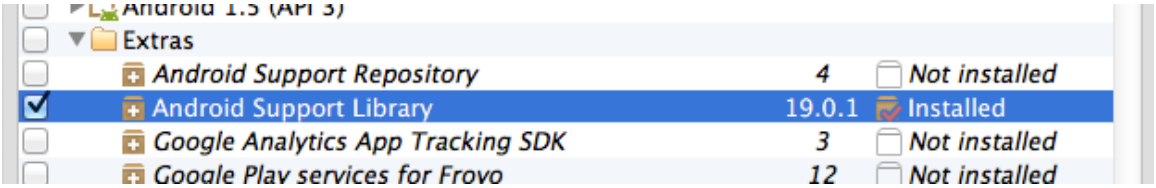
Lab 03: Flyout Navigation

Prerequisites

This lab can be done with iOS, Android, or both. For Android, you will need a development environment for Android setup; either a Mac or PC with Xamarin Studio or a Windows PC with Visual Studio and the Xamarin tools installed. For iOS, remember that if you are using Windows, you will still need an accompanying Mac on your network with XCode and the Xamarin tools to build and run the code.

Many of the Android projects require Android Support Libraries (v4 and v7 AppCompat). If you do not already have them, they can be installed from the Android SDK Manager (**Tools->Open Android SDK Manager...** from the Xamarin Studio menu). Scroll down to the **Extras** and choose to install the **Android Support Library** item.



See the **Xamarin.Android** setup documentation if you need help getting your Android environment setup:

http://docs.xamarin.com/guides/android/getting_started/installation/

See the **Xamarin.iOS** setup documentation if you need help getting your iOS environment setup:

http://docs.xamarin.com/guides/ios/getting_started/installation/

Downloads

There are several projects in a single solution that we will use to explore the various mobile navigation patterns on iOS and Android. You can download the projects from the Xamarin University website:

http://university.xamarin.com/

Lab Goals

The goal of this lab will be to get an understanding of how iOS and Android handle the tab navigation pattern. By completing this lab, you will learn about:

- The iOS and Android variants of tab navigation – how they are constructed.
- How tab selection is handled on iOS or Android (or both).

Steps

Open the Starting Solution

1. Launch Xamarin Studio and open **MobileNavigationPatterns** solution file included in your lab resources.
2. Depending on the platform you want to use, make sure either **AndroidFlyout** or **iOSFlyout** is the startup project.
3. Proceed to the next step for iOS or skip to the next section for Android.

iOS Flyout Navigation

Creating the Flyout UI

1. Within the **iOSFlyout** project, open **AppDelegate.cs**.
2. To present a flyout navigation system on iOS, you can choose a number of third-party projects. For this lab, we are using the Flyout Navigation component from the Xamarin Component Store. To make use of it, we just need to use the **FlyoutNavigationController** class. We create one of those for a class-level field in `FinishedLaunching`. and set it to be the window's `RootViewController`.

```
public static EvolveFlyoutNavigationController FlyoutNav;

public override bool FinishedLaunching (UIApplication app, NSDictionar
y options)
{
    // ...
    FlyoutNav = new EvolveFlyoutNavigationController ();
    // ...
    window.RootViewController = FlyoutNav;
}
```

3. Similar to the tab bar controllers in Lab 02, we need to tell the flyout controller about our navigation items and the view controllers to present for each one. In this case, the code to do so is isolated in a subclass of `FlyoutNavigationController`. Open **Navigation/EvolveFlyoutNavigationController.cs**.
4. Inside `ViewDidLoad`, we assign list items that will make up the navigation choices to the `NavigationRoot` property using the `RootElement` class borrowed from **MonoTouch.Dialog**, which provides a system for rapidly creating user interfaces. We customize the style of the navigation items here as well.

```
NavigationRoot = new RootElement ("Navigation") {
    new Section () {
        new StyledStringElement ("Sessions")    { BackgroundColor = UI
Color.Clear, TextColor = UIColor.LightGray },
        new StyledStringElement ("Speakers")    { BackgroundColor = UI
Color.Clear, TextColor = UIColor.LightGray },
```

```
        new StyledStringElement ("About Evolve"){ BackgroundColor = UI
Color.Clear, TextColor = UIColor.LightGray },
    }
};
```

5. Then, we create the controllers that will be used for the navigation choices and assign them to the `ViewControllers` property (order is important here).

```
var vc1 = new UINavigationController (new SessionsViewController ());
var vc2 = new UINavigationController (new SpeakersViewController ());
var vc3 = new UINavigationController (new AboutViewController ());

// Supply view controllers corresponding to menu items:
ViewControllers = new UIViewController[] {
    vc1, vc2, vc3
};
```

6. Run the application in the simulator and click the flyout button (AKA hamburger button) in the navigation bar. Switch between navigation items to see the content change.

Stack Within Separate Navigation Items

7. With the application running, click to the Sessions item in the flyout and then navigate to any session details. At this point, the flyout button is no longer available and has been replaced by a back button. Because the flyout system has no indicator except from a top-level container, stack is not maintained between navigation items.

To be able to offer maintained stack and navigation item switching from within a stack, an additional method of accessing the flyout must be provided. One way that was popular for a while was to allow the user to swipe from the left to bring in the flyout navigation view. With iOS 7 introducing the left-to-right swipe gesture to go back, many apps that once used the same gesture to trigger the flyout from any view have since changed their app's structure.

Android Flyout Navigation

Creating the Flyout UI

This project requires Android Support Libraries (v4 and v7 AppCompat). If you haven't already installed them from the SDK Manager, please see the **Prerequisites** section above.

1. To create a flyout navigation system in Android, you can use the DrawerLayout system introduced in Android v3.0 (and back-ported to earlier versions with the support libraries). To see the design of our DrawerLayout system, open **Resources/layout/Main.axml**.
2. Switch from Content to **Source** view with the tab on the bottom. Notice our `DrawerLayout` actually contains our entire UI, both the `ListView` that will contain the navigation items and also the `FrameLayout` that will contain the current content fragment.

```
<android.support.v4.widget.DrawerLayout xmlns:android="http://schemas.
android.com/apk/res/android"
    xmlns:local="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <FrameLayout
        android:id="@+id/content_frame"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
    <ListView
        android:id="@+id/flyout"
        android:divider="@android:color/transparent"
        android:dividerHeight="0dp"
        android:choiceMode="singleChoice"
        android:layout_width="240dp"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:background="#111" />
</android.support.v4.widget.DrawerLayout>
```

3. Open **MainActivity.cs**. Our activity must again inherit from the support `ActionBarActivity`, as it did in Lab 02 for tabs. Unlike in tabs, however, we do not need to implement an interface to receive notification when a flyout navigation item is selected.

```
public class MainActivity : ActionBarActivity
```

4. Next, in `OnCreate`, we grab references to the layout items we need from the view. For our ListView, we give its Adapter a simple ArrayAdapter with our hard-coded menu strings.

```
drawerLayout = FindViewById<DrawerLayout> (Resource.Id.drawer_layout);
//...
drawerList = FindViewById<ListView> (Resource.Id.flyout);
drawerList.Adapter = new ArrayAdapter<string> (this, Resource.Layout.d
rawer_list_item, sections);
```

5. To get the hamburger icon with a **DrawerLayout**, we create an `ActionBarDrawerToggle`, passing it a icon drawable to use and some strings that are used for accessibility messaging. Then, we tell our DrawerLayout that our toggle will need to know about flyout state changes so it can update.

```
drawerToggle = new ActionBarDrawerToggle (this, drawerLayout, Resource
.Drawable.ic_drawer, Resource.String.drawer_open, Resource.String.draw
er_close);
drawerLayout.SetDrawerListener (drawerToggle);
```

6. To handle navigation item clicks, we simply tie up a click handler to the list used in the `DrawerLayout`. We also call that handler explicitly to pre-select the first item.

```
drawerList.ItemClick += (object sender, AdapterView.ItemClickEventArgs
 e) => ListItemClicked (e.Position);
ListItemClicked (0);
```

7. Inside our ListItemClicked handler method, we do the usual fragment replacement, but we also check the current item, update the title, and close the drawer. (The remaining overridden methods are there to keep the drawer's state in sync.)

```
void ListItemClicked (int position)
{
    SupportFragmentManager.PopBackStack (null, (int)PopBackStackFlags.
Inclusive);

    Fragment fragment = null;
    switch (position) {
    case 0:
        fragment = new SessionListFragment ();
        break;
    case 1:
        fragment = new SpeakerListFragment ();
        break;
    case 2:
        fragment = new AboutFragment ();
        break;
    }

    // Insert the fragment by replacing any existing fragment
    SupportFragmentManager.BeginTransaction ()
        .Replace (Resource.Id.content_frame, fragment)
        .Commit ();

    // Highlight the selected item, update the title, and close the dr
awer
    drawerList.SetItemChecked (position, true);
    SupportActionBar.Title = sections [position];
    drawerLayout.CloseDrawer (drawerList);
}
```

8. Run the application in the emulator and click around the navigation items. The fragment swap is changing out which content is shown without adding anything to the back stack.

Stack Within Separate Navigation Items

9. While still running the app, click through to a details fragment from the **Speakers** navigation item.
10. Now, press the back button once. As with Lab 02, notice that going to the detail fragment was part of the back stack.
11. Open **SpeakerListFragment.cs** and find the `ShowDetails` method. To add the fragment replacement to the back stack, we simply added a call to `AddToBackStack` on the `FragmentTransaction`.

```
FragmentManager.BeginTransaction ()
    .Replace (Resource.Id.content_frame, details)
    .AddToBackStack (null)
    .Commit ();
```

12. Return to **MainActivity.cs**. Since we are in control of what happens when a navigation item is selected, just as we were with tabs in lab 02, we have a couple choices about handling this back-stack navigation within a navigation item. We could maintain the stack manually and recreate it when a tab is selected after leaving. Alternatively, as we do in this lab, we simply clear out the back stack so that it doesn't create any issues when using the back button after switching tabs.

```
SupportFragmentManager.PopBackStack (null, FragmentManager.PopBackStac
kInclusive);
```

Summary

In this lab, we saw how we can use the flyout navigation pattern in iOS and Android for our applications. We reviewed how to create flyout systems using a component on iOS and the ActionBar DrawerLayout on Android and saw how to handle navigation item switches by the user. As well, we saw how back stack is maintained (or not) within flyout navigation changes.