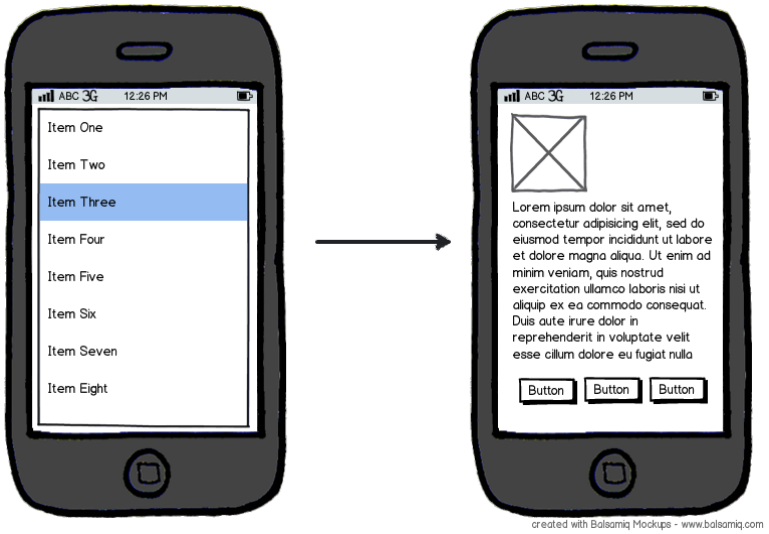

Figure 2.1 List/Details user experience

By starting the application as outlined in this chapter, you'll have the groundwork for a much larger application that can run on a variety of devices. This application example, spread over many chapters, will show you how to navigate a mobile application, persist data effectively, and react to a variety of screen sizes. In this chapter you'll create the beginnings of the application, learn to navigate through the `ViewNavigator`, pull data from Rotten Tomatoes, and add in the basic views for the application.

2.1 Starting up your application

When creating a Flex application the starting point is always a specific Flex class called `Application`. For mobile development Adobe added a few more application-type classes to fit the mobile paradigm.

The first is `ViewNavigatorApplication`, an `Application` that includes a single `ViewNavigator`—we'll discuss the `ViewNavigator` more later on—to push and pop views for navigation. The second new `Application` subclass, and the one you'll use to start your Rotten Tomatoes application, is the `TabbedViewNavigatorApplication`. This `Application` subclass is similar to `ViewNavigatorApplication`, except instead of supporting just one `ViewNavigator` and its single stack of views, `TabbedViewNavigatorApplication` supports multiple `ViewNavigators`, allowing the

user to access the various `ViewNavigator`s by selecting from their respective tabs on screen. As I showed in the previous chapter, you can create this `Application` subclass totally in code, or you can use the Flash Builder IDE to help guide you through the setup process. In this section you'll create your application using Flash Builder.

2.1.1 Creating a TabbedViewNavigatorApplication

Using Flash Builder you can create a new Flex Mobile project and select Tabbed Application to create the `TabbedViewNavigatorApplication` subclass.

Start this process by selecting File > New > Flex Mobile Project, as shown in figure 2.2. You can see how by going through File > New or right-clicking in the Package Explorer you can create a new Flex Mobile Project. Once you create a new Flex Mobile project, the Flash Builder IDE will guide you through the setup process.

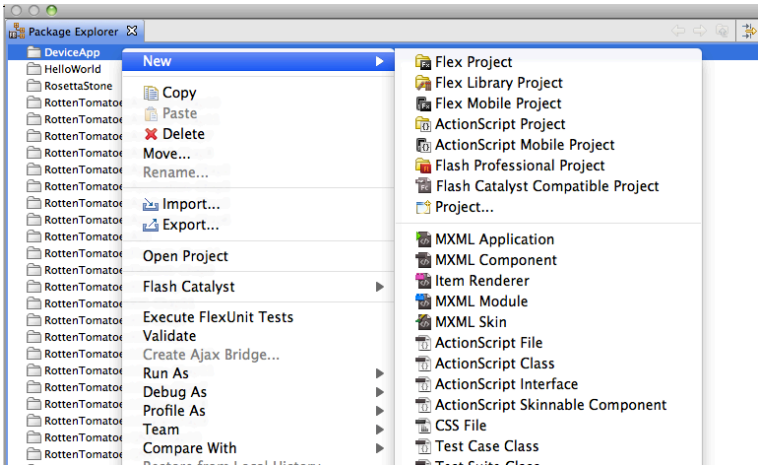

Figure 2.2 Start a new Flex Mobile project.

As Flash Builder leads you through the steps necessary to create a new Flex Mobile application and creates the initial files, you need to set the name of your application and the location to save it to (see figure 2.3).

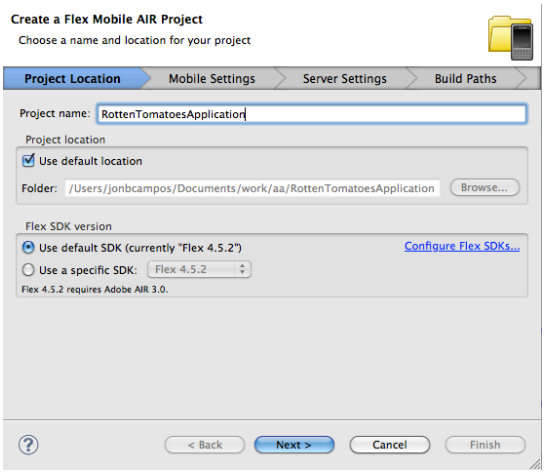

Figure 2.3 Naming your application

After entering the name of your application—use `RottenTomatoesApplication`—and deciding where to save it, click Next and select Tabbed Application in the Application Template section, as shown in figure 2.4.

In this section it's easy to create tabs for your application. You can see in figure 2.4 that I created three tabs named Tab 1, Tab 2, and Tab 3. These values are completely arbitrary at this point because you'll be making changes later to these values directly in your code. As with every option you select at this point, everything can be adjusted in code later on; all of your selections create a bit of code to make your life easier.

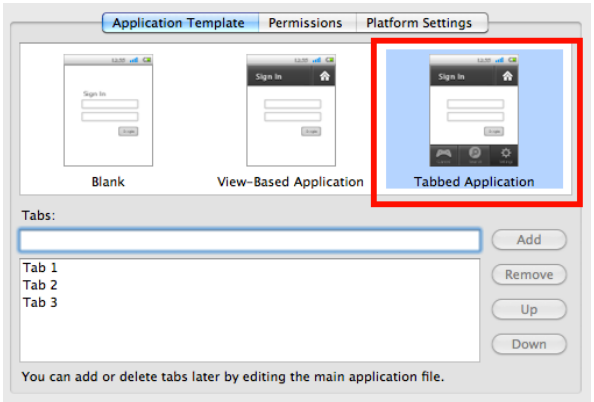

Figure 2.4 Select Tabbed Application.

Once you've made your tabs, click Finish and complete the setup dialog box. The resulting code, or code that you can enter manually to create the exact same effect as using the IDE, is shown in the following listing.

Listing 2.1 Main application—RottenTomatoesApplication.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:TabbedViewNavigatorApplication                    <-- Main application tag
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    applicationDPI="160">
<s:ViewNavigator label="Tab 1" width="100%" height="100%"
    firstView="views.Tab1View"/>                     <-- Tab 1 ViewNavigator
<s:ViewNavigator label="Tab 2" width="100%" height="100%"
    firstView="views.Tab2View"/>                     <-- Tab 2 ViewNavigator
<s:ViewNavigator label="Tab 3" width="100%" height="100%"
    firstView="views.Tab3View"/>                     <-- Tab 3 ViewNavigator
<fx:Declarations>
<!-- Place non-visual elements (e.g., services, value objects) here -->
</fx:Declarations>
</s:TabbedViewNavigatorApplication>
```

{chap 2 code}/src/RottenTomatoesApplication.mxml

Notice that the `TabbedViewNavigator` holds an array of `ViewNavigator`s, one per tab that you intend to create, and each `ViewNavigator` contains its own first view. The `label` property on the `ViewNavigator` controls the button labels on the main `TabNavigator` (see figure 2.5). Finally, the three initial views are created to hold the user interaction components, each set with the `firstView` property.

2.1.2 The views

Flash Builder generates individual views for each tab you created. From figure 2.4 you know that you created an application with three tabs. The following code shows those three views, one code segment per tab. The three resulting views—`Tab1View`, `Tab2View`, and `Tab3View`—are so similar that it almost feels wasteful to give the code for each, but just so that all the code is documented, I'll show the code for each view:

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        title="Tab 1">                               <-- View's title
    <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>
</s:View>
```

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        title="Tab 2">
    <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>
</s:View>
```

View's title

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        title="Tab 3">
    <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>
</s:View>
```

View's title

Each of the view classes is unique, and any further changes that you make to them will only add to their uniqueness, but currently the only thing that's different between each view is the title. The `title` property on the `View` component is used as the title of the `ActionBar` component. In figure 2.5 you can see the running application including the `ActionBar`, the `TabBar`, and the `View` component.

Right now if you were to run the application, with your three views, the resulting application would look like figure 2.5.

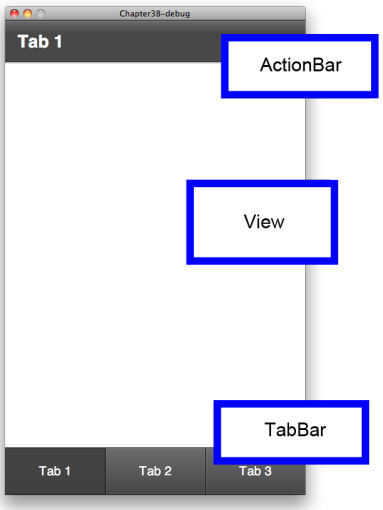

Figure 2.5 Current running application

You've now created a working application that you'll continue to expand on. As I stated earlier, currently the application only allows you to navigate through the three different tabs, but soon you'll be adding much more complex functionality. Next, we'll look at all the different ways to run the application in the desktop simulator and on the devices you currently own.

2.2 Running your application

Before going too far it's important to know how to run and debug your application so that you can test the application in development. With the ability to run your application you can see the state of your programming in action. You can also pause the application and view variable specifics at runtime, ensuring your application's values are as expected.

You can run, debug, and deploy your application using the command line or other IDEs other than Flash Builder, but when I show examples of how to use an IDE to run your application, we'll be looking at examples from Flash Builder because it's currently the most widely used IDE. In this section we'll walk through the steps necessary to run your application on the desktop simulator for quick testing and then move on to running the application on either your Android, QNX (BlackBerry), or iOS device—or all three.

2.2.1 Desktop run/debug configurations

Now that you have an application—simple or not—you'll want to run it to ensure that your application looks and acts the way you intend. The easiest way to test your application is to use the mobile simulator provided by the ADL tool. To be clear, even if you don't have a mobile device, you can run and test your application effectively, although when you release your application I'd highly recommend testing it on as many devices as possible.

> **ADL**
>
> ADL stands for AIR Debug Launcher. This tool, provided by the Adobe Open Source Stack, is useful to run and debug AIR applications on your desktop. Packaged within Flash Builder and the Flex SDK, it's launched each time you run an application on the desktop. One caveat is that you can have only one instance of the ADL tool running at a time. Just make sure to close the ADL tool each time you've finished testing.

Although running the simulator is fast and easy, and it allows you to quickly customize the size of the device you're simulating, there are some downsides to using it. The main issue is that within the simulator you don't get any of the device's native features while testing, such as gestures, multitouch, cameras, and so on (a partial list of not-included features is shown in table 2.1). Any additional APIs that you create using native extensions (see chapter 8) are most likely not available.

Table 2.1 Available and unavailable device capabilities from ADL

Feature	Availability
Camera	No
Camera roll	No
Accelerometer	No
Geolocation	No
Home, Search, Menu, and Back keys	Yes
StageWebView	Yes
Microphone	No
Multitouch	No, unless you have a monitor that supports multitouch input
Gestures	No, unless you have a trackpad that supports gestures
Screen orientation changes	Yes
Email	Yes
Text messages	No
Phone calls	No
SQLite databases	Yes
Caching (via local shared object)	Yes

One big benefit of testing on the device is that you can see how long things take to render and compute. Simulating is deceptive, because it runs an application on a machine more powerful than the actual device and with connection speeds faster than what your users will receive over the air. When testing on the device you may see visual slowdowns and services taking a long time to return. Performance gains and losses aren't exposed in the simulator because of the power of the hardware.

To simulate the orientation changes and hardware buttons of the Android device in the ADL tool, a few shortcut keys are provided within ADL to simulate these device actions (see table 2.2).

Table 2.2 ADL shortcut keys

Hardware feature	Details	Windows	Macintosh
Back button	Simulates hitting the Back button	Ctrl-B	Command-B
Search button	Simulates hitting the Search button	Ctrl-S	Command-S
Menu button	Simulates hitting the Menu button	Ctrl-M	Command-M
Rotate left	Simulates rotating the device left (counterclockwise)	Ctrl-L	Command-L
Rotate right	Simulates rotating the device right (clockwise)	Ctrl-R	Command-R
Close button	Closes the app	Ctrl-W	Command-W

Two paths are available to get started when you need to run or debug an application. Both lead to the Run Configurations, where you'll customize how your application is simulated.

The first method is found in the toolbar (see figure 2.6). This quick access to run/debug will immediately run/debug the last project run, or you can select the exact project through the dropdown. If you have no previously run options, you'll need to choose Run Configurations to configure the run options for your application.

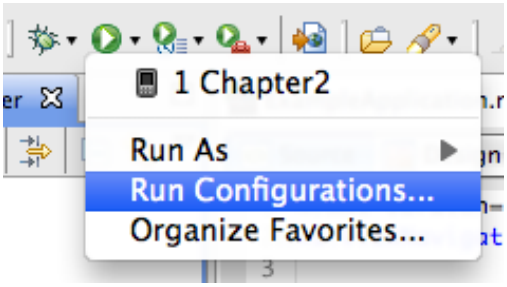

Figure 2.6 Toolbar Run dropdown

The second path is to right-click the application file that you want to run and navigate down to Run As/Debug As > Mobile Application (see figure 2.7).

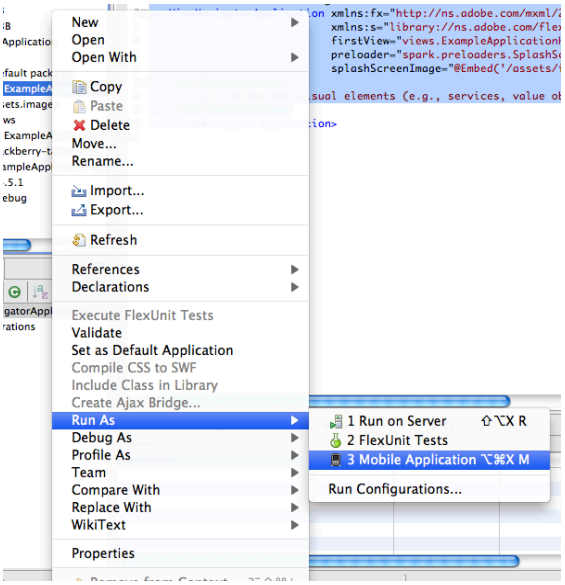

Figure 2.7 Run Configurations from Package Explorer

Either way you go, you should now see the Create, Manage, and Run Configurations dialog box (see figure 2.8).

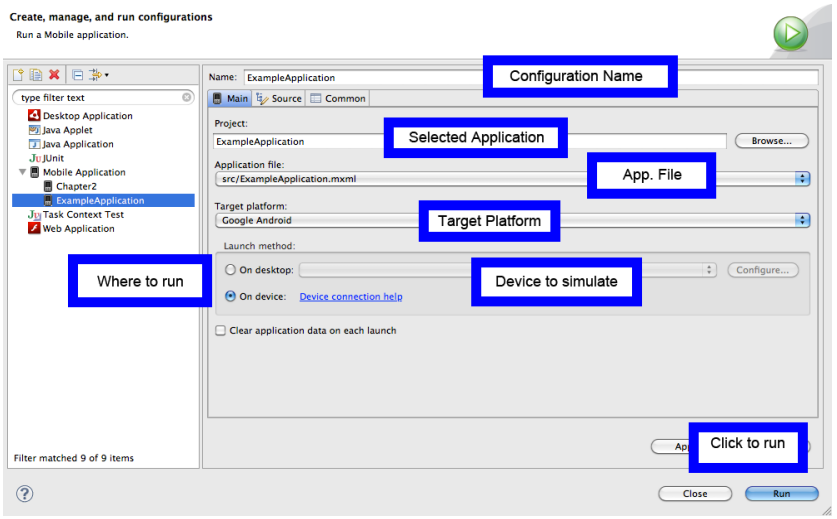

Figure 2.8 Create, Manage, and Run Configurations dialog box

From this dialog box, you can customize how the ADL tool will run your application on the desktop. Within this dialog box, you can name your run configuration for easy access in the future.

Over time, you'll probably find it helpful to create multiple run configurations for various device configurations and name each configuration uniquely, like "Mobile App Evo" and "Mobile App iPad" for Evo and iPad configurations, respectively. This will save you development time by not having to continuously change the run configuration.

To adjust your run configurations, first select the project to run and the application file to launch. Then select which target platform you'll be simulating from, where to launch the application, and finally which device to simulate. Once you've made all of your selections, click Run, and the ADL tool will run your application.

By simulating your application on the desktop, you can click around and interact with your application as you could on your device. If you were debugging your application, any ActionScript breakpoints that you set would pause your application and provide details on variables and other data at that moment in the application.

When you select devices in the device dropdown configuration list, a device that you want may be missing from the detail list. In the next section we'll look at how to add devices to your device list for your simulator.

2.2.2 Adding a device configuration

If the device that you want to simulate isn't in the dropdown list, you have two options: either import the required device configuration or, if the configuration isn't in the import list, add the custom configuration. Both of these options are available in the Device Configurations dialog box (see figure 2.9), which is accessible by clicking the Configure button next to the device selection dropdown.

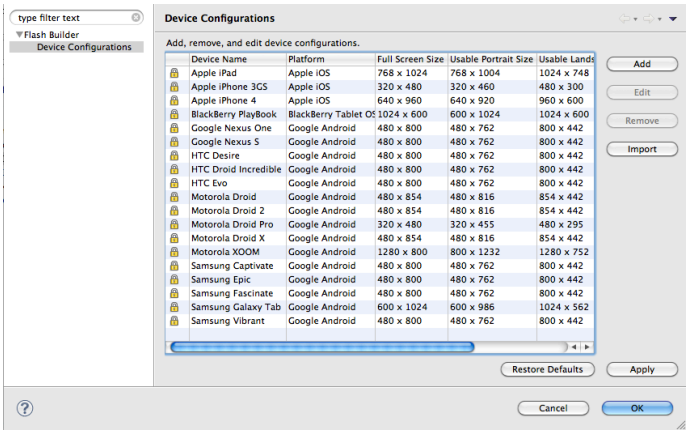

Figure 2.9 Device Configurations dialog box

In the Device Configurations window you can view the available devices or import new device profiles that Adobe adds to Flash Builder. If the device you want to add isn't in the import list, you can click the Add button to quickly add a new device by entering the screen resolution and DPI.

Because you're trying to reach as many devices as possible, you'll want to also run on the Barnes and Noble Nook. The Nook is an Android-based device that isn't included by default in the device list, so you need to add it to your device list. To add this device, you need to click Add in the Device Configurations dialog box.

In the resultant dialog box (see figure 2.10), you'll specify a few parameters and then you'll be able to accurately simulate the Nook on your desktop.

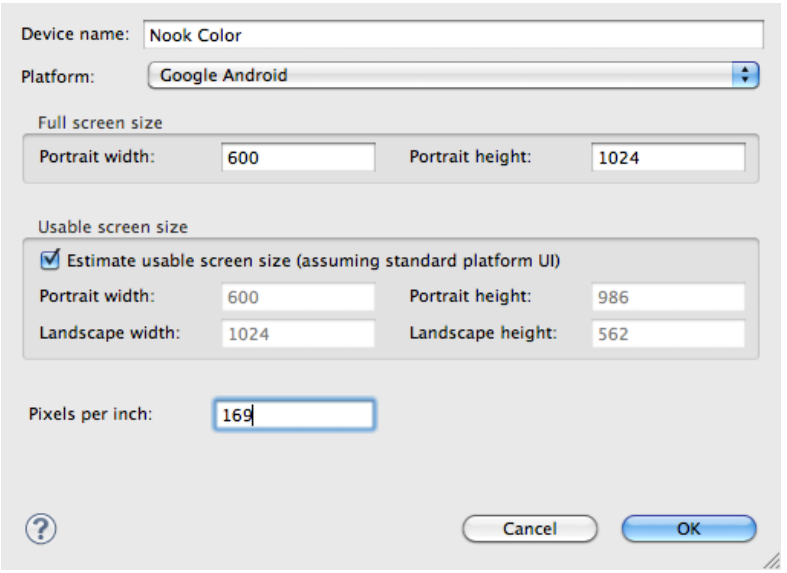

Figure 2.10 Adding a device configuration

First, name the device `Nook Color`, and then set the platform to Google Android. Then enter the portrait width and height—these are the screen dimensions in portrait orientation. By default Flash Builder will adjust the usable screen size for a standard platform UI. What this means is that the actual size of your application is the total width and height minus the size of the status bar and any other platform UI. If the device has a nonstandard status bar, then you can make these adjustments by deselecting Estimate Usable Screen Size and adjusting the width and height as necessary. Finally, enter the pixels per inch for the Nook. With all of your settings configured, you can click OK and start simulating the Nook for development.

With the ability to add any custom device, you have every possible device at your fingertips. Now select to run your applications based on your newly developed device, and continue testing for great results.

2.2.3 Device run/debug configurations

If you own an Android-, iOS-, or QNX (BlackBerry)-based device, you can also connect directly to the device and debug/run from the device with ease. Each device takes a slightly different set of steps to connect and deploy to the device based on the manufacturer's requirements. The following three links outline the connection method for Android, iOS, and QNX (BlackBerry) devices, respectively:

Android—http://www.adobe.com/devnet/air/articles/packaging-air-apps-android.html

iOS—http://www.adobe.com/devnet/air/articles/packaging-air-apps-ios.html

QNX (BlackBerry)—http://www.adobe.com/devnet/air/articles/packaging-air-apps-blackberry.html

Once you're connected, the process to test on the native devices is simple. One big benefit of testing on the device is that you can see how long it takes to run your application on the end user's device.

Simulating is deceptive

Rendering, computing, and pulling data on a machine more powerful and with faster connection speeds than what your users will receive over the air won't expose where your application needs work. My recommendation is to test on the desktop first and then test mobile-specific features and performance on the device.

I know that felt like a lot of work, but the hard parts are over. From here on out you don't need to keep setting up your project; you're ready to start programming and building out the functionality of your application. In the next section you'll create the first of the list/details views.

2.3 Building your first application views

Adding more tools to your belt, you'll now become a master of the `ViewNavigator` with the ability to navigate through your application by adding and removing views. In this section you'll start adding three important views to your application. These will stand as the bedrock of your larger application. You'll create your main menu, move to a list of movies, and then see details based on a specific selected title.

2.3.1 Navigating your application with the ViewNavigator

Before adding and removing views, you need to understand how the `ViewNavigator` works. As we've discussed, the `TabbedViewNavigatorApplication` is a subclass of the

Application class, which is required as the starting point for any Flex application and holds an array of ViewNavigator components. The ViewNavigator holds a stack of views and manages the addition and removal of views from the stage. When managing the stack of views, the ViewNavigator performs three functions: adding/removing views, cleaning up old views, and remembering the list of views in the stack. To illustrate the example we'll work with a simple list of views, the same views you'll use in your application. The three views include a main menu, a list of movies, and details on a selected movie title (see figure 2.11).

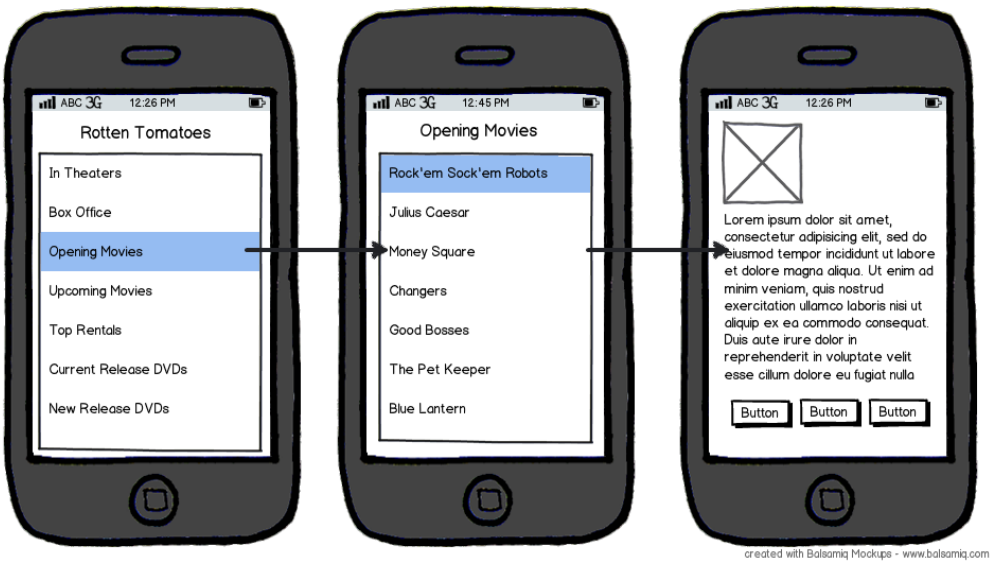

Figure 2.11 Our three views

At any given time when using the ViewNavigator, only a single view is active at a time. What happens to the other views? By default, the ViewNavigator destroys any views that aren't currently in use. The benefit to destroying unused views is that the application no longer has to maintain the data for the view. But the problem with this approach is that the way that the view looks—any data that was input, any sliders moved, your position in a list—will be completely lost when you leave the view. If you navigate back to a view that was destroyed, you'll have to set back anything you've changed so that for the user the view looks unchanged.

DESTRUCTION POLICY

You do have the option to stop a view from being destroyed by working with the destructionPolicy:

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark" title="Tab 1"
        destructionPolicy="never" >
    <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>
</s:View>
```

A view isn't destroyed

If you have a view that takes a long time to create and would best be stored in memory, then you can turn off this destruction policy on a view-by-view basis. This prevents the view from being destroyed and preserves the entire view state. I recommend not leaning on this technique because you could quickly hurt the performance of your application by keeping too much in active memory. For your application, you won't keep this code change.

FIRST VIEW

As stated when you set up your application, the view set to the `firstView` property in `ViewNavigator` is the first view shown when the `ViewNavigator` is created:

```
<s:ViewNavigator label="Tab 1" width="100%" height="100%"
firstView="views.Tab1View"/>
```

In this case you want the `firstView` to be the main menu so that on startup you see the main menu (see figure 2.12).

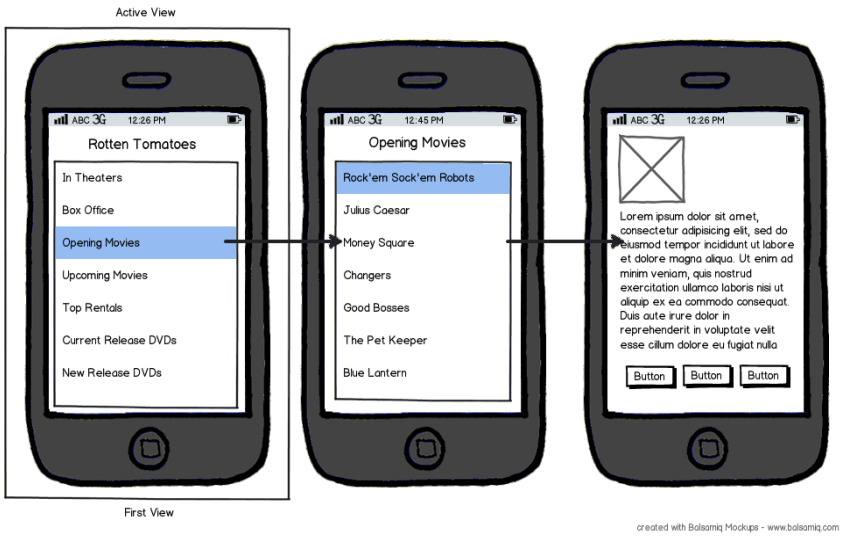

Figure 2.12 First view

The first thing to do is to navigate to the next view when a menu item is selected. To do this you push a view onto the view stack (see figure 2.13). The push function on the `ViewNavigator` includes four parameters:

```
pushView(viewClass:Class, data:Object=null, context:Object=null,
transistion:ViewTransitionBase=null);
```

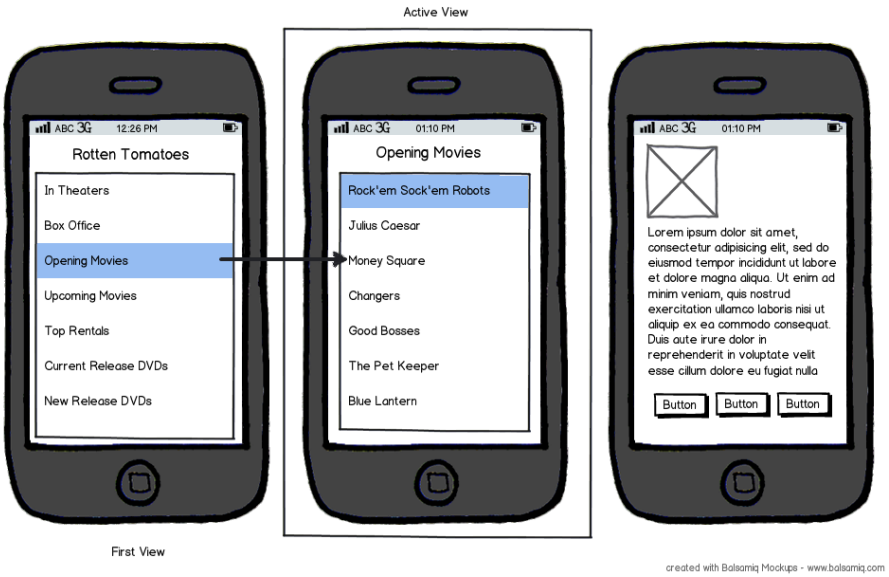

Figure 2.13 Pushing a view

The first and only required parameter is the class that needs to be created; please notice that this isn't an instance of the view you desire to create, just the class. The reason that the method wants a class rather than an instance is so that the `ViewNavigator` controls when a view is created or destroyed.

The second parameter is the generic `data` object. Any data that you set will automatically be passed to the newly created view and set on the view's `data` property. This is a great way to pass data to the next view, giving the view information that may be used to construct the new view.

The third parameter is another generic parameter called `context`. Although any form of data can be passed in the context and retrieved by using the generated view's `context` property, the idea is to use this parameter to pass information to the generated view that will provide information about where the new view came from. For example, in your Rotten Tomatoes application, you may want to pass to the movie title details view information in the context that alerts the details that a specific title is in theaters or on DVD. With this bit of

additional data, you can change the details view to reflect this information, making it easier to know where the generated view is coming from.

The fourth and final parameter is the `transition` effect. By default, the `ViewNavigator` will slide new views onto the screen when you push a new view onto the stack. If you'd like to use another effect to show when the view is added—even a custom effect—you can set an instance of this effect in the fourth parameter. Any effect used for a view transition must be a subclass of the `ViewTransitionBase`. The `ViewTransitionBase` is a special effect class that's optimized to move a view onto a `ViewNavigator`.

Pushing views onto your `ViewNavigator` is great, but at some point you have to navigate back down the stack and see where you were; this is referred to as *popping views*. We're going to look at the three different ways to pop a view from the stack.

POPVIEW

Using the `popView` method (see figure 2.14) is the easiest and most basic way to remove your current view:

```
popView(transition:ViewTransitionBase=null)
```

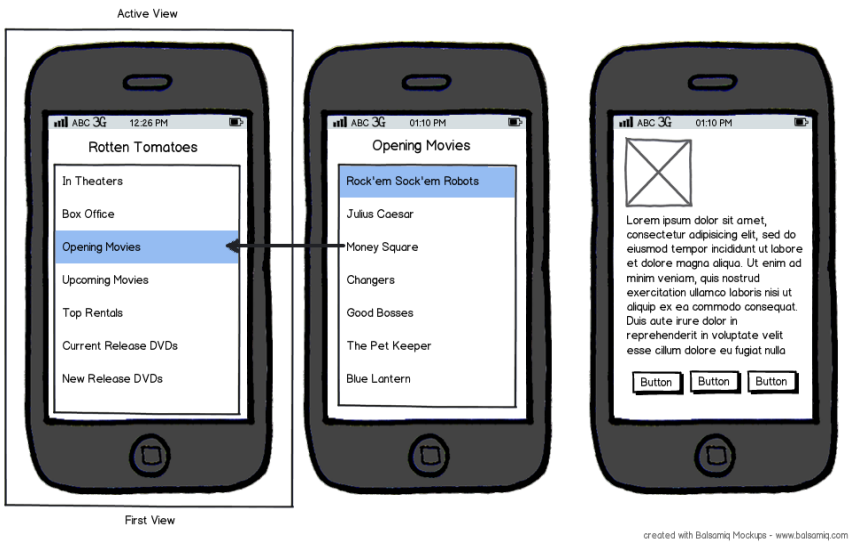

Figure 2.14 `popView`

As indicated by the name, `popView` pops the current view from the `ViewNavigator` and returns you to the last view in the stack. If you don't specify a transition, then the default

slide-to-the-left transition is used. The main thing to take away is that `popView` removed only one view from the stack, the current view.

POPTOFIRSTVIEW

When `popView` isn't good enough and you need to immediately move back to the `firstView`, it's time to use `popToFirstView` (see figure 2.15).

popToFirstView(transition:ViewTransitionBase=null)

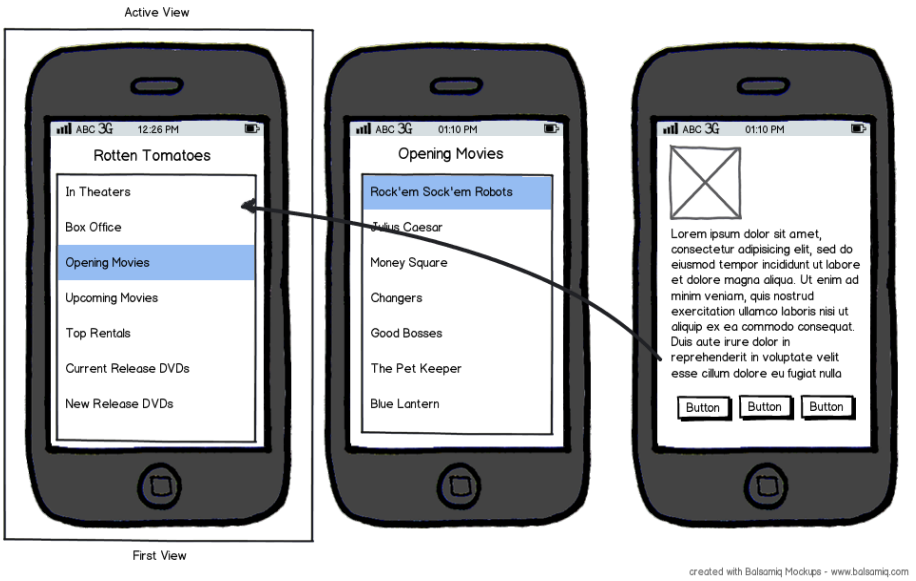

Figure 2.15 `popToFirstView`

This ultra-helpful function pops off all the views in the stack and returns the user to the first view.

POPALL

Finally, the last way to pop views is to remove all the views from the stack in one quick function (see figure 2.16):

popAll(transition:ViewTransitionBase=null)

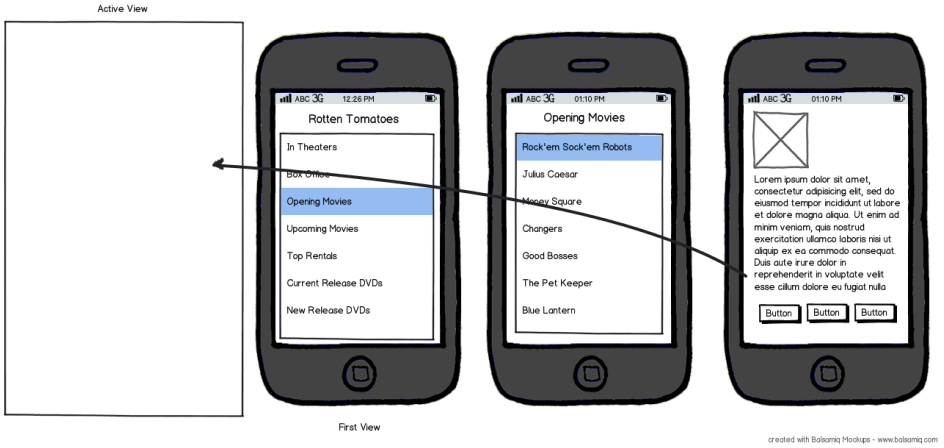

Figure 2.16 `popAll`

The `popAll` method removes all the views including the `firstView`. After calling this method you'll need to push a new view onto the `ViewNavigator` or your application will have no content.

With the ability to push and pop views to the `ViewNavigator` you now can take control of the navigation elements of your application.

2.3.2 Providing context with the ActionBar

You've already seen the `ActionBar` when you ran the application but probably didn't realize its helpfulness. The `ActionBar` is a common mobile user interaction component that's helpful for user input, navigation, and view context. Currently, you're only using the `ActionBar` to provide view context by showing the title of the selected view (see figure 2.17).

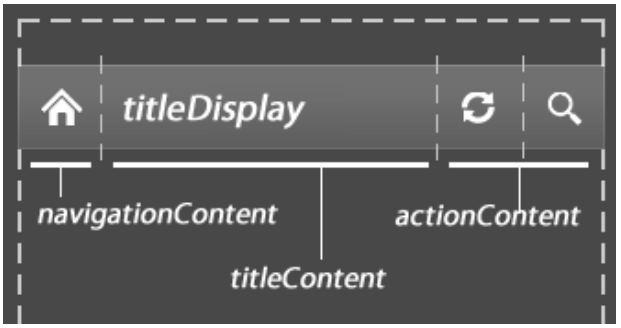

Figure 2.17 The ActionBar

The ActionBar also includes a few different modes that dictate how the ActionBar is laid out in the view.

HIDE THE ACTIONBAR

One option is to hide the ActionBar:

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark" title="Tab 1"
        actionBarVisible="false"> <-- Remove ActionBar
    <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>
</s:View>
```

If you want to remove the ActionBar from a view, set the actionBarVisible property to false. By default this property is true.

OVERLAY THE ACTIONBAR

The next option is to overlay the ActionBar on the view's contents:

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark" title="Tab 1"
        overlayControls="true"> <-- overlayControls property
    <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>
</s:View>
```

If you want to have the ActionBar overlay or float over the view visual components, then you need to set the overlayControls property to true (see figure 2.18). By default this property is false.

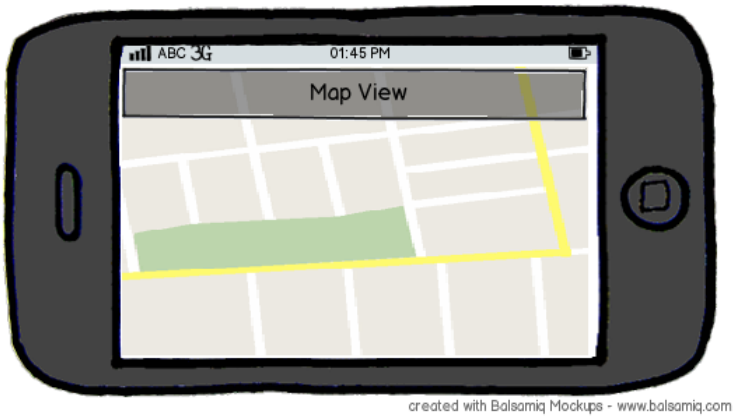

Figure 2.18 Controls overlaid

The `ActionBar` is connected to the `ViewNavigator`, and as such when new views are pushed and popped from the `ViewNavigator`, the title and content of the `ActionBar` are set to reflect the current view's content. If you want to override the visual appearance of the `ActionBar`, then you need to change the skin of the `ActionBar` as part of the `ViewNavigator` skin.

The one section in the `ActionBar` that's special is the `titleContent`. By default you can set the text shown in the `titleContent` by setting the `title` property on the `ActionBar`. If you set different content to the `titleContent`, then the `title` property is ignored, because the default label will be overridden by whatever `titleContent` you've set.

With the intricacies of the `ActionBar` explained, you can now move forward with the customization of your views and create the visual layout of the application.

2.3.3 Updating the main application

The first thing you need to do is update the main application and ensure that it includes two tabs, one for your browse menu and one for your search functionality:

```
<?xml version="1.0" encoding="utf-8"?>
<s:TabbedViewNavigatorApplication
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:ViewNavigator label="Browse" width="100%" height="100%"
        firstView="views.BrowseView" />
    <s:ViewNavigator label="Search" width="100%" height="100%"
        firstView="views.SearchView" />
    <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
```

Browse `ViewNavigator`

Search `ViewNavigator`

```
    </fx:Declarations>
</s:TabbedViewNavigatorApplication>
```

Right now, your application can't be compiled. Don't worry, that's expected. You should be receiving errors for the missing `views.BrowseView` and `views.SearchView` classes. In the following sections you'll create the two missing views.

2.3.4 Creating the browse view

The first new component you'll make is `BrowseView.mxml`, which serves as the central access point for all of your navigation. As shown in figure 2.19, `BrowseView` is a simple list that provides access to the variety of lists that you'll include for your users.

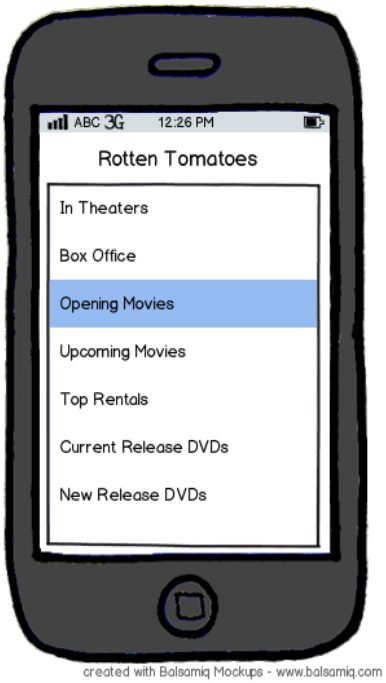

Figure 2.19 `BrowseView.mxml`

Using Flash Builder it's easy to create the `BrowseView` using the same New menu that we used to create the Flex Mobile Project. Using the Package Explorer window, right-click the views package and select New > MXML Component (see figure 2.20).

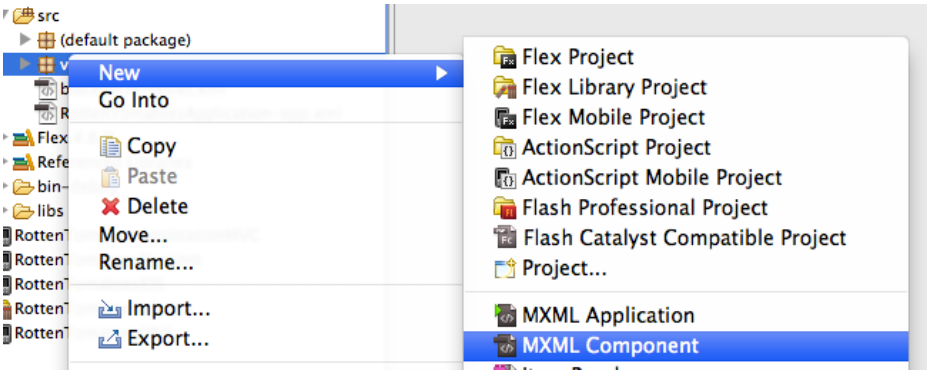

Figure 2.20 New MXML component

This will bring up a New MXML Component dialog box, where you can name the new component, choose the package for the new component, and set the component to base your new component on, in this case the `View` class (see figure 2.21).

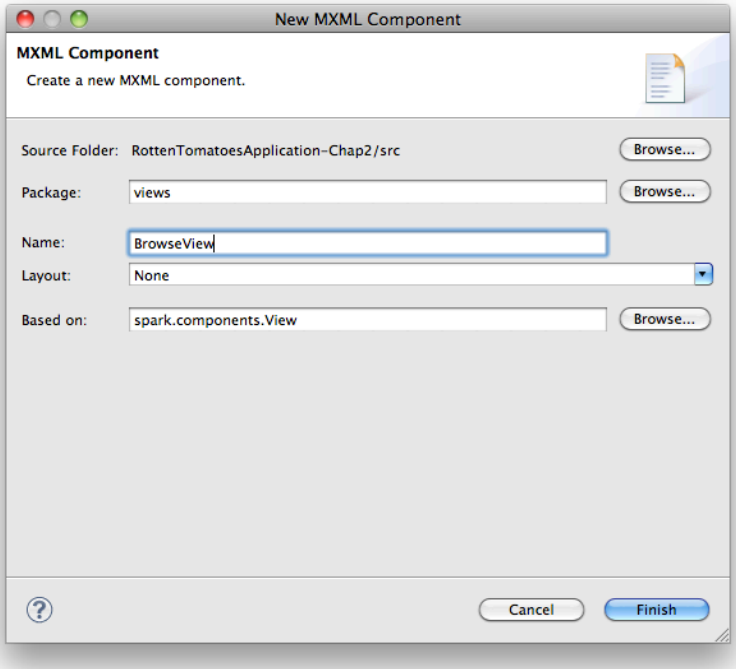

Figure 2.21 New MXML Component dialog box

This will result in the default code for your `BrowseView`:

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        title="BrowseView">
    <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>
</s:View>
```

With the basic component ready, you can add the list component for your menu, as shown in the following listing; the title for the view has also been changed from BrowseView to Rotten Tomatoes.

Listing 2.2 BrowseView.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    title="Rotten Tomatoes">
    <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>
    <s:List width="100%" height="100%"
            labelField="label">
        <s:dataProvider>
            <s:ArrayList>
                <fx:Object label="In Theaters"/>
                <fx:Object label="Box Office"/>
                <fx:Object label="Opening Now"/>
                <fx:Object label="Coming Soon To Theaters"/>
                <fx:Object label="Top Movie Rentals"/>
                <fx:Object label="Currently on DVD"/>
                <fx:Object label="New To DVD"/>
                <fx:Object label="Soon To DVD"/>
            </s:ArrayList>
            </s:dataProvider>
    </s:List>
</s:View>
```

List component

List's data

{chap 2 code}/src/views/BrowseView.mxml

The code for the visual layout of the `BrowseView` is simple, just a list with some data. The list component is a Flex component that visualizes data using `itemRenderer`s. The data represented in the list component is set with the `dataProvider` property. You've set the list to take the entire view, setting the `width` and `height` to `100%`. The next step will be to add

the ActionScript methods that will control the user interaction of the visual elements; in this case you want to move views when a user selects an item in a list (see the following listing).

Listing 2.3 BrowseView.mxml with change handler

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark" title="Rotten Tomatoes">
    <fx:Script>
    <![CDATA[

    import spark.events.IndexChangeEvent;

    protected function list1_changeHandler(event:IndexChangeEvent):void
    {
    var list:List = event.target as List;
    var selectedItem:Object = list.selectedItem;
    navigator.pushView(ListView, null, selectedItem.label);
    }

    ]]>
    </fx:Script>
    <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>
    <s:List width="100%" height="100%"
            labelField="label"
            change="list1_changeHandler(event)">
        <s:dataProvider>
            <s:ArrayList>
                <fx:Object label="In Theaters"/>
                <fx:Object label="Box Office"/>
                <fx:Object label="Opening Now"/>
                <fx:Object label="Coming Soon To Theaters"/>
                <fx:Object label="Top Movie Rentals"/>
                <fx:Object label="Currently on DVD"/>
                <fx:Object label="New To DVD"/>
                <fx:Object label="Soon To DVD"/>
                </s:ArrayList>
        </s:dataProvider>
    </s:List>
</s:View>
```

List change handler

Get the selected item

Push the new view

{chap 2 code}/src/views/BrowseView.mxml

To do this you need to listen for the `change` event from the list component. In the handler function you get a reference to the list component by referencing the event's target parameter; then you get the item that was selected on the list. Gathering the selection

information, you need to pass the selected browse option to the next view. By using the `context` parameter you can tell which `ListView` you'll be creating and what sort of data to show.

> **Where did "navigator" come from?!**
>
> Components within Flex, such as the `View` component, are complex components with many subcomponents, methods, properties, and references to other components. In this case you're using a reference to the main `ViewNavigator` through the `navigator` property. The main `ViewNavigator` includes the `pop` and `push` methods, not the `View` component itself.

Right now, there's no `ListView`, and you'll get an error that stops you from running the application. In the next section you'll rectify this error by creating the `ListView` component and show exactly how you use the `context` value.

For now, you can comment out the offending line to run your application:

```
. . .
var list:List = event.target as List;
var selectedItem:Object = list.selectedItem;
//navigator.pushView(ListView, null, selectedItem.label);
. . .
```

2.3.5 Creating the list view

When a user selects a specific list, you need to navigate to the `ListView` and populate the view with the selected list of movies (see the following listing).

Listing 2.4 ListView.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        title="ListView">
    <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>
    <s:List width="100%" height="100%" id="list">
        <s:itemRenderer>
            <fx:Component>
                <s:IconItemRenderer
                    iconField="thumbnailPoster"
                    iconWidth="61" iconHeight="91"
                    labelField="title"/>
            </fx:Component>
        </s:itemRenderer>
    </s:List>
```

List component

List's ItemRenderer

```
    <!-- error label -->
    <s:Label id="errorLabel" width="100%"
          backgroundColor="#000000" color="#FFFFFF"
          paddingBottom="20" paddingTop="20"
          textAlign="center" verticalCenter="0"
          visible="false" includeInLayout="false"/>

</s:View>
```

Label overlay to display any errors

{chap 2 code}/src/views/ListView.mxml

The `ListView` is a simple view that includes only a single list component with a custom `itemRenderer` to creatively display the list data. Your `itemRenderer` is the mobile-optimized complex renderer `IconItemRenderer`. The `IconItemRenderer`—provided by the Flex SDK—includes an icon, two text labels stacked vertically, and a decorator image on the right side. As you already know, by setting properties such as the `labelField`, `labelFunction`, `iconField`, `iconFunction`, `messageField`, and `messageFunction`, you can control the fields from the data object that the list is visualizing.

With the basic view laid out, you now need to add the service to your `ListView`. For this you'll use the `RottenTomatoesService` component, provided by the RottenTomatoesAS3 Library and available for download from GitHub at https://github.com/jonbcampos/RottenTomatoesAS3.

RottenTomatoesAS3 SWC file

If you're having a hard time finding the SWC file, or are just lazy like me, you can use the following link to download the RottenTomatoesAS3 SWC file.

https://github.com/downloads/jonbcampos/RottenTomatoesAS3/RottenTomatoesAS3.swc

Once you've downloaded the RottenTomatoesAS3.swc, you can include the SWC info in your project's libs directory (see figure 2.22).

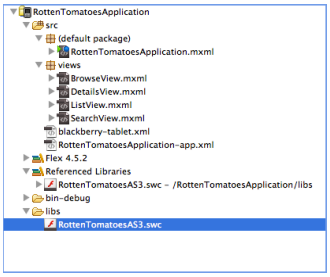

Figure 2.22 RottenTomatoesAS3.swc

To include the service into your view, you need to add the service component into your declarations section for nonvisual elements:

```
. . .
<fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
    <rottentomatoes:RottenTomatoesService id="service"
                apikey="yourApiKey"/>
</fx:Declarations>
. . .
```

Service

Api key input

You'll give the service an `id` so you can reference the service via ActionScript and set the `apiKey` property with your `apiKey` provided by Rotten Tomatoes.

rottentomatoes custom namespace

```
xmlns:rottentomatoes="com.rottentomatoes.*"
```

In Flash Builder you don't have to type the namespaces yourself. If you start typing "RottenTomatoesService," you'll notice that autocomplete gives you options for `RottenTomatoesService`. When you select to create the `RottenTomatoesService`, Flash Builder will automatically add in the new namespace; otherwise, you'd need to add this for your application to compile successfully.

To kick off the service request you need to use a few methods (see the following listing).

Listing 2.5 View with methods stubbed out—ListView.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:rottentomatoes="com.rottentomatoes.*"
        viewActivate="_onView_ViewActivateHandler(event)">
<fx:Script>
<![CDATA[
import com.rottentomatoes.events.RottenTomatoesFaultEvent;
import com.rottentomatoes.events.RottenTomatoesResultEvent;
import spark.events.ViewNavigatorEvent;
import spark.events.IndexChangeEvent;

private function
_onView_ViewActivateHandler(event:ViewNavigatorEvent):void{}

private function
_onService_ResultHandler(event:RottenTomatoesResultEvent):void{}

private function
```

```
_onService_FaultHandler(event:RottenTomatoesFaultEvent):void{}

private function _onList_ChangeHandler(event:IndexChangeEvent):void{}

]]>
</fx:Script>
<fx:Declarations>
<!-- Place non-visual elements (e.g., services, value objects) here -->
<rottentomatoes:RottenTomatoesService id="service"
                                    apikey="yourApiKey"/>
</fx:Declarations>
<!-- list of movie titles -->
<s:List width="100%" height="100%" id="list"
    change="_onList_ChangeHandler(event)">
<s:itemRenderer>
        <fx:Component>
            <s:IconItemRenderer
                iconField="thumbnailPoster"
                iconWidth="61" iconHeight="91"
                labelField="title"/>
        </fx:Component>
    </s:itemRenderer>
</s:List>

<!-- error label -->
<s:Label id="errorLabel" width="100%"
    backgroundColor="#000000" color="#FFFFFF"
    paddingBottom="20" paddingTop="20"
    textAlign="center" verticalCenter="0"
    visible="false" includeInLayout="false"/>

</s:View>
```

{chap 2 code}/src/views/ListView.mxml

The first method responds to the `viewActivate` event. The `viewActivate` event is fired when the view activates after the transition effect completes from the `ViewNavigator`. The reason you use the `viewActivate` event instead of any other event from the view component is because this event signifies when the transition is complete, keeping the transition effect smooth and uninterrupted. The second method is the service `result` handler. This function will set the results of the service to the list component. The third method will alert the user to any error that returns from the service. The final method is the change handler for the list component. This method will later be used to push the `DetailsView`, providing more finite information on a movie title.

You may remember from section 2.3.4 that you passed the `context` property to the `ListView` when pushing it onto the `ViewNavigator` stack. In the `viewActivate` handler,

you'll respond to the `context` property and make a service call based on the `context` (see the following listing).

Listing 2.6 Making a service call—ListView.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:rottentomatoes="com.rottentomatoes.*"
        viewActivate="_onView_ViewActivateHandler(event)">
    <fx:Script>
    <![CDATA[
    import com.rottentomatoes.events.RottenTomatoesFaultEvent;
    import com.rottentomatoes.events.RottenTomatoesResultEvent;
    import mx.collections.ArrayList;
    import spark.events.IndexChangeEvent;
    import spark.events.ViewNavigatorEvent;

    private function
_onView_ViewActivateHandler(event:ViewNavigatorEvent):void{
        var context:String = navigator.context as String;
        title = context;
    service.addEventListener(RottenTomatoesResultEvent.RESULT,
_onService_ResultHandler);
    service.addEventListener(RottenTomatoesFaultEvent.FAULT,
_onService_FaultHandler);
        switch(context){
        case "In Theaters":
            service.getInTheaterMovies();
            break;
        case "Box Office":
            service.getBoxOfficeMovies();
            break;
        case "Opening Now":
            service.getOpeningMovies();
            break;
        case "Coming Soon To Theaters":
            service.getUpcomingMovies();
            break;
        case "Top Movie Rentals":
            service.getTopRentals();
            break;
        case "Currently on DVD":
            service.getCurrentReleaseDvd();
            break;
        case "New To DVD":
            service.getNewReleaseDvd();
```

Add event listeners for Rotten Tomatoes Service

Add event listeners for Rotten Tomatoes Service

Select service based on context

```
            break;
        case "Soon To DVD":
            service.getUpcomingDvd();
            break;
        }
    }

    private function
_onService_ResultHandler(event:RottenTomatoesResultEvent):void{
        list.dataProvider = new ArrayList(event.result as Array);
    }

    private function
_onService_FaultHandler(event:RottenTomatoesFaultEvent):void{
    errorLabel.visible = errorLabel.includeInLayout = true;
    errorLabel.text = event.fault.faultDetail;
    }

    private function _onList_ChangeHandler(event:IndexChangeEvent):void{}

    ]]>
    </fx:Script>
. . .
```

Select service based on context

Set list's `dataProvider`

Display any errors

{chap 2 code}/src/views/ListView.mxml

First, you'll use the view's `context` property to know which service to call and then add event handlers for the result and fault handlers.

The service `result` handler function is simple, setting the result value to the list's `dataProvider` property to display the list of movie titles. The `fault` handler is just as simple, displaying any service faults to the screen with `errorLabel`.

With the `ListView` almost complete, you can start making service calls and seeing movie data being displayed to the screen. Be sure to return to the `BrowseView` and uncomment the push view functionality.

```
. . .
protected function list1_changeHandler(event:IndexChangeEvent):void{
    var list:List = event.target as List;
    var selectedItem:Object = list.selectedItem;
    navigator.pushView(ListView, null, selectedItem.label);
}
. . .
```

At this point I'd recommend you take a second and click around in your app; you'll enjoy playing with the results. But before doing so, you'll want to comment out the line in your RottenTomatoesApplication.mxml that references the nonexistent `SearchView`. In the next

section you'll expand your abilities by adding the `SearchView`, giving users the ability to search through the Rotten Tomatoes library of movies.

2.3.6 Creating the search view

Although the ability to browse for movies based on a variety of predefined lists is great, many of your users will want to be able to search for the exact movie title—the `SearchView` will enable this feature (see the following listing).

Listing 2.7 SearchView.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    title="Search" xmlns:rottentomatoes="com.rottentomatoes.*">
    <fx:Script>
    <![CDATA[
    import com.rottentomatoes.events.RottenTomatoesFaultEvent;
    import com.rottentomatoes.events.RottenTomatoesResultEvent;
    import spark.events.IndexChangeEvent;

    private function _onSearchButton_ClickHandler(event:MouseEvent):void{}

    private function _onList_ChangeHandler(event:IndexChangeEvent):void{}

    private function
_onService_ResultHandler(event:RottenTomatoesResultEvent):void{}

    private function
_onService_FaultHandler(event:RottenTomatoesFaultEvent):void{}

    ]]>
    </fx:Script>

    <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
    <rottentomatoes:RottenTomatoesService id="service"
            apikey="yourApiKey"
            result="_onService_ResultHandler(event)"
            fault="_onService_FaultHandler(event)"/>
    </fx:Declarations>

    <s:titleContent>
        <s:TextInput id="searchInput" width="100%"/>
    </s:titleContent>

    <s:actionContent>
```

Rotten Tomatoes service

Text input

```
        <s:Button label="Search"
            click="_onSearchButton_ClickHandler(event)"/>
    </s:actionContent>

    <!-- list of movie titles -->
    <s:List width="100%" height="100%" id="list"
            change="_onList_ChangeHandler(event)">
        <s:itemRenderer>
            <fx:Component>
            <s:IconItemRenderer
                    iconField="thumbnailPoster"
                    iconWidth="61" iconHeight="91"
                    labelField="title"/>
            </fx:Component>
        </s:itemRenderer>
    </s:List>

    <!-- error label -->
    <s:Label id="errorLabel" width="100%"
             backgroundColor="#000000" color="#FFFFFF"
             paddingBottom="20" paddingTop="20"
             textAlign="center" verticalCenter="0"
             visible="false" includeInLayout="false"/>

</s:View>
```

Call the `search` method

Search list

Show errors

{chap 2 code}/src/views/SearchView.mxml

The `SearchView` is similar to the `ListView` in that it includes a list for the results, a service to retrieve the results, and an `errorLabel` to display if anything unexpected happens. The `SearchView` builds on the `ListView` by including a `TextInput` for the search term and a `Button` for the user to initiate the search function. To make the `SearchView` work, you'll need a few functions to react to user input and search results. After looking at the visual layout, you'll focus on the method implementations.

To build the functionality into the `SearchView`, you need to add some code to your event handlers:

```
import mx.utils.StringUtil;
private function _onSearchButton_ClickHandler(event:MouseEvent):void
{
    errorLabel.visible = errorLabel.includeInLayout = false;
    var term:String = StringUtil.trim( searchInput.text );
    if(term.length>0)
        service.getMoviesByTerm( term );
}
```

Hide any shown errors

Clean up the search string

Call the search method

In this function you want to initiate a search only if there's text in the text input. If text exists, then you want to search for the movie titles. One helpful thing to do is to use the trim

method to remove any leading or trailing spaces so that any extra spaces don't affect your search results. Finally, you want to remove the `errorLabel` if it's being shown from previous errors.

After making the service call, the next task is to deal with the results from the service (see the following listing).

Listing 2.8 Result/fault handlers

```
private function
_onService_ResultHandler(event:RottenTomatoesResultEvent):void{
    var results:Array = event.result as Array;
    list.dataProvider = new ArrayList( results );
    if(results.length==0){
        errorLabel.visible = errorLabel.includeInLayout = true;
        errorLabel.text = "No Results Returned";
    }
}

private function
_onService_FaultHandler(event:RottenTomatoesFaultEvent):void{
    errorLabel.visible = errorLabel.includeInLayout = true;
    errorLabel.text = event.fault.faultDetail;
}
```

Get the results

Set the list's `dataProvider`

Check if results exist

Show "no results"

Show any errors

{chap 2 code}/src/views/SearchView.mxml

First, you need to respond to any successful results. In the result handler you need to pull the results and set the `dataProvider` with the results. You also need to deal with the possibility that there will be no results. Rather than just not showing any results, you'll alert the user that there are no results by using the `errorLabel`.

Like the fault handler in the `ListView`, the fault handler in the `SearchView` needs to display any faults that are returned by the service.

We're going to skip the list change handler method right now until after the next section when you create the `DetailsView`. Once you have created the `DetailsView`, you'll come back and add the push method into the `SearchView` list change handler. Definitely stop now and do some searching for your favorite movies.

2.3.7 Creating the details view

Whether you navigate from the `BrowseView` or the `SearchView`, you'll reach the `DetailsView` eventually. The `DetailsView` shows specifics about a single title, including the Rotten Tomatoes Scores and the movie synopsis, as shown in the following listing.

Listing 2.9 DetailsView.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
```

```
        viewActivate="_onView_ViewActivateHandler(event)">
    <fx:Script>
    <![CDATA[
    import com.rottentomatoes.vos.MovieVO;
    import spark.events.ViewNavigatorEvent;

    private function
_onView_ViewActivateHandler(event:ViewNavigatorEvent):void{
        var movie:MovieVO = data as MovieVO;
        title = movie.title;
        image.source = movie.detailedPoster;
        details.text = movie.synopsis;
        audienceScoreImage.source = movie.audienceIcon;
        audienceScoreDetails.text = movie.audienceScore+"%";
        criticsScoreImage.source = movie.criticsIcon;
        criticsScoreDetails.text = movie.criticsScore+"%";
    }

    ]]>
    </fx:Script>
    <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>
    <s:Scroller width="100%" height="100%">
        <s:VGroup width="100%" height="100%"
                              clipAndEnableScrolling="true">

        <!-- first section -->
        <s:HGroup width="100%"
            paddingBottom="5" paddingLeft="5"
            paddingRight="5" paddingTop="5">

            <!-- left, image -->
            <s:Image id="image" width="180" height="267"/>

            <!-- right, scores -->
            <s:VGroup width="100%">

                <!-- audience section -->
                <s:Label width="100%" text="Audience Score"/>
                <s:HGroup width="100%"
                     paddingBottom="5" paddingLeft="5"
                      paddingRight="5" paddingTop="5">
                    <s:Image id="audienceScoreImage"/>
                    <s:Label id="audienceScoreDetails"
                                 width="100%"/>
                </s:HGroup>
```

viewActivate handler

Set data on View

```
                <!-- critics section -->
                <s:Label width="100%" text="Critics Score"/>
                <s:HGroup width="100%"
                    paddingBottom="5" paddingLeft="5"
                     paddingRight="5" paddingTop="5">
                     <s:Image id="criticsScoreImage"/>
                     <s:Label id="criticsScoreDetails"
                                     width="100%"/>
                     </s:HGroup>
                </s:VGroup>
            </s:HGroup>

            <!-- second group -->
            <s:Label width="100%" text="Synopsis"/>
            <s:Label id="details" width="100%"/>

        </s:VGroup>
    </s:Scroller>
</s:View>
```

{chap 2 code}/src/views/DetailsView.mxml

When the view is ready, you'll set the data passed from the `data` property to the visual components in the `DetailsView`.

To complete this section, you'll go back to the `ListView.mxml` and `SearchView.mxml`. Both of these components include a method to respond to the list selection change:

```
. . .
private function _onList_ChangeHandler(event:IndexChangeEvent):void{
    navigator.pushView(DetailsView, list.selectedItem);
}
. . .
```

Push view with data

When a title is selected from the list, you push a new view onto the stack and pass the selected title to the `DetailsView`.

With the `DetailsView` complete and integrated, the basics of your application are complete. You can search and browse movie titles and view more specifics on the selected movie title.

2.4 Persisting navigator state

When you run your application, you may get annoyed when you close your application on the `DetailsView` and open the application to see the `firstView` again. You, and your users, may instead expect to see the `DetailsView` when your application starts up again. To solve this problem, the `Application`, the `ViewNavigatorApplication` and the

`TabbedViewNavigatorApplication` include a property called `persistNavigatorState`:

```
<?xml version="1.0" encoding="utf-8"?>
<s:TabbedViewNavigatorApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        persistNavigatorState="true">
    <s:ViewNavigator label="Browse" width="100%" height="100%"
        firstView="views.BrowseView"/>
    <s:ViewNavigator label="Search" width="100%" height="100%"
    firstView="views.SearchView"/>
    <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>
</s:TabbedViewNavigatorApplication>
```

The `persistNavigatorState` property tells the application to store the state of the `ViewNavigator` and any data that you've set using the `data` and `context` properties in your push method. There are three caveats to using this functionality.

The first caveat is that `persistNavigatorState` doesn't store or set the state of the view or the data pulled by the application. Don't worry about how you'll store this information; in the next chapter we'll focus on creating this solution.

The second caveat is that this functionality is turned off by default. If you intend to use this functionality, you need to remember to turn on the switch.

The third and final caveat is that the navigator's state will be forgotten if the application is uninstalled and then reinstalled or if the application's data cache is cleared. You shouldn't worry about these two scenarios, because it should be apparent to the user that if they remove the application or clear the application's memory, the navigator state will also be removed.

2.5 Summary

We've come a long way on this journey, but there are many more steps that we need to take. Hopefully you can already see what amazing applications you can put together. In the upcoming chapters you'll find smart ways to persist your data even after the application has closed, respond to varying screen sizes, and update your application to being enterprise ready with a full-strength MVC architecture.

Key takeaways:

- The `ViewNavigator` pushes and pops views to control application navigation.
- You can run an application on the desktop or on your device of choice.
- Adding new desktop configurations is easy.
- You can prevent views from destruction by the destruction policy.
- You can customize `ActionBar`s for your application or remove them completely.