

40. Implementing an Android Navigation Drawer

In this, the final of this series of chapters dedicated to the Android material design components, the topic of the navigation drawer will be covered. Comprising the `DrawerLayout`, `NavigationView` and `ActionBarDrawerToggle` classes, a navigation drawer takes the form of a panel appearing from the left-hand edge of screen when selected by the user and containing a range of options and sub-options which can be selected to perform tasks within the application.

40.1 An Overview of the Navigation Drawer

The navigation drawer is a panel that slides out from the left of the screen and contains a range of options available for selection by the user, typically intended to facilitate navigation to some other part of the application. Figure 40-1, for example, shows the navigation drawer built into the Google Play app:

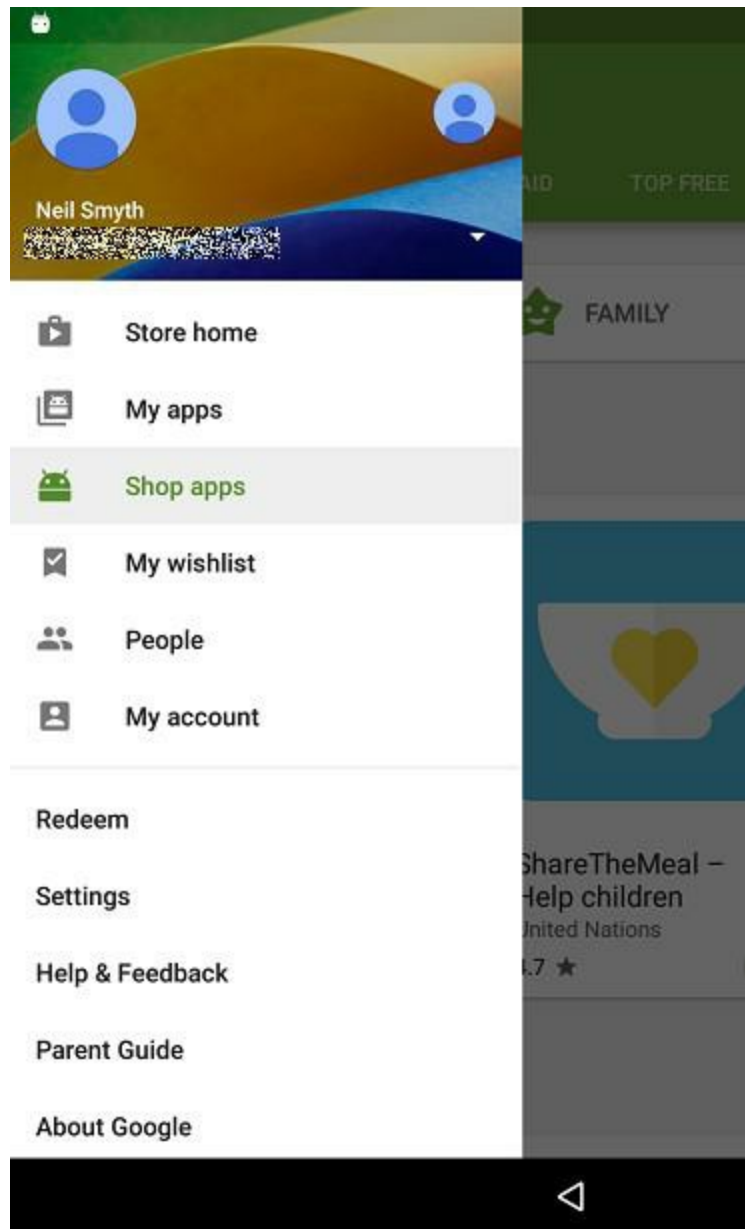


Figure 40-1

A navigation drawer is made up of the following components:

- An instance of the `DrawerLayout` component

- An instance of the `NavigationView` component embedded as a child of the `DrawerLayout`.
- A menu resource file containing the options to be displayed within the navigation drawer.
- An optional layout resource file containing the content to appear in the header section of the navigation drawer.
- A listener assigned to the `NavigationView` to detect when an item has been selected by the user.
- An `ActionBarDrawerToggle` instance to connect and synchronize the navigation drawer to the app bar. The `ActionBarDrawerToggle` also displays the drawer indicator in the app bar which presents the drawer when tapped.

The following XML listing shows an example navigation drawer implementation which also contains an include directive for a second layout file containing the standard app bar layout.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.DrawerLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:openDrawer="start">

    <include
        layout="@layout/app_bar_main"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <android.support.design.widget.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:fitsSystemWindows="true"
        app:headerLayout="@layout/nav_header_main"
        app:menu="@menu/activity_main_drawer" />

</android.support.v4.widget.DrawerLayout>
```

40.2 Opening and Closing the Drawer

When the user taps the drawer indicator in the app bar, the drawer will automatically appear. Whether the drawer is currently open may be identified via a call to the `isDrawerOpen()` method of the `DrawerLayout` object passing through a gravity setting:

```
if (drawer.isDrawerOpen(GravityCompat.START)) {
    // Drawer is open
}
```

The `GravityCompat.START` setting indicates a drawer open along the x-axis of the layout. An open drawer may be closed via a call to the `closeDrawer()` method:

```
drawer.closeDrawer(GravityCompat.START);
```

Conversely, the drawer may be opened using the `openDrawer()` method:

```
drawer.openDrawer(GravityCompat.START);
```

40.3 Responding to Drawer Item Selections

Handling selections within a navigation drawer is a two-step process. The first step is to specify an object to act as the item selection listener. This is achieved by obtaining a reference to the `NavigationView` instance in the layout and making a call to its `setNavigationItemSelectedListener()` method, passing through a reference to the object that is to act as the listener. Typically the listener will be configured to be the current activity, for example:

```
NavigationView navigationView =  
    (NavigationView) findViewById(R.id.nav_view);  
navigationView.setNavigationItemSelectedListener(this);
```

The second step is to implement the `onNavigationItemSelectedListener()` method within the designated listener. This method is called each time a selection is made within the navigation drawer and is passed a reference to the selected menu item as an argument which can then be used to extract and identify the selected item id:

```
@Override  
public boolean onNavigationItemSelectedListener(MenuItem item) {  
    // Handle navigation view item clicks here.  
    int id = item.getItemId();  
  
    if (id == R.id.nav_slideshow) {  
  
    } else if (id == R.id.nav_manage) {  
  
    } else if (id == R.id.nav_share) {  
  
    } else if (id == R.id.nav_send) {  
  
    }  
  
    DrawerLayout drawer =  
        (DrawerLayout) findViewById(R.id.drawer_layout);  
    drawer.closeDrawer(GravityCompat.START);  
    return true;  
}
```

If it is appropriate to do so, and as outlined in the above example, it is also important to close the drawer after the item has been selected.

40.4 Using the Navigation Drawer Activity Template

While it is possible to implement a navigation drawer within any activity, the easiest approach is to select the Navigation Drawer Activity template when creating a new project or adding a new activity to an existing project:

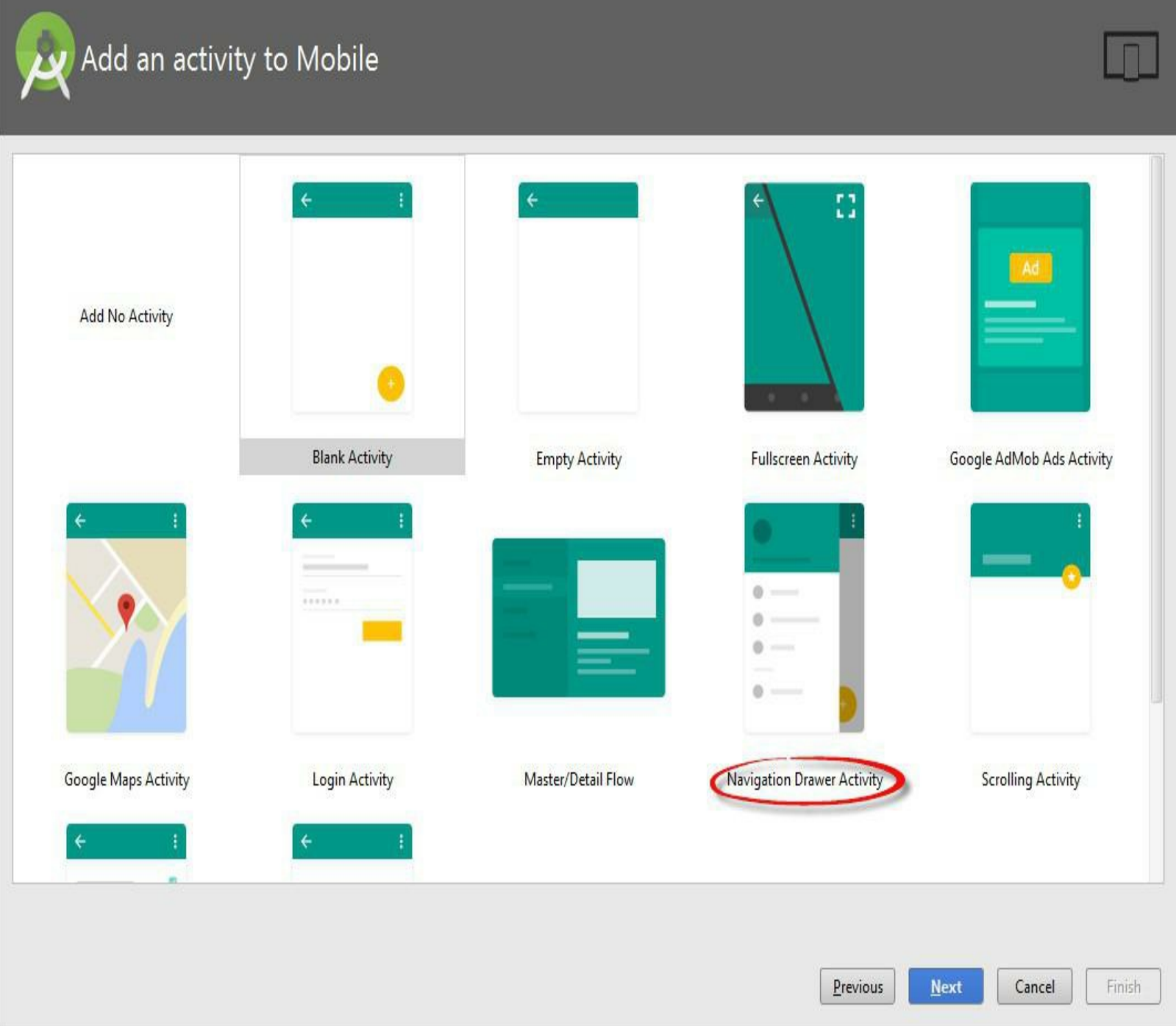


Figure 40-2

This template creates all of the components and requirements necessary to implement a navigation drawer, requiring only that the default settings be adjusted where necessary.

40.5 Creating the Navigation Drawer Template Project

Create a new project in Android Studio, entering *NavDrawerDemo* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of a Navigation Drawer Activity named *NavDrawerActivity* with a corresponding layout file named *activity_nav_drawer*. Click on the *Finish* button to initiate the project creation process.

40.6 The Template Layout Resource Files

Once the project has been created, it will contain the following XML resource files located under *app -> res -> layout* in the Project tool window:

- **activity_nav_drawer.xml** – This is the top level layout resource file. It contains the `DrawerLayout` container and the `NavigationView` child. The `NavigationView` declaration in this file indicates that the layout for the drawer header is contained within the *nav_header_nav_drawer.xml* file and that the menu options for the drawer are located in the *activity_nav_drawer_drawer.xml* file. In addition, it includes a reference to the *app_bar_nav_drawer.xml* file.
- **app_bar_nav_drawer.xml** – This layout resource file is included by the *activity_nav_drawer.xml* file and is the standard app bar layout file built within a `CoordinatorLayout` container as covered in the preceding chapters. As with previous examples this file also contains a directive to include the content file which, in this case, is named *content_nav_drawer.xml*.
- **content_nav_drawer.xml** – The standard layout for the content area of the activity layout. This layout consists of a `ConstraintLayout` container and a “Hello World!” `TextView`.
- **nav_header_nav_drawer.xml** – Referenced by the `NavigationView` element in the *activity_nav_drawer.xml* file this is a placeholder header layout for the drawer.

40.7 The Header Coloring Resource File

In addition to the layout resource files, the *side_nav_bar.xml* file located under *app -> drawable* may be modified to change the colors applied to the drawer header. By default, this file declares a rectangular color gradient transitioning horizontally from dark to light green.

40.8 The Template Menu Resource File

The menu options presented within the navigation drawer can be found in the *activity_nav_drawer_drawer.xml* file located under *app -> res -> menu* in the project tool window. By default, the menu consists of a range of text based titles with accompanying icons (the files for which are all located in the *drawable* folder). For more details on menu resource files, refer to the chapter entitled [Creating and Managing Overflow Menus on Android](#).

40.9 The Template Code

The *onCreate()* method located in the *NavDrawerActivity.java* file performs much of the initialization work required for the navigation drawer:

```
DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);

ActionBarDrawerToggle toggle = new ActionBarDrawerToggle(
    this, drawer, toolbar,
    R.string.navigation_drawer_open,
    R.string.navigation_drawer_close);

drawer.setDrawerListener(toggle);
toggle.syncState();

NavigationView navigationView = (NavigationView)
    findViewById(R.id.nav_view);
```

```
navigationView.setNavigationItemSelectedListener(this);
```

The code obtains a reference to the `DrawerLayout` object and then creates an `ActionBarDrawerToggle` object, initializing it with a reference to the current activity, the `DrawerLayout` object, the toolbar contained within the app bar and two strings describing the drawer opening and closing actions for accessibility purposes. The `ActionBarDrawerToggle` object is then assigned as the listener for the drawer and synchronized.

The code then obtains a reference to the `NavigationView` instance before declaring the current activity as the listener for any item selections made within the navigation drawer.

Since the current activity is now declared as the drawer listener, the `onNavigationItemSelectedListener()` method is also implemented in the `NavDrawerActivity.java` file. The implementation of this method in the activity matches that outlined earlier in this chapter.

Finally, an additional method named `onBackPressed()` has been added to the activity by Android Studio. This method is added to handle situations whereby the activity has a “back” button to return to a previous activity screen. The code in this method ensures that the drawer is closed before the app switches back to the previous activity screen:

```
@Override
public void onBackPressed() {
    DrawerLayout drawer =
        (DrawerLayout) findViewById(R.id.drawer_layout);
    if (drawer.isDrawerOpen(GravityCompat.START)) {
        drawer.closeDrawer(GravityCompat.START);
    } else {
        super.onBackPressed();
    }
}
```

40.10 Running the App

Compile and run the project and note the appearance of the drawer indicator as highlighted in Figure 40-3:

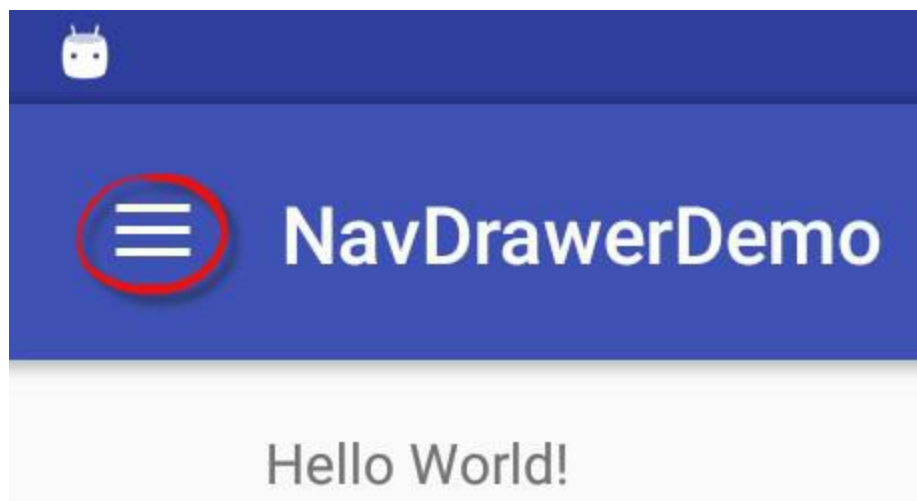


Figure 40-3

Tap the indicator and note that the icon rotates as the navigation drawer appears:

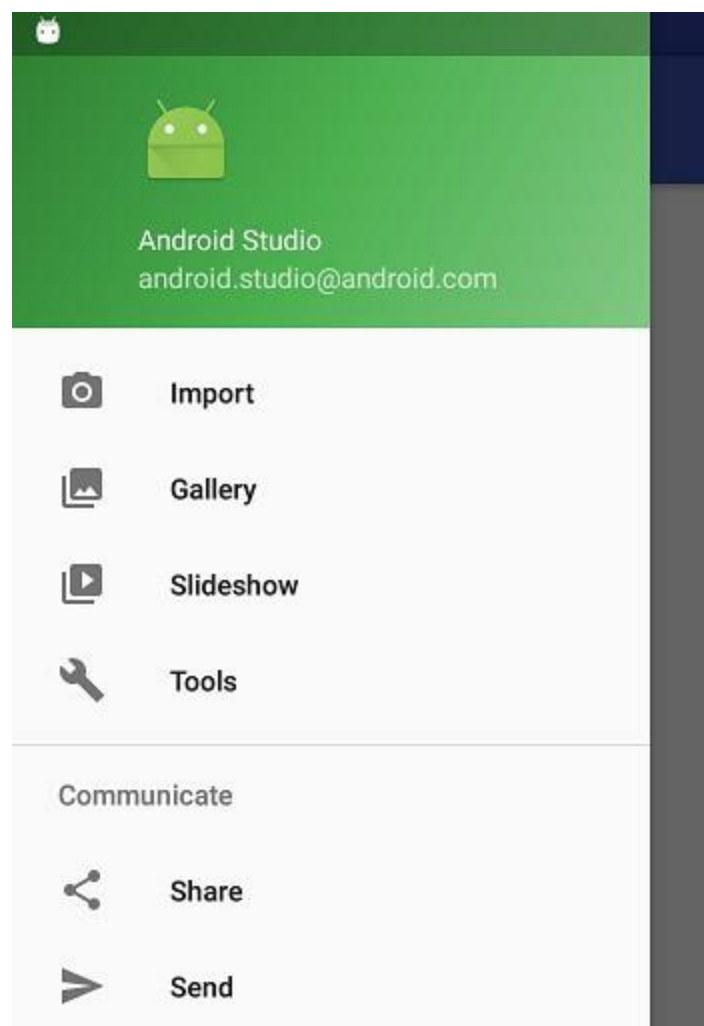


Figure 40-4

40.11 Summary

The navigation drawer is a panel that extends from the left-hand edge of an activity screen when an indicator is selected by the user. The drawer contains menu options available for selection and serves as a useful application navigation tool that conforms to the material design guidelines. Although it is possible to add a navigation drawer to any activity, the quickest technique is to use the Android Studio Navigation Drawer Activity template and then customize it for specific requirements. This chapter has outlined the components that make up a navigation drawer and highlighted how these are implemented within the template.

41. An Android Studio Master/Detail Flow Tutorial

This chapter will explain the concept of the Master/Detail user interface design before exploring, in detail, the elements that make up the Master/Detail Flow template included with Android Studio. An example application will then be created that demonstrates the steps involved in modifying the template to meet the specific needs of the application developer.

41.1 The Master/Detail Flow

A master/detail flow is an interface design concept whereby a list of items (referred to as the *master list*) is displayed to the user. On selecting an item from the list, additional information relating to that item is then presented to the user within a *detail* pane. An email application might, for example, consist of a master list of received messages consisting of the address of the sender and the subject of the message. Upon selection of a message from the master list, the body of the email message would appear within the detail pane.

On tablet sized Android device displays in landscape orientation, the master list appears in a narrow vertical panel along the left-hand edge of the screen. The remainder of the display is devoted to the detail pane in an arrangement referred to as *two-pane mode*. Figure 41-1, for example, shows the master/detail, two-pane arrangement with master items listed and the content of item one displayed in the detail pane:

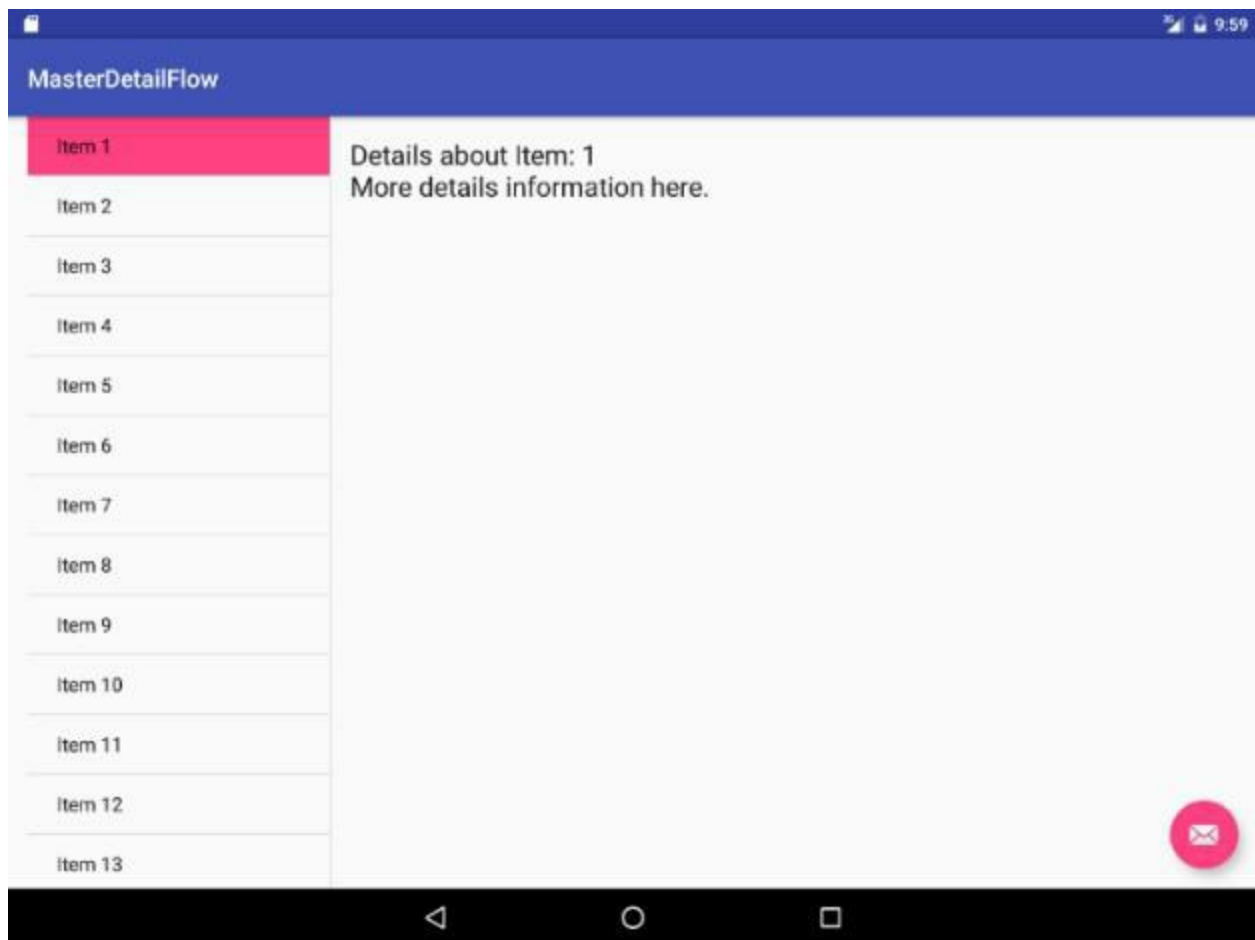


Figure 41-1

On smaller, phone sized Android devices, the master list takes up the entire screen and the detail pane

appears on a separate screen which appears when a selection is made from the master list. In this mode, the detail screen includes an action bar entry to return to the master list. Figure 41-2 for example, illustrates both the master and detail screens for the same item list on a 4" phone screen:

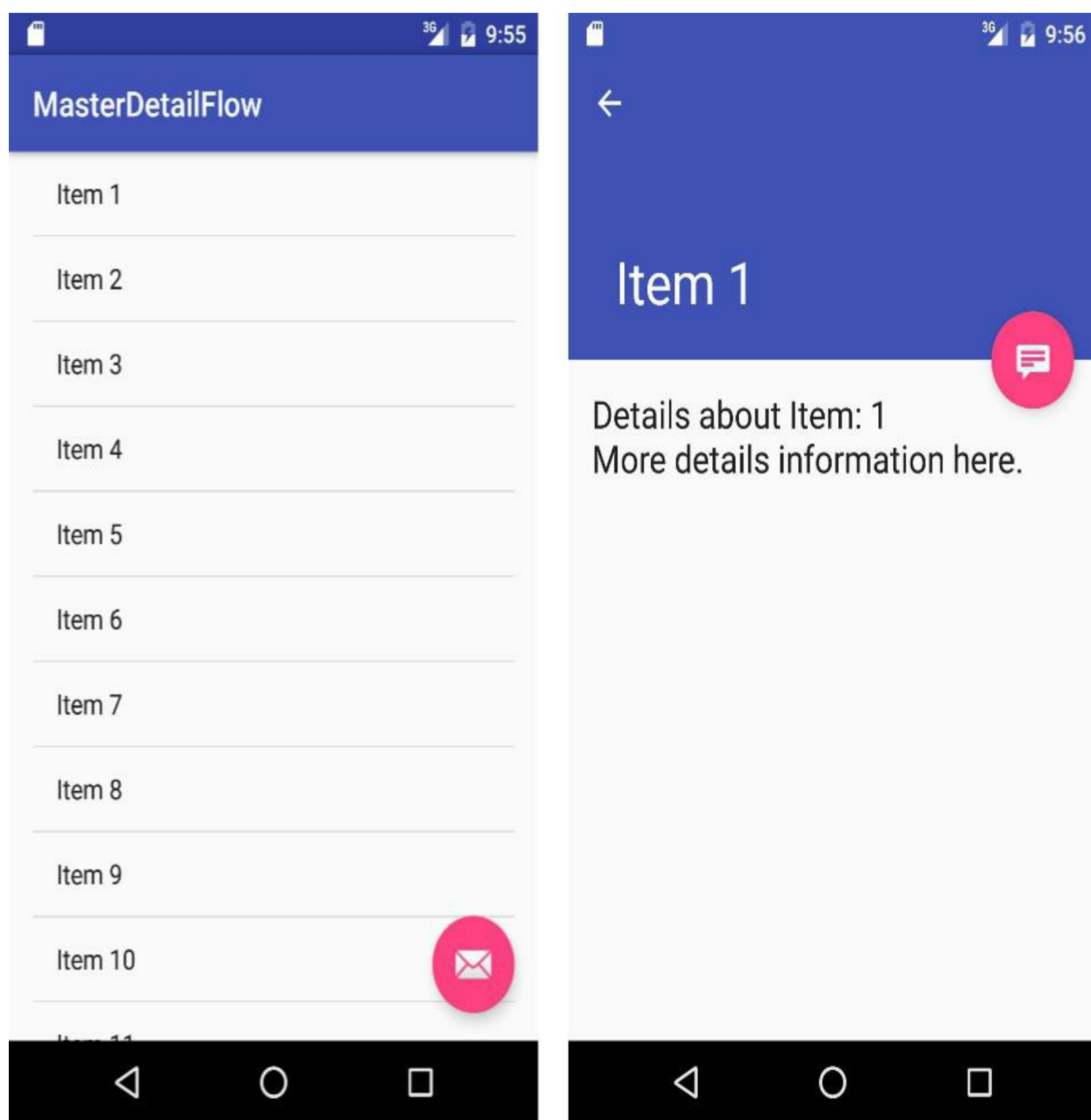


Figure 41-2

41.2 Creating a Master/Detail Flow Activity

In the next section of this chapter, the different elements that comprise the Master/Detail Flow template will be covered in some detail. This is best achieved by creating a project using the Master/Detail Flow template to use while working through the information. This project will

subsequently be used as the basis for the tutorial at the end of the chapter.

Create a new project in Android Studio, entering *MasterDetailFlow* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). When selecting a minimum SDK of less than API 14, Android Studio creates a Master/Detail Flow project template that uses an outdated and less efficient approach to handling the list of items displayed in the master panel. After the project has been created, the *minSdkVersion* setting in the *build.gradle (module: app)* file located under *Gradle Scripts* in the Project tool window may be changed to target older Android versions if required.

When the activity configuration screen of the New Project dialog appears, select the *Master/Detail Flow* option as illustrated in Figure 41-3 before clicking on *Next* once again:

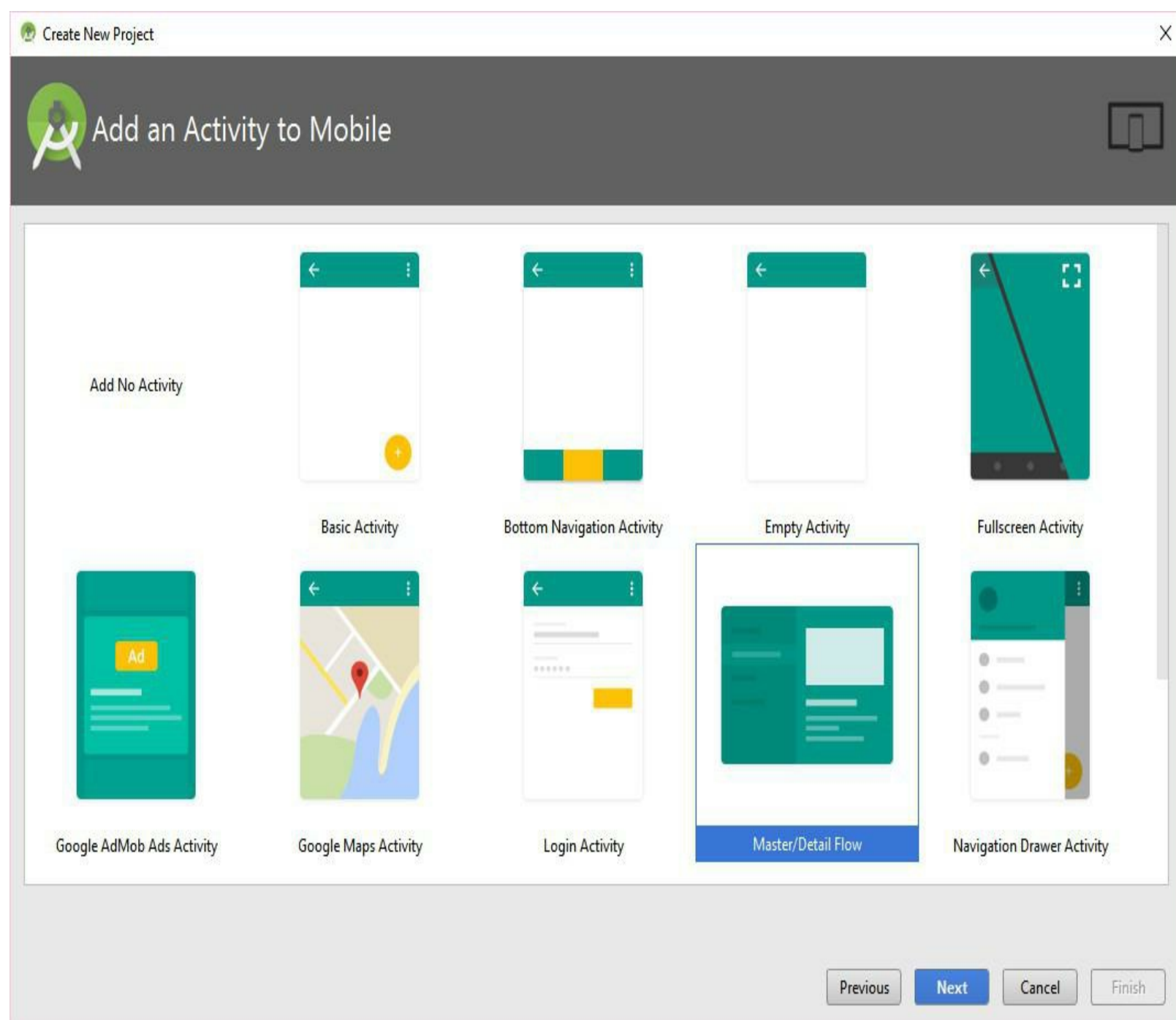
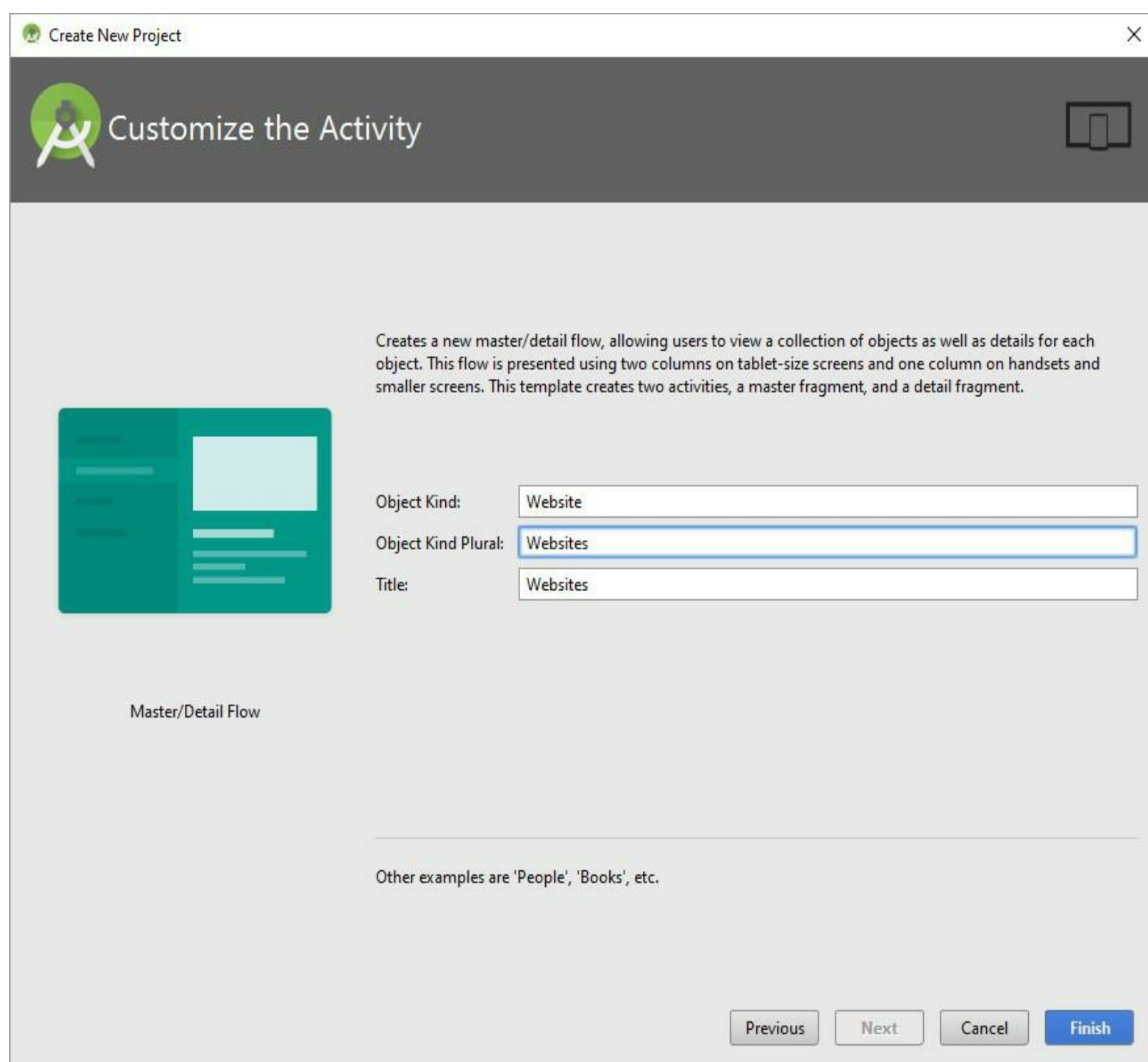


Figure 41-3

The next screen (Figure 41-4) provides the opportunity to configure the objects that will be displayed

within the master/detail activity. In the tutorial later in this chapter, the master list will contain a number of web site names which, when selected, will load the chosen web site into a web view within the detail pane. With these requirements in mind, set the *Object Kind* field to “Website”, and the *Object Kind Plural* and *Title* settings to “Websites”.



The screenshot shows the 'Create New Project' dialog box in Android Studio. The title bar says 'Create New Project' with a close button. The main header is 'Customize the Activity' with an Android logo icon. Below the header, there is a description of the Master/Detail Flow template: 'Creates a new master/detail flow, allowing users to view a collection of objects as well as details for each object. This flow is presented using two columns on tablet-size screens and one column on handsets and smaller screens. This template creates two activities, a master fragment, and a detail fragment.' To the left of this text is a teal-colored icon representing the Master/Detail Flow. Below the icon is the label 'Master/Detail Flow'. To the right of the description are three input fields: 'Object Kind:' with the value 'Website', 'Object Kind Plural:' with the value 'Websites', and 'Title:' with the value 'Websites'. Below these fields is a line of text: 'Other examples are 'People', 'Books', etc.' At the bottom right, there are four buttons: 'Previous', 'Next', 'Cancel', and 'Finish'.

Create New Project

Customize the Activity

Creates a new master/detail flow, allowing users to view a collection of objects as well as details for each object. This flow is presented using two columns on tablet-size screens and one column on handsets and smaller screens. This template creates two activities, a master fragment, and a detail fragment.

Master/Detail Flow

Object Kind: Website

Object Kind Plural: Websites

Title: Websites

Other examples are 'People', 'Books', etc.

Previous Next Cancel Finish

Figure 41-4

Finally, click Finish to create the new Master/Detail Flow based application project.

41.3 The Anatomy of the Master/Detail Flow Template

Once a new project has been created using the Master/Detail Flow template, a number of Java and XML layout resource files will have been created automatically. It is important to gain an understanding of these different files in order to be able to adapt the template to specific requirements. A review of the project within the ~~the Android~~ Android Studio Project tool window will reveal the

following files, where *<item>* is replaced by the Object Kind name that was specified when the project was created (this being “Website” in the case of the *MasterDetailFlow* example project):

- **activity_<item>_list.xml** – The top level layout file for the master list, this file is loaded by the *<item>ListActivity* class. This layout contains a toolbar, a floating action button and includes the *<item>_list.xml* file.
- **<item>ListActivity.java** – The activity class responsible for displaying and managing the master list (declared in the *activity_<item>_list.xml* file) and for both displaying and responding to the selection of items within that list.
- **<item>_list.xml**– The layout file used to display the master list of items in single-pane mode where the master list and detail pane appear on different screens. This file consists of a RecyclerView object configured to use the LinearLayoutManager. The RecyclerView element declares that each item in the master list is to be displayed using the layout declared within the *<item>_list_content.xml* file.
- **<item>_list.xml (w900dp)** – The layout file for the master list in the two-pane mode used on tablets in landscape (where the master list and detail pane appear side by side). This file contains a horizontal LinearLayout parent within which resides a RecyclerView to display the master list, and a FrameLayout to contain the content of the detail pane. As with the single-pane variant of this file, the RecyclerView element declares that each item in the list be displayed using the layout contained within the *<item>_list_content.xml* file.
- **<item>_content_list.xml** – This file contains the layout to be used for each item in the master list. By default, this consists of two TextView objects embedded in a horizontal LinearLayout but may be changed to meet specific application needs.
- **activity_<item>_detail.xml** – The top level layout file used for the detail pane when running in single-pane mode. This layout contains an app bar, collapsing toolbar, scrolling view and a floating action button. At runtime this layout file is loaded and displayed by the *<item>DetailActivity* class.
- **<item>DetailActivity.java** – This class displays the layout defined in the *activity_<item>_detail.xml* file. The class also initializes and displays the fragment containing the detail content defined in the *item_detail.xml* and *<item>DetailFragment.java* files.
- **<item>_detail.xml**– The layout file that accompanies the *<item>DetailFragment* class and contains the layout for the content area of the detail pane. By default, this contains a single TextView object, but may be changed to meet your specific application needs. In single-pane mode, this fragment is loaded into the layout defined by the *activity_<item>_detail.xml* file. In two-pane mode, this layout is loaded into the FrameLayout area of the *<item>_list.xml (w900dp)* file so that it appears adjacent to the master list.
- **<item>DetailFragment.java** – The fragment class file responsible for displaying the *<item>_detail.xml* layout and populating it with the content to be displayed in the detail pane. This fragment is initialized and displayed within the *<item>DetailActivity.java* file to provide the content displayed within the *activity_<item>_detail.xml* layout for single-pane mode and the *<item>_list.xml (w900dp)* layout for two-pane mode.

DummyContent.java – A class file intended to provide sample data for the template. This class can either be modified to meet application needs, or replaced entirely. By default, the content provided by this class simply consists of a number of string items.

41.4 Modifying the Master/Detail Flow Template

While the structure of the Master/Detail Flow template can appear confusing at first, the concepts will become clearer as the default template is modified in the remainder of this chapter. As will become evident, much of the functionality provided by the template can remain unchanged for many master/detail implementation requirements.

In the rest of this chapter, the *MasterDetailFlow* project will be modified such that the master list displays a list of web site names and the detail pane altered to contain a *WebView* object instead of the current *TextView*. When a web site is selected by the user, the corresponding web page will subsequently load and display in the detail pane.

41.5 Changing the Content Model

The content for the example as it currently stands is defined by the *DummyContent* class file. Begin, therefore, by selecting the *DummyContent.java* file (located in the Project tool window in the *app -> java -> com.ebookfrenzy.masterdetailflow -> dummy* folder) and reviewing the code. At the bottom of the file is a declaration for a class named *DummyItem* which is currently able to store two *String* objects representing a content string and an ID. The updated project, on the other hand, will need each item object to contain an ID string, a string for the web site name, and a string for the corresponding URL of the web site. To add these features, modify the *DummyItem* class so that it reads as follows:

```
public static class DummyItem {
    public String id;
    public String website_name;
    public String website_url;

    public DummyItem(String id, String website_name,
        String website_url)
    {
        this.id = id;
        this.website_name = website_name;
        this.website_url = website_url;
    }

    @Override
    public String toString() {
        return website_name;
    }
}
```

Note that the encapsulating *DummyContent* class currently contains a *for* loop that adds 25 items by making multiple calls to methods named *createDummyItem()* and *makeDetails()*. Much of this code will no longer be required and should be deleted from the class as follows:

```
public static Map<String, DummyItem> ITEM_MAP = new HashMap<String,
    DummyItem>();
```

```
private static final int COUNT = 25;
```

```

static {
    // Add some sample items.
    for (int i = 1; i <= COUNT; i++) {
        addItem(createDummyItem(i));
    }
}
-
private static void addItem(DummyItem item) {
    ITEMS.add(item);
    ITEM_MAP.put(item.id, item);
}
-
private static DummyItem createDummyItem(int position) {
    return new DummyItem(String.valueOf(position), "Item " + position,
makeDetails(position));
}
-
private static String makeDetails(int position) {
    StringBuilder builder = new StringBuilder();
    builder.append("Details about Item: ").append(position);
    for (int i = 0; i < position; i++) {
        builder.append("\nMore details information here.");
    }
    return builder.toString();
}

```

This code needs to be modified to initialize the data model with the required web site data:

```

public static final Map<String, DummyItem> ITEM_MAP =
    new HashMap<String, DummyItem>();

static {
    // Add 3 sample items.
    addItem(new DummyItem("1", "eBookFrenzy",
        "http://www.ebookfrenzy.com"));
    addItem(new DummyItem("2", "Amazon",
        "http://www.amazon.com"));
    addItem(new DummyItem("3", "New York Times",
        "http://www.nytimes.com"));
}

```

The code now takes advantage of the modified DummyItem class to store an ID, web site name and URL for each item.

41.6 Changing the Detail Pane

The detail information shown to the user when an item is selected from the master list is currently displayed via the layout contained in the *website_detail.xml* file. By default, this contains a single view in the form of a TextView. Since the TextView class is not capable of displaying a web page, this needs to be changed to a WebView object for this tutorial. To achieve this, navigate to the *app -> res -> layout -> website_detail.xml* file in the Project tool window and double-click on it to load it into the Layout Editor tool. Switch to Text mode and delete the current XML content from the file. Replace this content with the following XML:

```

<WebView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent

```



```

        android:layout_height="match_parent"
        android:id="@+id/website_detail"
        tools:context=
            "com.ebookfrenzy.masterdetailflow.WebsiteDetailFragment">
    </WebView>

```

Switch to Design mode and verify that the layout now matches that shown in Figure 41-5:



Figure 41-5

41.7 Modifying the WebsiteDetailFragment Class

At this point the user interface detail pane has been modified but the corresponding Java class is still designed for working with a `TextView` object instead of a `WebView`. Load the source code for this class by double-clicking on the *WebsiteDetailFragment.java* file in the Project tool window.

In order to load the web page URL corresponding to the currently selected item only a few lines of code need to be changed. Once this change has been made, the code should read as follows (note also the addition of the import directive for the `android.webkit.WebView` library):

```

package com.ebookfrenzy.masterdetailflow;

import android.app.Activity;
import android.support.design.widget.CollapsingToolbarLayout;
import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;
import android.webkit.WebView;

```



```

import android.webkit.WebViewClient;
import android.webkit.WebView;
import android.webkit.WebResourceRequest;

import com.ebookfrenzy.masterdetailflow.dummy.DummyContent;

public class WebSiteDetailFragment extends Fragment {
    .
    .
    .
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        if (getArguments().containsKey(ARG_ITEM_ID)) {
            // Load the dummy content specified by the fragment
            // arguments. In a real-world scenario, use a Loader
            // to load content from a content provider.
            mItem =
DummyContent.ITEM_MAP.get(getArguments().getString(ARG_ITEM_ID));

            Activity activity = this.getActivity();
            CollapsingToolbarLayout appBarLayout = (CollapsingToolbarLayout)
activity.findViewById(R.id.toolbar_layout);
            if (appBarLayout != null) {
                appBarLayout.setTitle(mItem.website_name);
            }
        }

        @Override
        public View onCreateView(LayoutInflater inflater,
            ViewGroup container, Bundle savedInstanceState) {
            View rootView = inflater.inflate(
                R.layout.fragment_website_detail, container, false);

            // Show the dummy content as text in a TextView.
            if (mItem != null) {
                ((WebView) rootView.findViewById(R.id.website_detail))
                    .loadUrl(mItem.website_url);
                WebView webView = (WebView)
                    rootView.findViewById(R.id.website_detail);
                webView.setWebViewClient(new WebViewClient() {
                    @Override
                    public boolean shouldOverrideUrlLoading(
                        WebView view, WebResourceRequest request) {
                        return super.shouldOverrideUrlLoading(
                            view, request);
                    }
                });
                webView.getSettings().setJavaScriptEnabled(true);
                webView.loadUrl(mItem.website_url);
            }

            return rootView;

```

```
}
}
```

The above changes modify the *onCreate()* method to display the web site name on the app bar:

```
appBarLayout.setTitle(mItem.website_name);
```

The *onCreateView()* method is then modified to find the view with the ID of *website_detail* (this was formally the *TextView* but is now a *WebView*) and extract the URL of the web site from the selected item. An instance of the *WebViewClient* class is created and assigned the *shouldOverrideUrlLoading()* callback method. This method is implemented so as to force the system to use the *WebView* instance to load the page instead of the Chrome browser. Finally, JavaScript support is enabled on the *webView* instance and the web page loaded.

41.8 Modifying the WebsiteListActivity Class

A minor change also needs to be made to the *WebsiteListActivity.java* file to make sure that the web site names appear in the master list. Edit this file, locate the *onBindViewHolder()* method and modify the *setText()* method call to reference the web site name as follows:

```
public void onBindViewHolder(final ViewHolder holder, int position) {
    holder.mItem = mValues.get(position);
    holder.mIdView.setText(mValues.get(position).id);
    holder.mContentView.setText(mValues.get(position).website_name);
    .
    .
    .
}
```

41.9 Adding Manifest Permissions

The final step is to add internet permission to the application via the manifest file. This will enable the *WebView* object to access the internet and download web pages. Navigate to, and load the *AndroidManifest.xml* file in the Project tool window (*app -> manifests*), and double-click on it to load it into the editor. Once loaded, add the appropriate permission line to the file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.masterdetailflow" >

    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        .
        .
        .
    </application>
</manifest>
```

41.10 Running the Application

Compile and run the application on a suitably configured emulator or an attached Android device. Depending on the size of the display, the application will appear either in small screen or two-pane mode. Regardless, the master list should appear primed with the names of the three web sites defined in the content model. Selecting an item should cause the corresponding web site to appear in the

detail pane as illustrated in two-pane mode in Figure 41-6:

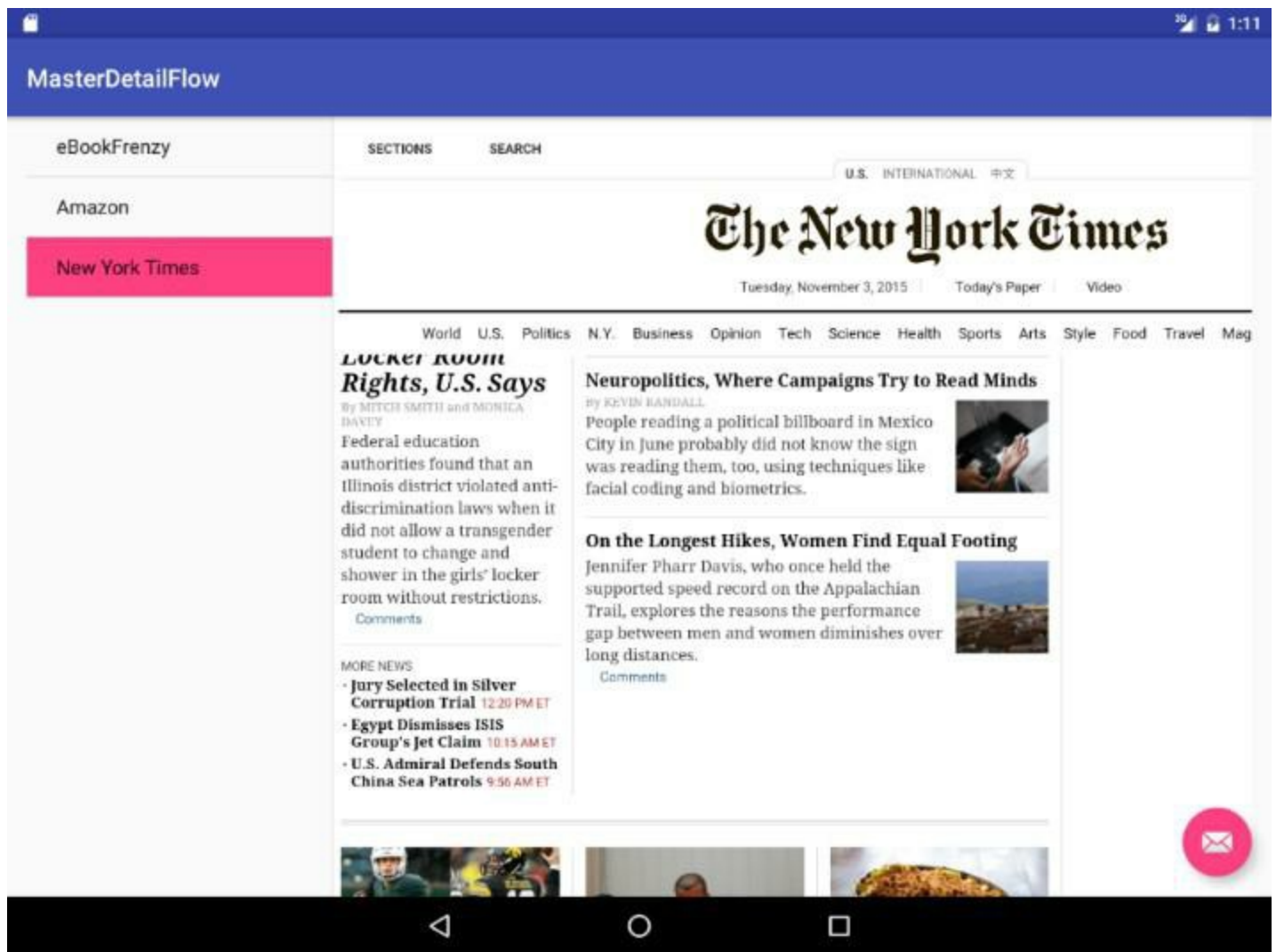


Figure 41-6

41.11 Summary

A master/detail user interface consists of a master list of items which, when selected, displays additional information about that selection within a detail pane. The Master/Detail Flow is a template provided with Android Studio that allows a master/detail arrangement to be created quickly and with relative ease. As demonstrated in this chapter, with minor modifications to the default template files, a wide range of master/detail based functionality can be implemented with minimal coding and design effort.