Flex Mobile IN ACTION

SAMPLE CHAPTER

Jonathan Campos

MANNING

Flex Mobile in Action
By Jonathan Campos

Chapter 7

brief contents

7

Architecting multiscreen applications

This chapter covers

- Architecting multiscreen applications
- Customizing applications by platform
- Using a library

Your amazing tablet-ready Robotlegs-based application is ready for testing, additional features, and—in this chapter—separating the application for platform-specific changes, be it Android Tablet, Android Phone, iPad, iPhone, BlackBerry Tablet, Sony Tablet, or any other place for custom application types. As new platforms and screen types are created, you need an application that's easily customizable without being over architected or bloated by code or images. If you were to create just one application for all screens, this task would be almost impossible, but in this chapter you'll split the application into platform-specific application files.

If you're creating an application for just one platform, then splitting your application is a luxury rather than a necessity, though helpful. When creating multiscreen applications, splitting the application makes it easier to customize your application to the platform.

For example, if you're creating an iOS application, you'll need to include a back button to move to the previous screen. But if you include a visible back button in Android applications—ignoring the built-in back button—your users will be upset because you obviously didn't design your application for the Android platform. In this chapter you'll learn how to architect your application to be multiscreen optimized, how to use a library for maximum code reuse, and finally how to make platform-specific customizations.

7.1 Laying out a multiscreen application

The basis of making a good multiscreen application is to use a hub-and-spoke application design paradigm centralized on a master library holding as many common components as possible (see figure 7.1).

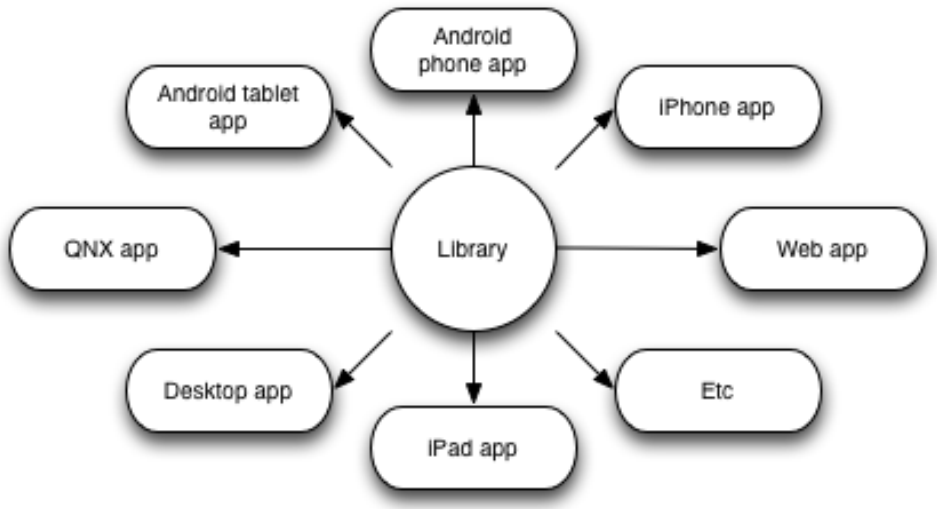


Figure 7.1 Hub and spoke application design

By using a library as the core of your application stack, you can include your models, services, commands, context, utilities, custom renderers, and any other reusable classes, leaving only the creation of custom views and mediators for the spoke projects.

Flex libraries are projects that can't run on their own but instead are a way to easily collect all of your code into one container. You've already been using the result of libraries, SWC files—the compiled collection of code. The Flex framework is distributed using SWCs along with the RottenTomatoesAS3 API that you used previously. When your library is put together, each individual device's application will be able to handle the device's unique input style while the data models stay the same across all platforms. You can share your code rather than copying and pasting between applications.

You've already created the structure using Robotlegs to make this possible without changing your application, along with using a central state model to simplify moving between views. You'll also use the `ViewNameUtil` that you created earlier to refer to specific views without calling specific view classes. In this section we'll look at what you'll be putting into the library and how you'll extend this library to create custom applications for each platform.

7.1.1 Hub library

First, you need to decide what classes you can move into your library that each of your applications will need to access to work properly. Earlier we said that the views and mediators will need to be customized to each platform, but everything else can be easily shared.

Later, as you add features, you can decide if the features will be available to all applications within the library or only to specific platforms. Depending on your decision, you'll put code into specific spoke projects or the hub library.

Within the library you'll include your data models, service layer, commands, utilities, events, and a context to tie these elements together that you can extend later (see figure 7.2).

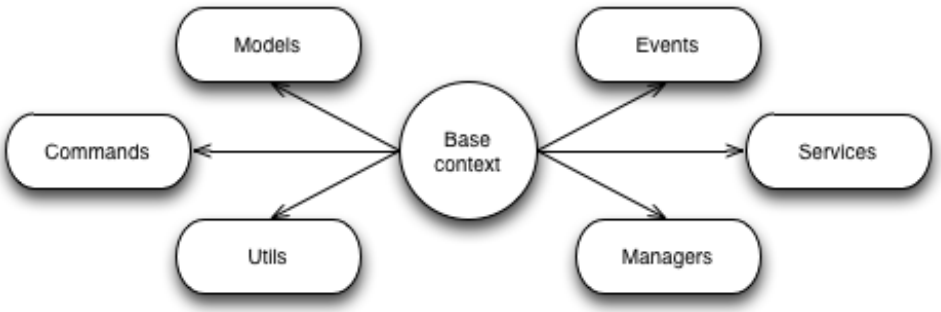


Figure 7.2 Visualized library

Next, we'll look at how you'll customize each spoke application to make it appropriate for each platform.

7.1.2 Spoke applications

As you extend the library for each application, you only need to build the views, mediators, and a subclass of your context to include all the functionality from your base library, as shown in figure 7.3.

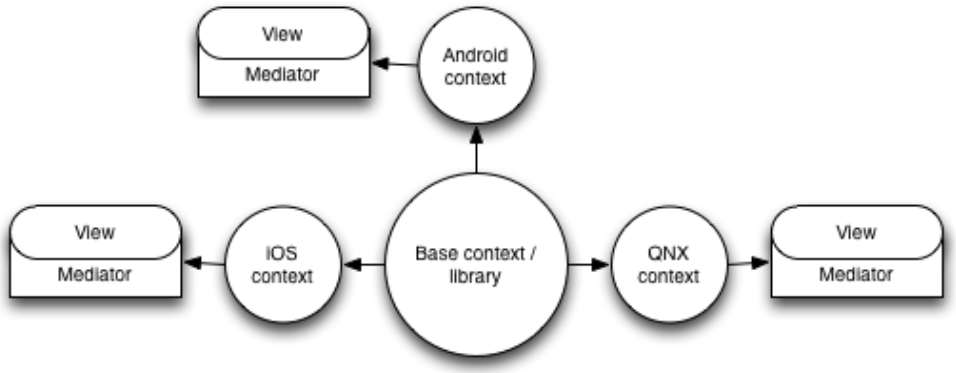


Figure 7.3 Visualized spoke applications

In the next section you'll reorganize your current application into a library so that you can create platform-specific applications later.

7.2 Libraries

Earlier we touched on what a Flex library is. Now we'll go into some more detail as to what's involved with a Flex library and how to use it, and then you'll create your own to act as the hub of your overarching application.

7.2.1 What's a Flex library?

Flex library projects are similar to Flex applications because they can encompass code, images, and other compiled libraries—also known as SWC files, CSS files, and skins created for your library.

Unlike Flex applications, there's no starting point to a Flex library project in the way that Flex applications include an application file. Therefore the compiler doesn't start at the root node, the application file, and determine what dependencies are required to run the application. Instead, the entire contents of the library are automatically compiled into the SWC file.

Finally, it's important to know that there are two ways to include the contents of a Flex library with your application. The first way, as you're currently using the RottenTomatoesAS3 library, is to use the compiled SWC that's created from the library. As you already know from using the RottenTomatoesAS3.swc file, you still get your code hinting and other helpful tips using the SWC, but if you try to jump to the code implementation of the method, you get an error (see figure 7.4).

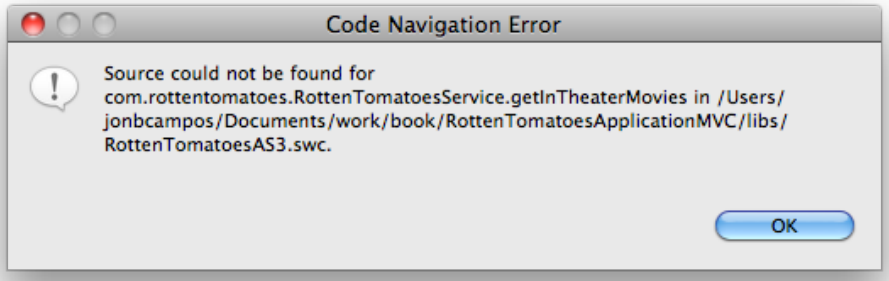


Figure 7.4 Source not found SWC error

The second method to include your Flex library is to create a connection between the application and the Flex library (see figure 7.5). When using just the SWC, if you make a change to the library, then other projects won't recompile to include the newest changes in the library. When you're making a connection between the library and other projects, any changes in the library cause a recompile on each of the connected projects.

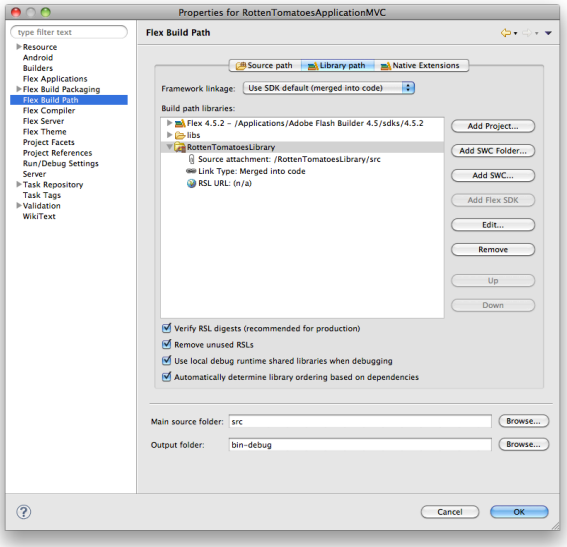


Figure 7.5 Connected library

Furthermore, if you decide to see the implementation of a method, you can skip right to the proper code.

> **Advanced library users**
>
> There are ways to connect the source code without connecting a library as well as many other advanced library features. I'd recommend you doing some additional research into Flex libraries as your requirements emerge.

You now have all the information necessary to connect the library you create with your application. Next, you'll go step by step and create your mobile application's library.

7.2.2 Setting up the library

You now create a new Flex library project by choosing New > Flex Library Project, similarly to the way you create a new Flex Mobile Project. With this option selected, you'll get the following dialog box (see figure 7.6).

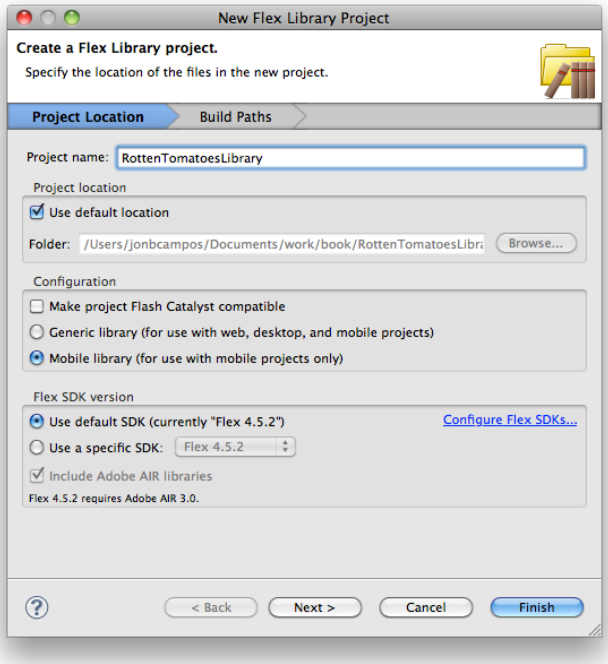


Figure 7.6 New Flex Library Project dialog box

After naming your Flex library project, you set the Configuration option to the Mobile Library setting to include mobile-specific components such as the `PersistenceManager` that you used in your models. With this one setting change, you can click Finish and continue on.

You'll then have an empty library. Continuing with your mobile library, you'll add packages to hold your classes as you move them over.

With your structure ready it's time to move over the common classes that you'll make available to all subprojects. In this case, copy all packages except for `com.unitedmindset.views` and `com.unitedmindset.mediators` and make sure to remove the appropriate mappings from your context. As you can already tell from the library's structure, many of your classes will move over directly with a quick copy-and-paste operation (see figure 7.7).

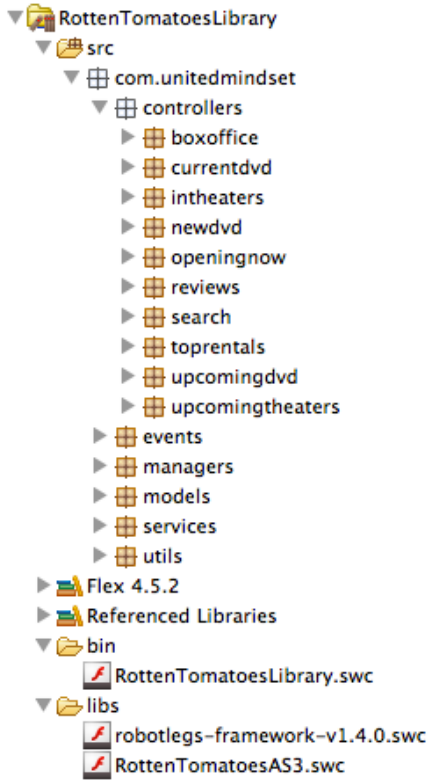


Figure 7.7 Filled-in library structure

Now that your classes are moved correctly, you'll create the base context for your sub-applications to extend. In the next section, you'll create this new class.

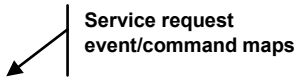
7.2.3 Creating your base context

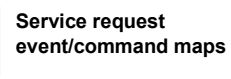
For each of your sub-applications you'll create and extend a base context that will include all of the injections and event/command mappings necessary to run your application. The view/mediator mappings will be missing from this base context because they'll be specific to the platform.

You can do some more copying and pasting here to make your life simple, creating `RottenTomatoesBaseContext.as` (see the following listing).

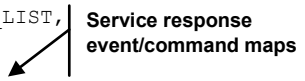
Listing 7.1 RottenTomatoesBaseContext.as

```
package com.unitedmindset{

import com.unitedmindset.controllers.boxoffice.*;
import com.unitedmindset.controllers.currentdvd.*;
import com.unitedmindset.controllers.intheaters.*;
import com.unitedmindset.controllers.newdvd.*;
import com.unitedmindset.controllers.openingnow.*;
import com.unitedmindset.controllers.reviews.*;
import com.unitedmindset.controllers.search.*;
import com.unitedmindset.controllers.toprentals.*;
import com.unitedmindset.controllers.upcomingdvd.*;
import com.unitedmindset.controllers.upcomingtheaters. *;
import com.unitedmindset.events.*;
import com.unitedmindset.models.*;
import com.unitedmindset.services.*;
import flash.display.DisplayObjectContainer;
import org.robotlegs.mvcs.Context;

public class RottenTomatoesBaseContext extends Context{
    public function RottenTomatoesBaseContext(
    contextView:DisplayObjectContainer=null, autoStartup:Boolean=true){
        super(contextView, autoStartup);
    }

    override public function startup():void{
        commandMap.mapEvent(RequestDataEvent.BOX_OFFICE_LIST,
        GetBoxOfficeCommand, RequestDataEvent);
        commandMap.mapEvent(RequestDataEvent.CURRENT_DVD_LIST,
        GetCurrentDvdCommand, RequestDataEvent);
        commandMap.mapEvent(RequestDataEvent.IN_THEATERS_LIST,
        GetInTheatersCommand, RequestDataEvent);
        commandMap.mapEvent(RequestDataEvent.NEW_DVD_LIST,
        GetNewDvdCommand, RequestDataEvent);
        commandMap.mapEvent(RequestDataEvent.OPENING_NOW_LIST,
        GetOpeningNowCommand, RequestDataEvent);
        commandMap.mapEvent(RequestDataEvent.SEARCH_LIST,
```



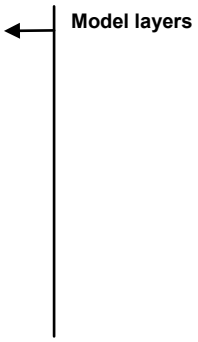



```
        GetSearchCommand, RequestDataEvent);
        commandMap.mapEvent(RequestDataEvent.TOP_RENTALS_LIST,
        GetTopRentalsCommand, RequestDataEvent);
        commandMap.mapEvent(RequestDataEvent.UPCOMING_DVD_LIST,
        GetUpcomingDvdCommand, RequestDataEvent);
        commandMap.mapEvent(RequestDataEvent.UPCOMING_THEATER_LIST,
        GetUpcomingTheatersCommand, RequestDataEvent);
        commandMap.mapEvent(RequestDataEvent.REVIEWS_LIST,
        GetReviewsCommand, RequestDataEvent);
```




```
        commandMap.mapEvent(ServiceResponseEvent.BOX_OFFICE_RESULT,
        GetBoxOfficeResultCommand, ServiceResponseEvent);
        commandMap.mapEvent(ServiceResponseEvent.CURRENT_DVD_RESULT,
        GetCurrentDvdResultCommand, ServiceResponseEvent);
        commandMap.mapEvent(ServiceResponseEvent.IN_THEATERS_RESULT,
        GetInTheaterResultCommand, ServiceResponseEvent);
        commandMap.mapEvent(ServiceResponseEvent.NEW_DVD_RESULT,
        GetNewDvdResultCommand, ServiceResponseEvent);
        commandMap.mapEvent(ServiceResponseEvent.OPENING_NOW_RESULT,
        GetOpeningNowResultCommand, ServiceResponseEvent);
        commandMap.mapEvent(ServiceResponseEvent.SEARCH_RESULT,
        GetSearchResultCommand, ServiceResponseEvent);
        commandMap.mapEvent(ServiceResponseEvent.TOP_RENTALS_RESULT,
        GetTopRentalsResultCommand, ServiceResponseEvent);
        commandMap.mapEvent(ServiceResponseEvent.UPCOMING_DVD_RESULT,
        GetUpcomingDvdResultCommand, ServiceResponseEvent);
        commandMap.mapEvent(ServiceResponseEvent.UPCOMING_THEATER_RESULT,
        GetUpcomingTheatersResultCommand, ServiceResponseEvent);
        commandMap.mapEvent(ServiceResponseEvent.REVIEWS_RESULT,
        GetReviewsResultCommand, ServiceResponseEvent);
```


```
        injector.mapSingletonOf(IRotTomService, RotTomService);
```




```
        injector.mapSingleton(BoxOfficeModel);
        injector.mapSingleton(CurrentDvdModel);
        injector.mapSingleton(InTheatersModel);
        injector.mapSingleton(NewDvdModel);
        injector.mapSingleton(OpeningNowModel);
        injector.mapSingleton(SearchModel);
        injector.mapSingleton(TopRentalsModel);
        injector.mapSingleton(UpcomingDvdModel);
        injector.mapSingleton(UpcomingTheatersModel);
        injector.mapSingleton(SelectedMovieModel);
        injector.mapSingleton(ApplicationStateModel);
    }

    override public function shutdown():void {}
```

```
        }
    }
```

{chap 7 code}/src/com/unitedmindset/com/unitedmindset/RottenTomatoesBaseContext.as

With your base context created, you can move forward and create platform-specific applications while reusing the lion's share of your code.

Reuse of view code

In the following sections I assume that you'll do some copy and pasting for your views and mediators. If you're absolutely sure that your views, and therefore the mediators, will be so similar that you can reuse their code, then by all means move some base version of your views and mediators into your library. If, however, your application's view varies based on the platform—as I assume it will—then you'll probably find that starting from scratch on your views is the easiest method to use to customize your application. For the upcoming examples I'll assume that you found it easier to start from scratch.

7.3 Android application

With your library created you can move forward and create an application specifically for the Android platform. Unique to the Android platform are the additional hardware buttons not found on other devices (see figure 7.8). Your application will respond to these input buttons, leaving the user feeling that the application is more native.

Figure 7.8 Android hardware buttons

In addition to the hardware buttons, Android applications have a consistent way of dealing with additional input options by bringing up a menu bar at the bottom of the screen (see figure 7.9).

Figure 7.9 Android menu UI

In this section you'll create and prep your Android application and then add in code to respond to the Android-specific features.

7.3.1 Setting up your Android application

Hopefully you saved a copy of your existing application, making this next step extremely easy. As before, we won't be looking at how to create each and every view because they're so similar, and instead we'll focus on `SearchView`.

First, you'll create a new Flex Mobile project and title it `RottenTomatoesAndroid`, making sure to add the `RottenTomatoesLib` in the fourth step (see figure 7.10).

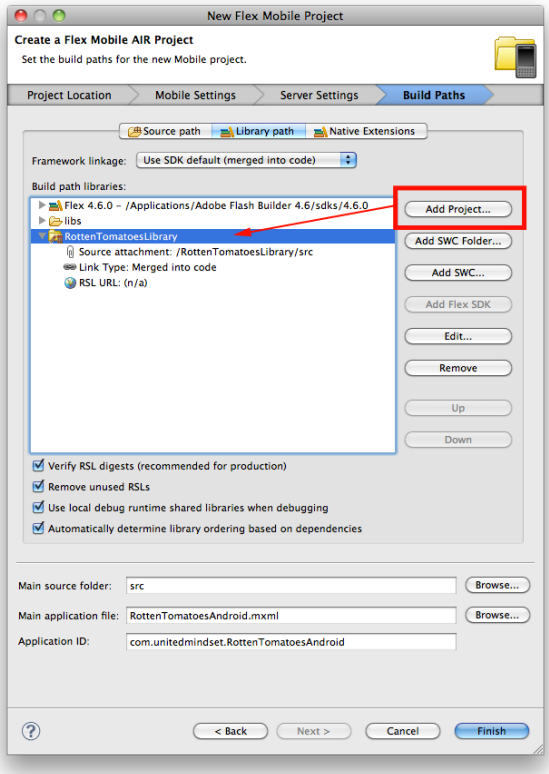


Figure 7.10 Adding the library to the Android project

With your application ready, you can copy and paste over your existing platform-agnostic views and mediators before extending the base context for your Android application (see the following listing) along with the CSS files and images.

Listing 7.2 RottenTomatoesContext.as

```
package com.unitedmindset{
    import com.unitedmindset.views.*;
    import com.unitedmindset.views.mediators.*;
    import flash.display.DisplayObjectContainer;

    public class RottenTomatoesContext extends
        RottenTomatoesBaseContext{
    public function RottenTomatoesContext(
        contextView:DisplayObjectContainer=null,
        autoStartup:Boolean=true){
```

Extend the base context

```
        super(contextView, autoStartup);
    }

    override public function startup():void{
    super.startup();

    mediatorMap.mapView(RottenTomatoesAndroid,
        ApplicationMediator);
    mediatorMap.mapView(MainMenuView, MainMenuMediator);
    mediatorMap.mapView(DetailsView, DetailsMediator);
    mediatorMap.mapView(SearchView, SearchMediator);
    mediatorMap.mapView(BoxOfficeView, BoxOfficeMediator);
    mediatorMap.mapView(CurrentDvdView, CurrentDvdMediator);
    mediatorMap.mapView(InTheatersView, InTheatersMediator);
    mediatorMap.mapView(NewDvdView, NewDvdMediator);
    mediatorMap.mapView(OpeningNowView, OpeningNowMediator);
    mediatorMap.mapView(TopRentalsView, TopRentalsMediator);
    mediatorMap.mapView(UpcomingDvdView,
        UpcomingDvdMediator);
    mediatorMap.mapView(UpcomingTheatersView, UpcomingTheatersMediator);
    }
    }
}
```

Update the `ApplicationMediator`'s view injection to reference the `RottenTomatoesAndroid` class

Include all base classes' startup code

View/mediator map

{chap 7 Android code}/src/com/unitedmindset/com/unitedmindset/RottenTomatoesContext.as

Finally, you'll set up your main application to look similar to your previous application, as shown in the following listing.

Listing 7.3 RottenTomatoesAndroid.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    preloader="spark.preloaders.SplashScreen"
    splashScreenImage="@Embed('/assets/images/splashscreen.png')"
    xmlns:unitedmindset="com.unitedmindset.*">
    <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
    <unitedmindset:RottenTomatoesContext id="context"
     contextView="{this}"/>
    </fx:Declarations>
    <s:SplitViewNavigator autoHideFirstViewNavigator="true"
        width="100%" height="100%"
        id="splitViewNavigator">
        <s:ViewNavigator width="250" height="100%" id="menuNavigator"
            firstView="com.unitedmindset.views.MainMenuView"/>
        <s:ViewNavigator width="100%" height="100%" id="navigator"/>
```

```
    </s:SplitViewNavigator>
</s:Application>
```

{chap 7 Android code}/src/com/unitedmindset/RottenTomatoesAndroid.mxml

Now with your application re-created for the Android platform—so easy—you can move into the next section and customize your search view to use the Android-like menu.

7.3.2 Customizing the view/mediators for Android capabilities

With your application re-created specifically for the Android platform, you can now add support for buttons that are only on the Android platform. To do this you head to the ListBaseView.mxml file and first add a menu to show to the user.

> **Window dressing**
>
> For this application there's a lot of "pretty-ing" that you could do both in skinning and visual effects to make your application look exactly like an Android UI. For the sake of time we'll skip these steps, because these nitpicky details don't teach any new theories of mobile application development.

For the menu you'll use an `ActionBar` colored gray like the default Android menu UI placed at the bottom of the view and with "next" and "previous" buttons, as shown here:

```
. . .
    <s:ActionBar id="bottomActionBar" width="100%"/>
</s:VGroup>

<s:ActionBar id="menuGroup"
        visible="false"
        bottom="0" width="100%"
        chromeColor="#999999">
    <s:actionContent>
        <s:Button label="Previous Page" id="prevPageButton"/>
        <s:Button label="Next Page" id="nextPageButton"/>
    </s:actionContent>
</s:ActionBar>
. . .
```

With the menu added to the `ListBaseView`, now every view that extends the `ListBaseView` also includes a menu ready for use. To utilize the menu you now add the logic within the mediator. First, you set the "previous" and "next" buttons to be enabled or disabled whenever the list value changes. This functionality is easy to add by adjusting the `_setView()` method:

```
private function _setView():void{
    . . .
    view.nextPageButton.enabled = model.isNextPage();
    view.prevPageButton.enabled = model.isPrevPage();
```

```
}
```

Next, you call for the next page or previous page when the user clicks the corresponding button:

```
private function _onNextButtonHandler(event:MouseEvent):void{
    model.getNextPage();
}

private function _onPrevButtonHandler(event:MouseEvent):void{
    model.getPrevPage();
}
```

Then you respond to the hardware buttons by listening to the key down event. In the handler method you'll determine which button was clicked and respond appropriately. For your application you'll show the menu if the menu button is clicked and go to the search page if the search button is clicked. You'll use the same coding tricks you learned in chapter 4 to respond to the hardware buttons:

```
import flash.events.KeyboardEvent;
import flash.ui.Keyboard;

private function _onKeyboardDownHandler(event:KeyboardEvent):void{
    switch(event.keyCode){
        case Keyboard.MENU:
        view.menuGroup.visible = (view.menuGroup.visible)?false:true;
            break;
        case Keyboard.SEARCH:
        stateModel.moveToView(ViewNameUtil.SEARCH_VIEW);
            break;
    }
}
```

With your handler methods created, the last task you complete is adding in the event listeners for your key down event and the previous/next button's click events in the `onRegister()` method:

```
override public function onRegister():void{
    . . .
    view.stage.addEventListener(KeyboardEvent.KEY_DOWN,
     _onKeyboardDownHandler);
    view.nextPageButton.addEventListener(MouseEvent.CLICK,
     _onNextButtonHandler);
    view.prevPageButton.addEventListener(MouseEvent.CLICK,
     _onPrevButtonHandler);
    . . .
}
```

But don't forget to remove your event listeners in the `onRemove()` method:

```
override public function onRemove():void{
    . . .
    view.stage.removeEventListener(KeyboardEvent.KEY_DOWN,
     _onKeyboardDownHandler);
    view.nextPageButton.removeEventListener(MouseEvent.CLICK,
     _onNextButtonHandler);
    view.prevPageButton.removeEventListener(MouseEvent.CLICK,
     _onPrevButtonHandler);
    . . .
}
```

With these simple changes your Android-specific functionality is complete. When the user hits the hardware menu button, a menu with the option to switch pages will appear. Then when the user changes pages, the functionality you've already coded in your model will handle the rest. With your Android application wrapped up, you can move forward with customizing your QNX/BlackBerry application.

> **Embedded images**
>
> Currently, only a few Android devices support the 320 DPI setting. If you don't intend to support these devices, this would be a good time to remove any 320 DPI code and images from your application, keeping your final release file as small as possible.

7.4 Creating your QNX-specific application

Building off your library, you'll now create an application using the same data as your previous applications, specializing your new application for the QNX platform. As with the Android platform, you'll add in the paging function off a menu customized to look appropriate for the QNX platform (see figure 7.11).

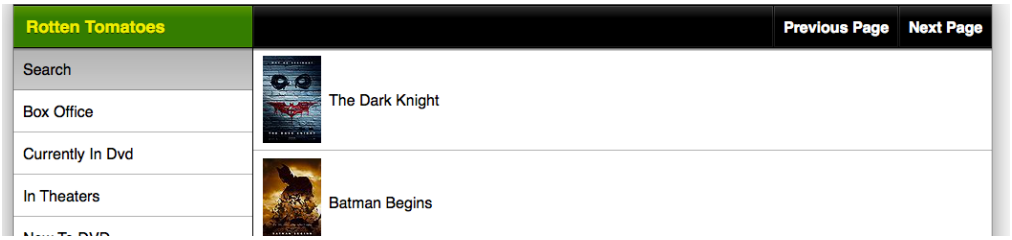


Figure 7.11 QNX menu

7.4.1 Setting up the QNX application

You'll now create a new Flex Mobile application, titled `RottenTomatoesQNX`, this one targeting the BlackBerry Tablet OS (see figure 7.12), and you'll link your newly created `RottenTomatoesLibrary` to your application.

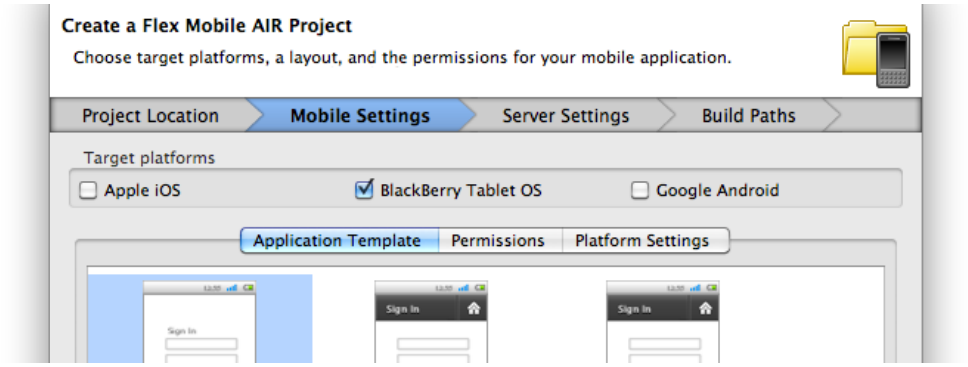


Figure 7.12 Target BlackBerry Tablet OS

With your application created and the `RottenTomatoesLibrary` included with the project—as shown earlier in figure 7.10—you need to check one more step: that the QNX libraries are included with your application. If necessary, feel free to review these steps from chapter 2 outlined in the following link: http://www.adobe.com/devnet/air/articles/packaging-air-apps-blackberry.html.

As a final check, go to the Project Properties > Flex Build Packaging > BlackBerry Tablet OS. If everything is set up properly and the required libraries are included, then your options will match those in figure 7.13.

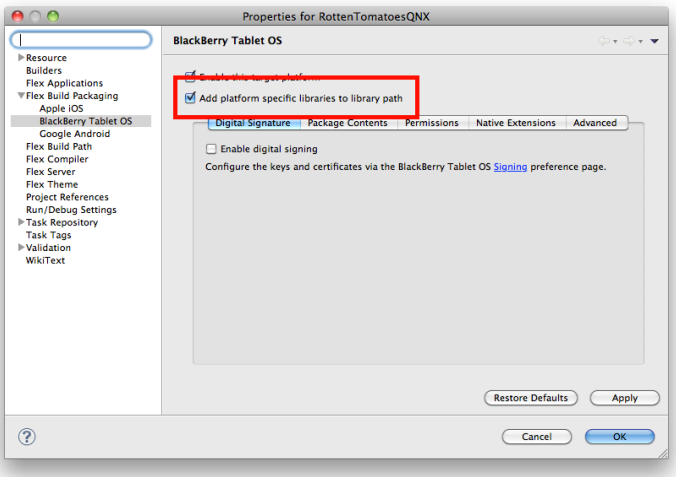


Figure 7.13 BlackBerry packaging options

You can now move forward with the creation of your application. As before with the Android application, you'll re-create your main application; copy over views, mediators, CSS files, and images; and finally create a new `RottenTomatoesContext` extending your base context.

First, you'll copy over the views and mediators that you created before in the platform-agnostic application. Then, create your `RottenTomatoesContext`, ensuring to extend the base context (see the following listing). Properly extending the base context will ensure that all of the functionality from the library is available to you.

Listing 7.4 RottenTomatoesContext.as

```
package com.unitedmindset{
    import com.unitedmindset.views.*;
    import com.unitedmindset.views.mediators.*;
    import flash.display.DisplayObjectContainer;

    public class RottenTomatoesContext extends RottenTomatoesBaseContext{
        public function RottenTomatoesContext(
        contextView:DisplayObjectContainer=null,
        autoStartup:Boolean=true){
            super(contextView, autoStartup);
        }

        override public function startup():void{
            super.startup();

    mediatorMap.mapView(RottenTomatoesQNX,
        ApplicationMediator);
    mediatorMap.mapView(BoxOfficeView, BoxOfficeMediator);
    mediatorMap.mapView(CurrentDvdView, CurrentDvdMediator);
    mediatorMap.mapView(DetailsView, DetailsMediator);
    mediatorMap.mapView(InTheatersView, InTheatersMediator);
    mediatorMap.mapView(MainMenuView, MainMenuMediator);
    mediatorMap.mapView(NewDvdView, NewDvdMediator);
    mediatorMap.mapView(OpeningNowView, OpeningNowMediator);
    mediatorMap.mapView(SearchView, SearchMediator);
    mediatorMap.mapView(TopRentalsView, TopRentalsMediator);
    mediatorMap.mapView(UpcomingTheatersView, UpcomingTheatersMediator);
    mediatorMap.mapView(UpcomingDvdView, UpcomingDvdMediator);
        }
    }
}
```

Update `ApplicationMediator`'s view injection

{chap 7 QNX code}/src/com/unitedmindset/com/unitedmindset/RottenTomatoesContext.as

With these two steps complete, you'll re-create your application before customizing your views for the QNX platform, as show in the following listing.

Listing 7.5 RottenTomatoesQNX.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    preloader="spark.preloaders.SplashScreen"
    splashScreenImage="@Embed('/assets/images/splashscreen.png')"
    xmlns:unitedmindset="com.unitedmindset.*">
    <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
    <unitedmindset:RottenTomatoesContext contextView="{this}"/>
    </fx:Declarations>
    <fx:Style source="/assets/css/main.css"/>
    <s:SplitViewNavigator autoHideFirstViewNavigator="true"
        width="100%" height="100%"
        id="splitViewNavigator">
        <s:ViewNavigator width="250" height="100%" id="menuNavigator"
            firstView="com.unitedmindset.views.MainMenuView"/>
        <s:ViewNavigator width="100%" height="100%" id="navigator"/>
    </s:SplitViewNavigator>
</s:Application>
```

{chap 7 QNX code}/src/com/unitedmindset/RottenTomatoesQNX.mxml

Your QNX application is now ready for use and currently looks and feels just like the platform-agnostic application. In the next section you'll customize the views to work specifically for the QNX platform.

7.4.2 Customizing the view/mediators for QNX capabilities

You can now customize your views to fit the QNX platform and the gesture swipes. For your application you'll give the user the ability to bring up a menu and page through the results. To do this you'll add a pop-up to the `ListBaseView`, providing a menu to each of your list views. Then you'll add the coding into your mediators to enable the menu slide-in and slide-out functionality.

EFFECTS AND UI

For a touch of class you'll have your menu slide in and bounce out using Flex Spark Effects. These effects, such as move, rotate, and fade effects, allow you to set the duration of the effect along with the starting and ending points. You'll add these effects into your ListBaseView.mxml declarations to use them on the menu pop-up in the next part:

```
<fx:Declarations>
    <s:Sine id="sine"/>
    <s:Bounce id="bounce"/>
    <s:Move id="slideIn" easer="{bounce}"/>
    <s:Move id="slideOut" easer="{sine}"/>
</fx:Declarations>
```

With your effects ready, you can add your menu into the view. You wrap the menu in a `PopupAnchor` component for two reasons: to bring up the menu over the rest of the user interface components, and to set an anchor where the component needs to pop up:

```
. . .
    <s:ActionBar id="bottomActionBar" width="100%"/>
</s:VGroup>

<s:PopUpAnchor width="100%" popUpPosition="below"
        id="menuPopup" popUpWidthMatchesAnchorWidth="true">
    <s:ActionBar chromeColor="#000000" id="menuGroup">
        <s:actionContent>
            <s:Button label="Previous Page" id="previousPage"/>
            <s:Button label="Next Page" id="nextPage"/>
        </s:actionContent>
    </s:ActionBar>
</s:PopUpAnchor>
. . .
```

You can now add the logic into your mediator to run your menu.

MENU-HANDLING LOGIC

To show and hide the menu in your Android application you just toggle the visibility of the menu. For the QNX platform, you'll pop up the menu when a user swipes down on their tablet. This process isn't difficult but does have many steps: you must listen to the swipe event, respond to the swipe event, and then toggle the menu depending on its state.

First, you'll add into the `onRegister()` and `onRemove()` methods your functionality to add and remove your events listeners:

```
override public function onRegister():void{
    . . .
    view.slideOut.addEventListener(EffectEvent.EFFECT_END,
    _onSlideOut_CompleteHandler);
    _addQnxHandlers();
    . . .
}
override public function onRemove():void{
    . . .
    view.slideOut.removeEventListener(EffectEvent.EFFECT_END,
     _onSlideOut_CompleteHandler);
    _removeQnxHandlers();
    . . .
}
```

The functionality to add the swipe is extremely simple. Implementing the `_addQnxHandlers()` and `_removeQnxHandlers()` to add and remove the required listeners is shown in the following listing.

Listing 7.6 Add and remove swipe handlers

```
private function _addQnxHandlers():void{
    try{
        var c:Class = getDefinitionByName("qnx.system.QNXApplication")
         as Class;
        var q:QNXApplication;
        c.qnxApplication.addEventListener(
        QNXApplicationEvent.SWIPE_DOWN,
        _onView_SwipeDownHandler);
    } catch(error:Error){
        //do nothing
    }
}

private function _removeQnxHandlers():void{
    try{
        var c:Class = getDefinitionByName("qnx.system.QNXApplication")
        as Class;
        var q:QNXApplication;
        c.qnxApplication.removeEventListener(
        QNXApplicationEvent.SWIPE_DOWN,
        _onView_SwipeDownHandler);
    } catch(error:Error){
        //do nothing
    }
}
```

Get access to the QNX application

Add swipe listener

Remove swipe listener

Now, when the swipe occurs, you toggle the menu's visibility:

```
private function _onView_SwipeDownHandler(event:QNXApplicationEvent):void{
    _toggleMenu();
}
```

The toggle functionality is semi-complex and is dependent on whether you're showing or hiding the menu. When you show the menu, you tell the pop-up to display, configure the slide-in effect, and set the state and listeners of the previous and next buttons. Then when you're hiding the menu, you remove the listeners from the previous and next buttons, because they'll be destroyed, and run the slide-out effect (see listing 7.7). The reason you add and remove the button's listeners in this code rather than the `onRegister()` and `onRemove()` methods is that these buttons will be created and destroyed by the `PopupAnchor` component and will not be available when the `onRegister()` method is called.

Listing 7.7 Toggle menu visibility

```
private function _toggleMenu():void{
    if(view.menuPopup.displayPopUp)
```

Toggle menu based on current visibility

```
            _hideMenu();
        else
            _showMenu();
    }

    private function _hideMenu():void{
        if(view.menuPopup.displayPopUp){
        view.nextPage.removeEventListener(MouseEvent.CLICK,
        _onNextPage);
        view.previousPage.removeEventListener(MouseEvent.CLICK,
        _onPreviousPage);

        view.slideOut.yFrom = 0;
        view.slideOut.yTo = -view.menuGroup.height;
        view.slideOut.play([view.menuGroup]);
        }
    }

    private function _showMenu():void{
        view.menuPopup.displayPopUp = true;

        view.slideIn.yFrom = -view.menuGroup.height;
        view.slideIn.yTo = 0;
        view.slideIn.play([view.menuGroup]);

        view.nextPage.addEventListener(MouseEvent.CLICK,
        _onNextPage);
        view.previousPage.addEventListener(MouseEvent.CLICK, _onPreviousPage);
        view.nextPage.enabled = model.isNextPage();
        view.previousPage.enabled = model.isPrevPage();
    }
```

Remove listeners

Configure and run effect

Display pop-up

Run slide-in effect

Add listeners and paging button's state

When the slide-out effect is complete, you set the popped-up menu's display to `false`, ensuring that the menu is removed completely:

```
    private function _onSlideOut_CompleteHandler(event:EffectEvent):void{
        view.menuPopup.displayPopUp = false;
    }
```

With this code complete, your menu shows and hides completely based on the QNX bevel swipe event. To enable the paging functionality, the last code you add into your mediator will respond to the page button click handlers. Because the functionality is already coded into your model, the code in the handler methods is extremely simple:

```
    private function _onNextPage(event:MouseEvent):void{
        model.getNextPage();
    }
```

```
private function _onPreviousPage(event:MouseEvent):void{
    model.getPrevPage();
}
```

Now that you've finished your QNX platform-specific code, you have two platform-specific applications written and ready for deployment. In the next section you'll complete the third platform in the series, making the iOS-specific customizations.

Embedded images

Currently the only QNX device, BlackBerry PlayBook, supports only the 240 DPI setting. If you want, this would be a good time to remove any 160 DPI and 320 DPI code and images from your application, keeping your final release file as small as possible.

7.5 Creating your iOS-specific application

With your library ready and the other platform-specific applications finished, the only thing left to do is create a mobile application specifically for the iOS platform. For the iOS platform you'll take the same steps that you did in previous sections, fully utilize the iOS platform's user interface with a slightly customized CSS sheet.

To get started you'll create a new Flex Mobile project targeting the iOS platform called `RottenTomatoesIOS` (see figure 7.14).

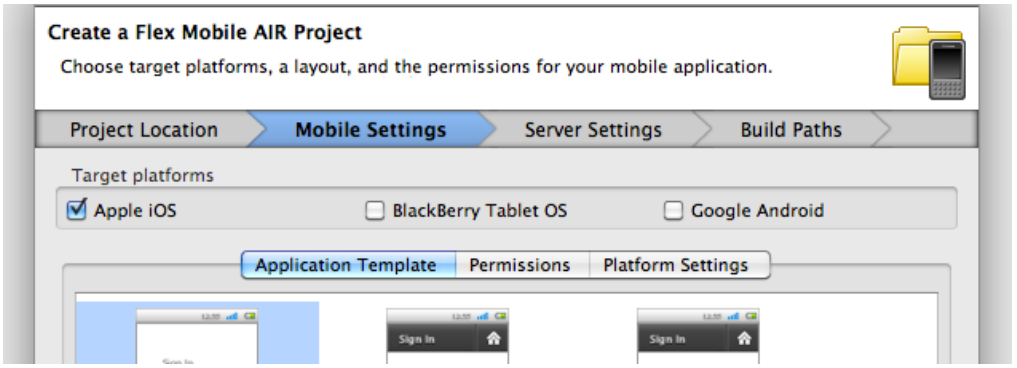


Figure 7.14 New project targeting the iOS platform

After ensuring that your application includes a reference to the `RottenTomatoesLibrary` (see figure 7.10), you can move to the next section and re-create your application prior to customizing your application.

In the next section you'll re-create your application by creating a subclass to your `RottenTomatoesBaseContext` and moving over your views from your platform-agnostic project.

7.5.1 Setting up the iOS platform application

Building out your application from a blank project into a working iOS application will go by quickly. You only need to copy over your CSS file, images, views, and mediators, then create a new context, and re-create your main application to match the platform-agnostic application used prior to your customization.

After copying and pasting over the required files, you'll create the new context to tie together all of the platform-agnostic views and mediators that you have ready in your application (see the following listing).

Listing 7.8 RottenTomatoesContext.as

```
package com.unitedmindset{
    import com.unitedmindset.views.*;
    import com.unitedmindset.views.mediators.*;
    import flash.display.DisplayObjectContainer;

    public class RottenTomatoesContext extends
        RottenTomatoesBaseContext{
        public function RottenTomatoesContext(
        contextView:DisplayObjectContainer=null,
        autoStartup:Boolean=true){
            super(contextView, autoStartup);
        }

        override public function startup():void{
            super.startup();

    mediatorMap.mapView(RottenTomatoesIOS,
        ApplicationMediator);
    mediatorMap.mapView(BoxOfficeView, BoxOfficeMediator);
    mediatorMap.mapView(CurrentDvdView, CurrentDvdMediator);
    mediatorMap.mapView(DetailsView, DetailsMediator);
    mediatorMap.mapView(InTheatersView, InTheatersMediator);
    mediatorMap.mapView(MainMenuView, MainMenuMediator);
    mediatorMap.mapView(NewDvdView, NewDvdMediator);
    mediatorMap.mapView(OpeningNowView, OpeningNowMediator);
    mediatorMap.mapView(SearchView, SearchMediator);
    mediatorMap.mapView(TopRentalsView, TopRentalsMediator);
    mediatorMap.mapView(UpcomingDvdView, UpcomingDvdMediator);
    mediatorMap.mapView(UpcomingTheatersView, UpcomingTheatersMediator);
        }
    }
}
```

Extend library's base context

Include startup configuration

Update ApplicationMediator's view injection

Map customized views to mediators

{chap 7 iOS code}/src/com/unitedmindset/com/unitedmindset/RottenTomatoesContext.as

The first two steps to re-creating your application are complete. The final step is to include the new context, kicking off your entire application, as shown in the next listing.

Listing 7.9 RottenTomatoesIOS.mxml

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    preloader="spark.preloaders.SplashScreen"
    splashScreenImage="@Embed('/assets/images/splashscreen.png')"
    xmlns:unitedmindset="com.unitedmindset.*">
    <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
    <unitedmindset:RottenTomatoesContext
        contextView="{this}"/>
    </fx:Declarations>
    <fx:Style source="/assets/css/main.css"/>
    <s:SplitViewNavigator autoHideFirstViewNavigator="true"
        width="100%" height="100%"
            id="splitViewNavigator">
        <s:ViewNavigator width="250" height="100%" id="menuNavigator"
            firstView="com.unitedmindset.views.MainMenuView"/>
        <s:ViewNavigator width="100%" height="100%" id="navigator"/>
    </s:SplitViewNavigator>
</s:Application>
```

iOS context

Main navigator

{chap 7 iOS code}/src/com/unitedmindset/RottenTomatoesIOS.as

You've fully re-created the base application that you made in the last chapter. In the next section you'll customize the list views to include new menu and paging options designed to look like other iOS user interfaces.

7.5.2 Customizing the view/mediators for iOS capabilities

With your application re-created you can now move away from the setup process and into the steps to customize the application.

First, you go to the CSS file, main.css, and make a small change. Because you'll want your buttons to look proper for the iOS platform, you'll turn on the bevel look for your `ActionBar`s:

```
s|ActionBar{
    chromeColor: #327d00;
    defaultButtonAppearance: beveled;
}
```

With this one-line addition, all of your buttons will have a beveled look, giving your application an iOS platform appearance.

Next, you go to the ListBaseView.mxml file and add your menu along with the button necessary to bring up the menu. Because the iOS hardware doesn't include a hardware menu

button in the same way as the Android platform, you need to give the application an onscreen menu button:

```
. . .
<s:navigationContent>
    <s:Button id="homeButton" label="Menu"
        includeInLayout="false" visible="false"
        includeInLayout.portrait="true" visible.portrait="true"/>
</s:navigationContent>

<s:actionContent>
    <s:Button label="Menu" id="menuButton"/>
</s:actionContent>
. . .
```

Then you add the menu user interface that you'll show when the user selects to bring up the menu:

```
. . .
    <s:ActionBar id="bottomActionBar" width="100%"/>
</s:VGroup>

<s:ActionBar id="menuGroup"
             visible="false"
             bottom="0" width="100%"
             chromeColor="#999999">
    <s:actionContent>
        <s:Button label="Previous Page" id="prevPageButton"/>
        <s:Spacer width="10"/>
        <s:Button label="Next Page" id="nextPageButton"/>
    </s:actionContent>
</s:ActionBar>
. . .
```

Window dressings

As with the other menus in the Android and QNX sections, you could have spent much more time creating a custom menu that visually fits perfectly with the platform. But these details are outside of the scope of the book and don't teach new theories.

You'll now move to the mediator to respond to the menu button's click event and display the menu.

First, you create the menu button's handler event. When a user clicks the menu button, you toggle the visibility of the menu:

```
private function _onMenuButton(event:MouseEvent):void{
    view.menuGroup.visible = (view.menuGroup.visible)?false:true;
}
```

When the user clicks the next or previous button, you use your model and call for the new page:

```
private function _onPrevButton(event:MouseEvent):void{
    model.getPrevPage();
}

private function _onNextButton(event:MouseEvent):void{
    model.getNextPage();
}
```

Because you want to ensure that the next and previous buttons are clickable only when the corresponding page exists, you return to your `_setView()` method and add in the checks for existing pages:

```
private function _setView():void{
    . . .
    view.nextPageButton.enabled = model.isNextPage();
    view.prevPageButton.enabled = model.isPrevPage();
}
```

Finally, with the methods complete, you need to return to your `onRegister()` and `onRemove()` methods, adding in and removing the event listeners, respectively:

```
override public function onRegister():void{
    . . .
    view.menuButton.addEventListener(MouseEvent.CLICK, _onMenuButton);
    view.prevPageButton.addEventListener(MouseEvent.CLICK, _onPrevButton);
    view.nextPageButton.addEventListener(MouseEvent.CLICK, _onNextButton);
    . . .
}

override public function onRemove():void{
    . . .
view.menuButton.removeEventListener(MouseEvent.CLICK, _onMenuButton);
view.prevPageButton.removeEventListener(MouseEvent.CLICK, _onPrevButton);
view.nextPageButton.removeEventListener(MouseEvent.CLICK, _onNextButton);
    . . .
}
```

Now when the user clicks the onscreen menu button, a menu will pop up at the bottom of the screen as expected, giving your user access to the list's paging features. This wraps up the current iOS customizations. You could stop now and start the deployment process with an application that feels more native than a platform agnostic application. But it's time to move on to the next chapter and add in some advanced features.

7.6 Summary

With the splitting surgery, you now have a single application building from a central library that can easily be customized to various platforms. I understand that the level of customization shown in this chapter was minimal, but hopefully you now see that by building an application in this fashion, you can easily make changes as necessary to a specific platform's user interface without hurting the customized experience of other applications.

In the next chapter we'll continue to look at advanced features that you can add to your application, including ways to access native code, track your user, and make money from advertising.

Key takeaways:

- Creating a library
- Building projects off a central library
- Customizing platform-specific projects to the target platform