Lab 1: Getting Started

In this lab, you'll learn the basics of Navigation through the process of migrating an existing app. You'll learn how to create a graph describing the available screens in your app, how to create routes to get from one screen to another, and how to safely ensure data is being passed from one screen to the next. You'll also learn how you can tie options menu items to app Navigation with less boilerplate code. Let's get started!

A starter project with basic functionality has been provided so that you can focus on the navigation aspects of the app.

Lab Setup:

Clone the repository

```
git clone https://github.com/ericmaxwell2003/contacts.git
```

When you clone the repository, you should be on the default branch, ***solution/lab_1***. Double check this using the *git branch* command.

```
git branch
```

You should see something like the sample shown below, where the ***** marks the branch you are on:

```
* starter/lab_1
Erics-MBP-2:app emaxwell$
```

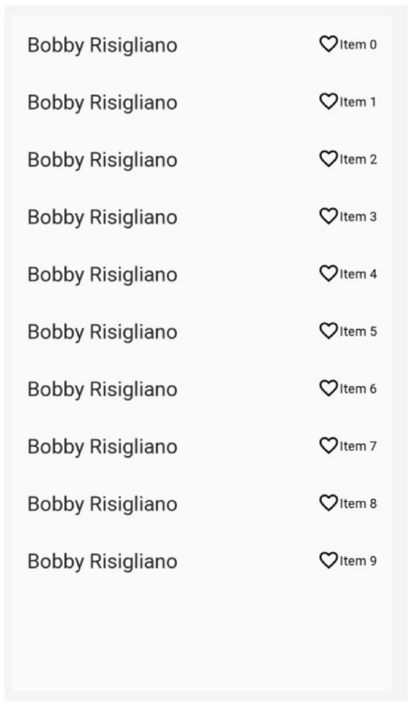
If you find that you are on a different branch, change to the **starter/lab_1** branch via the *git checkout* command.

```
git checkout starter/lab_1
```

Starter Code

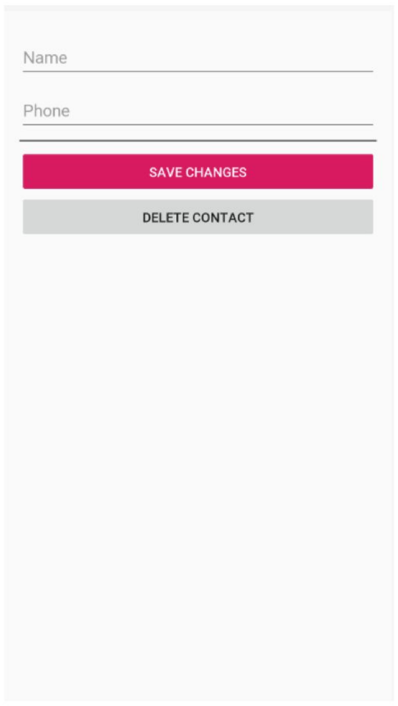
Before continuing, take a moment to explore the starter code. The starter app for this workshop consists of 2 screens, represented by 2 fragments. It uses a single activity to host them.

The screen details are defined in 3 layout files and look like what you see below.



fragment_contacts_list.xml

contact_item.xml



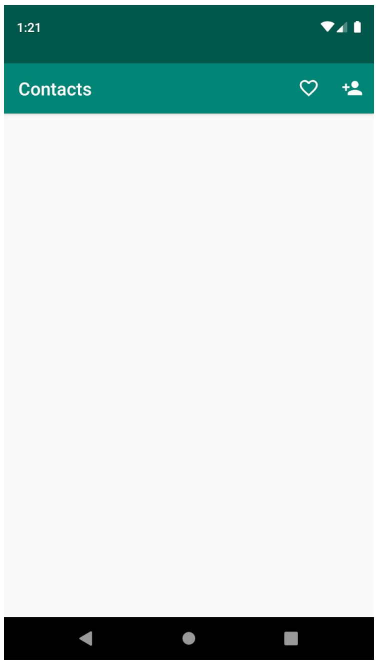
fragment_contact_detail.xml

So that you can focus on learning the Navigation framework, this application has already been set up with a database and a few supporting classes. The file structure inside the app module is described below:

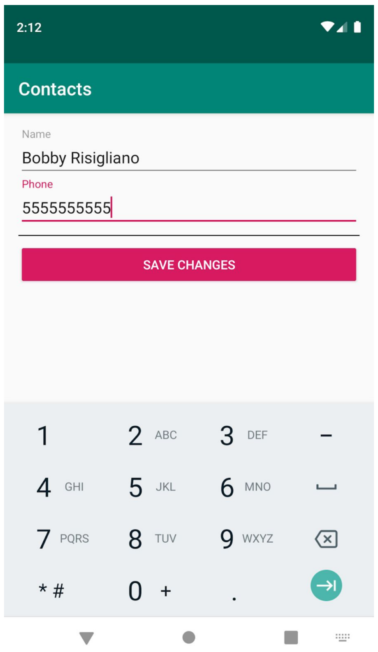
Package/Folder	Desc
com.acme.contacts	Root Package **<u>Contains:</u>** ***MainActivity*** - Single Activity for the app ***ContactsApplication*** - Custom Application class that performs some database initialization when the app starts. ***Contact*** - Model class that represents a contact in the app. ***BindingAdapters.kt*** - Custom data binding adapters for this app.
database	Room database setup. You won't need to explore this in this workshop, but it's there if you are curious.
detail	ContactDetailFragment and supporting classes for the details screen.
list	ContactsListFragment and supporting classes for the list screen.

Run the Starter App

Run the app at this point and you should see a screen that looks similar to the following.



Clicking on the add button in the upper left hand side of the screen takes you to the contact details screen where you can enter a name and phone number for the contact.



You can then save and go back, or choose to go back without saving.

Current State

This app currently hosts the contact lists and details fragments in the main activity, inside of a FrameLayout as shown below

File (activity_main.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
       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"
       tools:context=".MainActivity">

   <FrameLayout
           android:id="@+id/nav_host"
           android:layout_width="match_parent"
           android:layout_height="match_parent"
           tools:layout="@layout/fragment_contacts_list" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

The *MainActivity* class adds an instance of *ContactsListFragment* within this container when the app is first launched and provides a function for replacing it later with the *ContactDetailFragment*.

File (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {

   override fun onCreate(savedInstanceState: Bundle?) {
       ...

       if(savedInstanceState == null) {
           supportFragmentManager.beginTransaction()
               .add(R.id.nav_host, ContactsListFragment())
               .commit()
       }
   }

   fun navToContactDetails(contactId: String? = null) {

       val contactDetailFragment = ContactDetailFragment().apply {
           arguments = Bundle().apply {
               putString("contactId", contactId)
           }
       }
```

```
        supportFragmentManager
            .beginTransaction()
            .replace(R.id.nav_host, contactDetailFragment)
            .addToBackStack(null)
            .commit()
    }
}
```

Forward Navigation

The *navToContactDetails* function is called from the *ContactsListFragment* in order to navigate to the contact details screen anytime the *Add Contact* menu option is clicked, and also when an existing contact in the list is clicked.

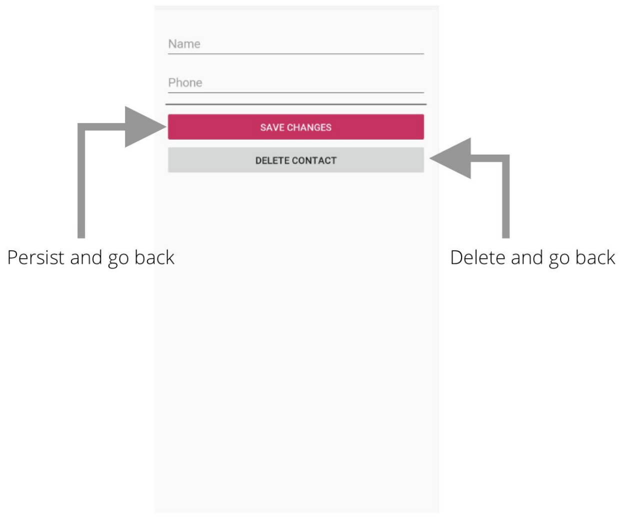
File (ContactsListFragment.kt)

```
class ContactsListFragment : Fragment() {
   ...
   override fun onOptionsItemSelected(item: MenuItem): Boolean {
       return when(item.itemId) {
              R.id.toggle_favorites_only -> {
                  contactsListVm.toggleShowFavoritesOnly()
                  requireActivity().invalidateOptionsMenu()
                  true
              }
              R.id.to_contact_detail -> {
                  (activity as MainActivity).navToContactDetails()
                  true
              }
              else ->  super.onOptionsItemSelected(item)
          }
   }

   private fun onContactItemClick(contact: Contact) {
       (activity as MainActivity).navToContactDetails(contact.id)
   }
  ...
}
```

Backward Navigation

In the contact details screen, back button functionality is handled automatically because the list fragment is added to the back stack before replacing it with the ContactDetailFragment.

However, in the case of clicking Save or Delete, the app programmatically backs out of the details screen after instructing the viewModel to save or delete the contact being edited.



In order to go back, the ContactDetailFragment accesses the activity's support fragment manager, as shown in the following snippet from *ContactDetailFragment*.

File (ContactDetailFragment.kt)

```
class ContactDetailFragment : Fragment() {
  ...
  fun onSaveContact() {
     contactDetailVm.saveContact()
     requireActivity().supportFragmentManager.popBackStack()
  }

  fun onDeleteContact() {
     contactDetailVm.deleteContact()
     requireActivity().supportFragmentManager.popBackStack()
  }
  ...
}
```

Take a moment to explore the code to get your bearings and move on when you're ready.

Your first task will be to introduce the Navigation Architecture Component library and use it to perform this basic navigation, instead of creating FragmentTransactions.

Introducing Navigation

To get started with Navigation, the first thing you need to do is to add a dependency on the framework. The dependencies have been added to this starter project already so that you don't have to try to download them during the lab.

Add Navigation Dependencies

Open the **app/build.gradle** file and add notice the two dependencies highlighted below

```
dependencies {
  ...
  implementation "androidx.navigation:navigation-fragment-ktx:2.1.0-beta02"
  implementation "androidx.navigation:navigation-ui-ktx:2.1.0-beta02"
  ...
}
```

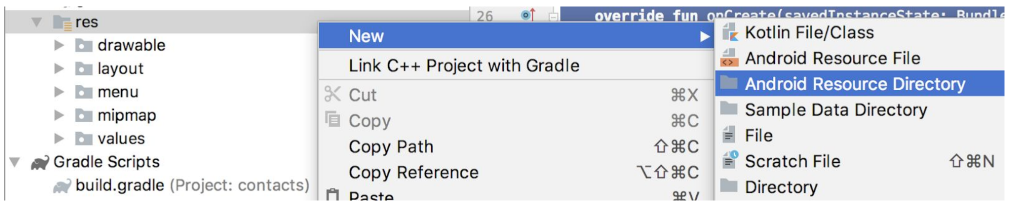
The first dependency, ***navigation-fragment-ktx***, contains all the basic APIs and library components needed to use Navigation within your app. It has a transitive dependency on the navigation runtime (***navigation-runtime-ktx***).

The second dependency, ***navigation-ui-ktx***, is optional but helpful for using Navigation with menu based or top level navigation based UIs. You be taking advantage of this optional dependency in this first lab.

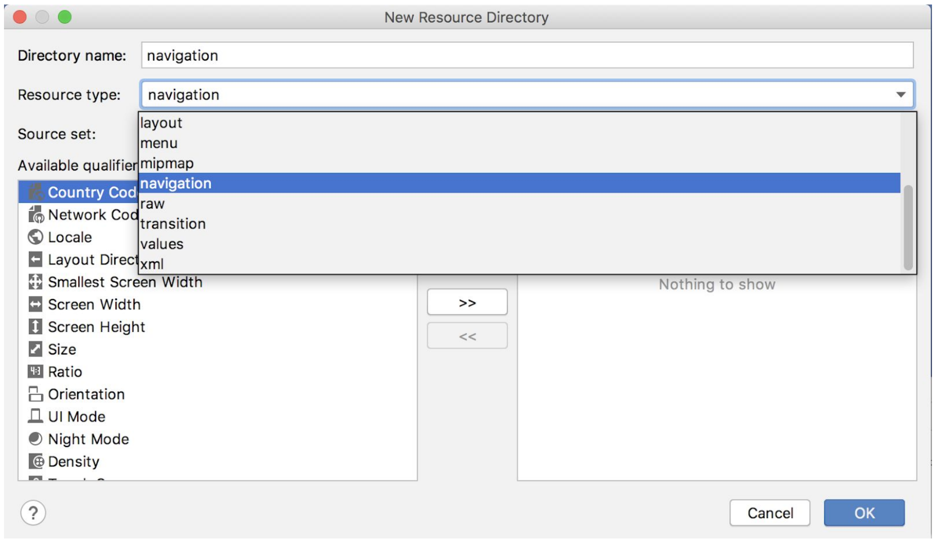
Adding a Navigation Graph

Next you'll need to add a navigation graph. The navigation graph is an Android XML resource file that describes screens in your app (*destinations*) and named routes to get from one screen to another (*actions*). Navigation resource files live in a resources folder called, *navigation*.

Create a navigation resource folder by **right clicking** on the **res** directory in the project explorer window and select **New -> Android Resource Directory.**



Select Navigation as the resource type. This will automatically set the directory name as *navigation*, which is fine. Accept all defaults and click okay.



New Resource Directory

Directory name: navigation

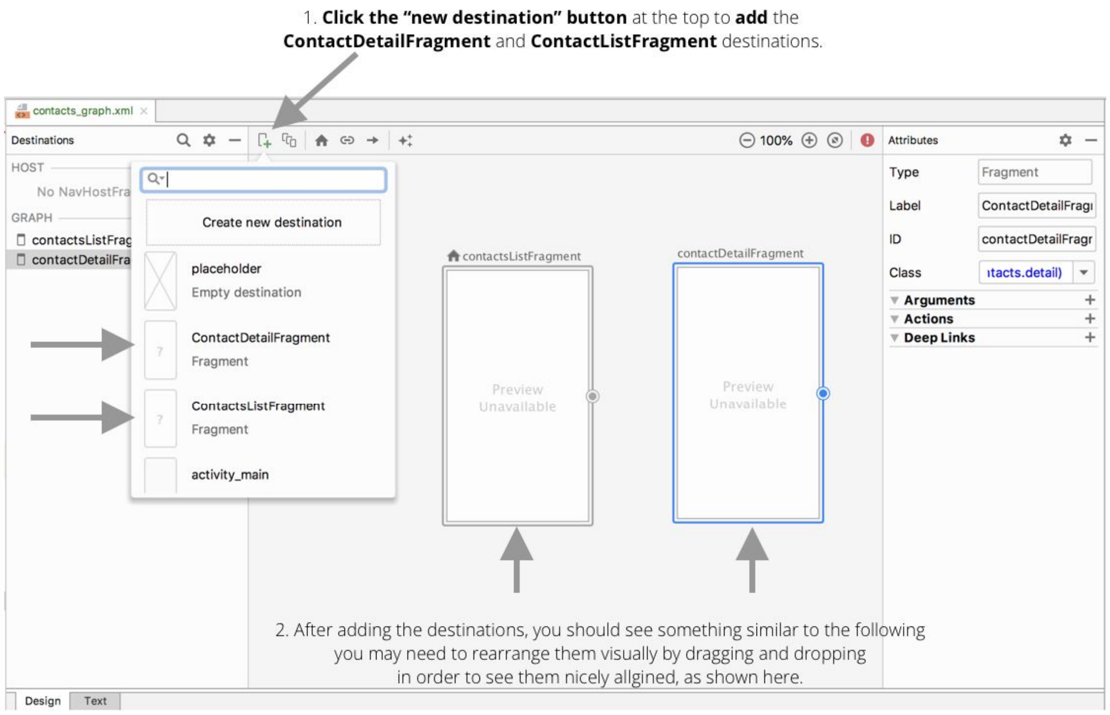
Resource type: navigation

Source set:

Available qualifier

Once that is created, create a navigation graph called ***contacts_graph.xml*** by **right clicking the navigation folder** and select **New -> Navigation Resource File**. Then, enter the name *contacts_graph* and click okay.

This will create an empty graph, to which you can add the list and details screens as destinations. To do that, you'll use the graphical editor and add them via the ***New Destination*** button. Follow the steps shown in the graphic below to do that.



1. **Click the "new destination" button** at the top to **add** the **ContactDetailFragment** and **ContactListFragment** destinations.

2. After adding the destinations, you should see something similar to the following you may need to rearrange them visually by dragging and dropping in order to see them nicely allgined, as shown here.

In the bottom left hand corner, you'll see two tabs, *Design* and *Text*. Click on the text tab and you should see that XML looks similar to the following.

File (contact_graph.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
           xmlns:app="http://schemas.android.com/apk/res-auto"
           android:id="@+id/contacts_graph"
           app:startDestination="@id/contactsListFragment">

   <fragment android:id="@+id/contactsListFragment"
             android:name="com.acme.contacts.list.ContactsListFragment"
             android:label="ContactsListFragment"/>

   <fragment android:id="@+id/contactDetailFragment"
             android:name="com.acme.contacts.detail.ContactDetailFragment"
             android:label="ContactDetailFragment"/>

</navigation>
```

For the purpose of this lab, you'll be working within XML solely because it's easier to communicate the steps via XML changes. In practice, you'll frequently find yourself bouncing between the *Text* and *Design* tabs to make changes to your resource graph.

What you see in the XML is a Nav Graph root element, ***navigation,*** and 2 fragment based destinations enclosed therein. Each destination has an id (***android:id***). The ***navigation*** element, represents a special type of destination called a *nav graph*.

The nav graph references a starting destination (***app:startDestination***). The startDestination indicates the first destination that Navigation should send the user to when the app starts up. The id value for the starting destination must match the id of one of the destinations contained within the navigation element. It is required that the navigation element has a startDestination attribute defined. You will get a build time error if it is not present.

You want to send the user to the *contactListFragment* destination when the app first starts. Depending on which screen you added first to your graph, Android Studio may have selected the *contactDetailFragment* destination instead. If this is the case, update ***app:startDestination*** value to point to the *contactListFragment* destination.

Destination Ids and Labels

It's a good idea to put careful thought into the id's and labels of your destinations. There isn't a specific standard for naming convention for these, however I have a few personal opinions/suggestions:

- Prefer snake case to match the idioms of XML when possible.
- Drop the *Fragment* suffix from the id's.

The Navigation framework will use the labels you provide to set the title in the support action bar automatically for you, so you should set those to meaningful values that you'd like to use as the default when the user navigates to that destination (screen).

With this in mind, update the existing XML to look like this:

File (contact_graph.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
           xmlns:app="http://schemas.android.com/apk/res-auto"
           android:id="@+id/contacts_graph"
           ~~app:startDestination="@id/contactsListFragment">~~
           app:startDestination="@id/contacts_list">

   ~~<fragment android:id="@+id/contactsListFragment"~~
   <fragment android:id="@+id/contacts_list"
             android:name="com.acme.contacts.list.ContactsListFragment"
             ~~android:label="ContactsListFragment"/>~~
             android:label="Contacts" />

   ~~<fragment android:id="@+id/contactDetailFragment"~~
   <fragment android:id="@+id/contact_detail"
             android:name="com.acme.contacts.detail.ContactDetailFragment"
             ~~android:label="ContactDetailFragment"/>~~
             android:label="Contact Details" />
</navigation>
```

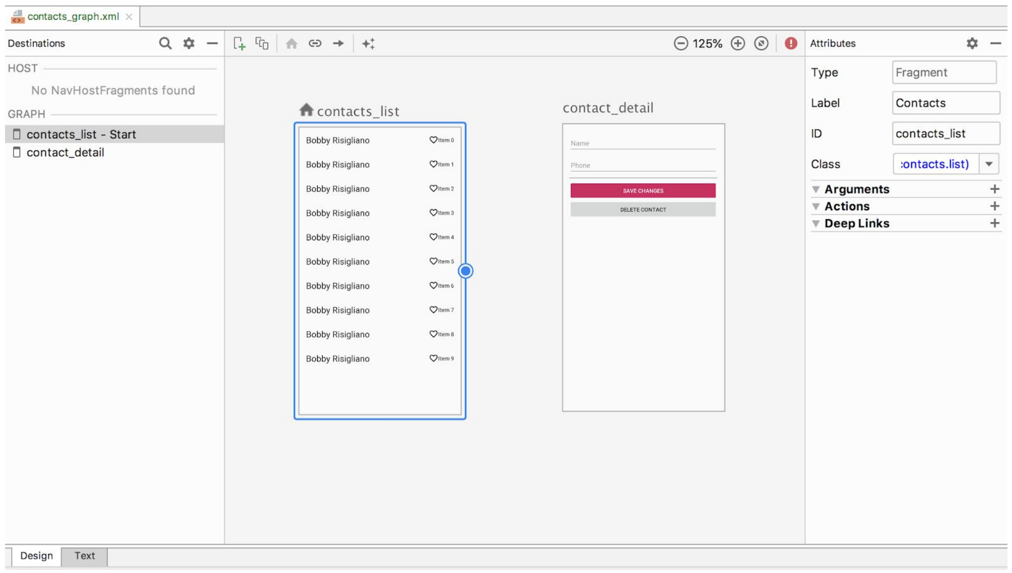
Adding Tools Layout Attributes

Finally, the visual navigation design editor can render the layout preview of your destinations if you add a ***tools:layout*** attribute to the destinations. Go ahead and do that now.

File (contact_graph.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<navigation 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/contacts_graph"
           app:startDestination="@id/contacts_list">

<fragment android:id="@+id/contacts_list"
          android:name="com.acme.contacts.list.ContactsListFragment"
          tools:layout="@layout/fragment_contacts_list"
          android:label="Contacts" />

<fragment android:id="@+id/contact_detail"
          android:name="com.acme.contacts.detail.ContactDetailFragment"
          tools:layout="@layout/fragment_contact_detail"
          android:label="Contact Details" />

</navigation>
```

Now if you go to the Design tab in the navigation editor, you should see something that looks like the following. Note how at a glance, you can see the layout and navigation of your app.



Type Fragment
Label Contacts
ID contacts_list
Class :ontacts.list)

Adding an Action

Now that you have configured your destinations, you can add an Action as a named route to get from the *contact_list* destination to the *contact_detail* destination. You will be referencing this action later when you want the Navigation framework to transition from the list to details screens.

Go back to the *Text* view of your navigation file if you're not there already and add the following:

File (contact_graph.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<navigation 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/contacts_graph"
           app:startDestination="@id/contacts_list">

   <fragment android:id="@+id/contacts_list"
         android:name="com.acme.contacts.list.ContactsListFragment"
         tools:layout="@layout/fragment_contacts_list"
         android:label="Contacts">

      <action android:id="@+id/to_contact_detail"
           app:destination="@id/contact_detail" />

   </fragment>

   <fragment android:id="@+id/contact_detail"
         android:name="com.acme.contacts.detail.ContactDetailFragment"
         tools:layout="@layout/fragment_contact_detail"
         android:label="Contact Details" />

</navigation>
```

This adds an action with an ***android:id*** value of *to_contact_detail* that you will reference in code as *R.id.to_contact_detail*. The ***app:destination*** attribute specifies the destination that the user will navigate to, when this action is followed. The value here should match the id of a destination in the graph. You will get a build time error if it does not.

Now that you have the navigation graph in place, it's time to host it within the application.

Hosting Navigation

Navigation requires a *NavHost* to host navigation within your app. Unless you're using custom, non-fragment based destinations within your app, you'll generally be using a framework supplied *NavHost class* called ***androidx.navigation.fragment.NavHostFragment***. There is nothing special per say about this fragment. It extends ***androidx.fragment.app.Fragment*** and implements the ***androidx.navigation.NavHost*** interface. However, it knows how to work with the fragment destinations that you define in your navigation graph and is doing all of the work to manage the navigation state within your app.

To get started using it, open ***activity_layout.xml*** and replace the frame layout element with a ***fragment*** element instead. Go ahead and make the following changes and then we'll talk about it.

File (activity_main.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
       xmlns:tools="http://schemas.android.com/tools"
       xmlns:app="http://schemas.android.com/apk/res-auto"
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       tools:context=".MainActivity">

~~   <FrameLayout~~
~~           android:id="@+id/nav_host"~~
~~           android:layout_width="match_parent"~~
~~           android:layout_height="match_parent"~~
~~           tools:layout="@layout/fragment_contacts_list" />~~

   <fragment
          android:id="@+id/nav_host"
          android:name="androidx.navigation.fragment.NavHostFragment"
          android:layout_width="match_parent"
          android:layout_height="match_parent"
          app:navGraph="@navigation/contacts_graph"
          app:defaultNavHost="true"
          tools:layout="@layout/fragment_contacts_list" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

The fragment element here represents a layout fragment that the Android framework will start and render as the activity's view is inflated. The ***android:name*** attribute specifies the fragment that will be inflated, ***androidx.navigation.fragment.NavHostFragment***.

All of the other attributes remain the same as before, however 2 new attributes are added, ***app:navGraph*** and ***app:defaultNavHost***. These attributes are defined by the navigation framework and the NavHostFragment uses these values to configure itself when it is created.

The ***app:navGraph*** element associates the contact navigation graph that you just created with the NavHostFragment.

The ***app:defaultNavHost*** attribute, when set to true, allows the Navigation framework to handle the back button. If you omit this attribute or set it to false, then hitting back will finish the

MainActivity and exit the app. Exiting the app is not what you would want the user to experience when pressing back from the contact detail screen, so here you've set it to true.

Navigating with NavController

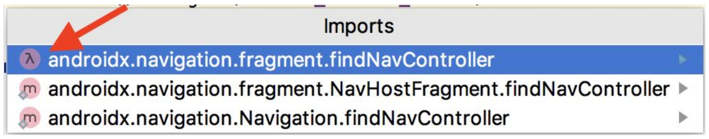
Now you have the major parts in place. You're ready to start using Navigation in your fragment destinations. You'll mostly be working with a class called the ***NavController***. The NavController knows how to interpret your navigation graph. It manages your applications navigation state and provides facilities to programmatically navigate to your apps destinations.

Open up ***ContactListFragment*** and replace the calls to *MainActivity.navToContactDetails* with *NavController.navigate(...)*.

File (ContactsListFragment.kt)

```
class ContactsListFragment : Fragment() {
   ...
   override fun onOptionsItemSelected(item: MenuItem): Boolean {
       return when(item.itemId) {
              ...
              R.id.to_contact_detail -> {
                  ~~(activity as MainActivity).navToContactDetails()~~
                  findNavController().navigate(R.id.to_contact_detail)
                  true
              }
              else ->  super.onOptionsItemSelected(item)
          }
   }

   private fun onContactItemClick(contact: Contact) {
       ~~(activity as MainActivity).navToContactDetails(contact.id)~~
       findNavController().navigate(R.id.to_contact_detail,
             Bundle().apply { putString("contactId", contact.id) })
   }
   ...
}
```

If presented with multiple import choices for findNavController(), choose the one with the lambda symbol.


Imports
androidx.navigation.fragment.findNavController
androidx.navigation.fragment.NavHostFragment.findNavController
androidx.navigation.Navigation.findNavController

Where did this ***findNavController()*** function come from? It is an extension function added to *androidx.fragment.app.Fragment* by the navigation framework. (If you're not using the *-ktx* version of the library, a static utility method is included that you can use instead).

Once you obtain a reference to the NavController, you call ***navigate(...)***, passing the id of the action you defined in the *contacts_graph* and in the case of *onContactItemClick(...)*, you pass the id of the contact for which to view/edit the details as bundle extra. The navigation framework will attach these to the destination fragment as *arguments*. The *NavController.navigate(...)* function has several overloaded ways to call it. At this point, you should be aware that you can pass a destination id, instead of an action id, to go directly to any destination in your graph. You don't specifically need to use an action here. The advantage of using and referencing an action is that you can tie additional configurations such as transition animations, default argument values, and other custom behaviors to actions declaratively in your nav graph, and avoid putting all of this into your Kotlin/Java code. Actions also provide a nice visual representation of destinations and how they are connected on your nav graph.

Backward Navigation

Next you'll want to update the ContactDetailFragment to call *NavController****.popBackStack()*** after saving or deleting a contact, rather than attempting to interact with the fragment manager itself. Open up ***ContactDetailFragment*** and make the following changes.

File (ContactDetailFragment.kt)

```
class ContactDetailFragment : Fragment() {
   ...
   var contactId: String? = null
   ...
   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       contactId = arguments?.getString("contactId")
   }
   ...
   fun onSaveContact() {
       contactDetailVm.saveContact()
       ~~requireActivity().supportFragmentManager.popBackStack()~~
       findNavController().popBackStack()
   }

   fun onDeleteContact() {
       contactDetailVm.deleteContact()
       ~~requireActivity().supportFragmentManager.popBackStack()~~
       findNavController().popBackStack()
   }
   ...
}
```

Finally, you can remove the manual navigation logic from your activity. Open ***MainActivity*** and remove ***navToContactDetails(...)*** and the logic in onCreate that loads the ContactListFragment in ***onCreate(...)***.

File (ContactDetailFragment.kt)

```
class MainActivity : AppCompatActivity() {

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContentView(R.layout.activity_main)

       ~~if(savedInstanceState == null) {~~
~~           supportFragmentManager.beginTransaction()~~
~~               .add(R.id.nav_host, ContactsListFragment())~~
~~               .commit()~~
~~       }~~
   }

  ~~fun navToContactDetails(contactId: String? = null) {~~

~~       val contactDetailFragment = ContactDetailFragment().apply {~~
~~           arguments = Bundle().apply {~~
~~               putString("contactId", contactId)~~
~~           }~~
~~       }~~

~~       supportFragmentManager~~
~~           .beginTransaction()~~
~~           .replace(R.id.nav_host, contactDetailFragment)~~
~~           .addToBackStack(null)~~
~~           .commit()~~
~~   }~~

}
```

Run the app and make sure everything works as it did before. If it doesn't double check the changes suggested in the lab.

You have the basic Navigation parts in place, and it's nice to not manage transactions manually using FragmentTransactions, but you're not making the most of Navigation yet. In the next few sections of this lab you'll be utilizing some useful features of Navigation, namely *Safe Args* to provide argument guarantees and type safety and *Navigation UI* to provide automatic options menu item navigation.

Safe Args Plugin

The Safe Args plugin provides code generation utilities that allows you to guarantee the presence of arguments and types as required for a given destination.

Why is this helpful?

Currently, ContactDetailFragment, assumes that it will be passed a contactId of type String in it's arguments bundle.

File (ContactDetailFragment.kt) -- Current Setup

```
class ContactDetailFragment : Fragment() {
   ...
   var contactId: String? = null
   ...
   override fun onCreate(savedInstanceState: Bundle?) {
       ...
       contactId = arguments?.getString("contactId")
   }
   ...
}
```

The presence of a value for *contactId* is not required in this case, because the app is programmed to treat a *null* value as an indication that this is a new contact. However, what if a value *is* passed, but it's an ***Int***?

Example

```
findNavController().navigate(R.id.to_contact_detail,
             Bundle().apply { putInt("contactId", 1) })
```

What would happen in *ContractDetailFragment.onCreate(...)*?

```
java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
```

Safe Args makes this mistake virtually impossible as long as you use the plugin on both sides of the equation.

Let's start using it.

Safe Args is a Gradle plugin which must be applied before you can use it.

Apply Plugin

Open the project ***build.gradle*** file and note the classpath dependency that has already been added there. In a new project, you would need to add this yourself.

File (build.gradle)

```
...
buildscript {
   ...
   dependencies {
        ...
        classpath "androidx.navigation:navigation-safe-args-gradle-plugin:2.1.0-beta02"
        ...
}
...
```

Next, open ***app/build.gradle***. Note the line applying the *androidx.navigation.safeargs.kotlin* plugin near the top.

File (app/build.gradle)

```
...
apply plugin: "androidx.navigation.safeargs.kotlin"

android {
   ...
}
...
```

This is all the setup required to use Safe Args in your app. Next you need to define the *contactId* argument details for *contact_detail* destination.

Defining Destination Arguments

Open ***contacts_graph.xml***. Add a new element type called ***argument*** as a child to the *contact_detail* destination's, ***fragment*** element.

File (contacts_graph.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<navigation 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/contacts_graph"
           app:startDestination="@id/contacts_list">
```

```
   <fragment android:id="@+id/contacts_list"
             android:name="com.acme.contacts.list.ContactsListFragment"
             tools:layout="@layout/fragment_contacts_list"
             android:label="Contacts">

       <action android:id="@+id/to_contact_detail"
               app:destination="@id/contact_detail"/>

   </fragment>

   <fragment android:id="@+id/contact_detail"
             android:name="com.acme.contacts.detail.ContactDetailFragment"
             tools:layout="@layout/fragment_contact_detail"
             android:label="Contact Details">

       <argument android:name="contactId"
                 app:argType="string"
                 app:nullable="true"
                 android:defaultValue="@null"/>

   </fragment>

</navigation>
```

This tells the Navigation framework that this destination takes an argument with the expected bundle key (***android:name***) and value type (***app:argType***). In this case, it is declared that this value may be null (***app:nullable***). Even though null is valid, a default value of *null* is provided by (***android:defaultValue="@null"***). This ***android:defaultValue*** attribute isn't completely required, but you'll soon discover its usefulness.

NavArgs

With Safe Args, any destination you define in your nav graph that contains one or more *argument* elements, will have a corresponding class generated that wraps its bundle arguments for easy storage and retrieval. The generated class is often called a *NavArgs* class because it implements a marker interface, *androidx.navigation.NavArgs*. The name for the generated class follows a naming convention set by the destination name. The format is *<destinationName>Args*. For example: ContactDetailFragment***Args***. By looking at the enclosing destination's ***android:name*** attribute value, you can infer what the name of the generated NavArgs class will be. In this case, the generated class looks like this:

File (ContactDetailFragmentArgs) [generated class]

```
data class ContactDetailFragmentArgs(val contactId: String? = null) : NavArgs {
   fun toBundle(): Bundle {
       val result = Bundle()
```

```
        result.putString("contactId", this.contactId)
        return result
    }

    companion object {
        @JvmStatic
        fun fromBundle(bundle: Bundle): ContactDetailFragmentArgs {
            bundle.setClassLoader(ContactDetailFragmentArgs::class.java.classLoader)
            val __contactId : String?
            if (bundle.containsKey("contactId")) {
                __contactId = bundle.getString("contactId")
            } else {
                __contactId = null
            }
            return ContactDetailFragmentArgs(__contactId)
        }
    }
}
```

Notice that this class wraps the ***contactId*** argument in a bundle. It provides constructor for creating an instance that takes the *contactId* in as a nullable String and provides a default so that the argument may be omitted. It also provides a mechanism for creating an instance of itself from an android.os.Bundle() and a way to transform it back out to a Bundle. Finally, because you're using the kotlin version of the Safe Args plugin, the generated class is generated as a Kotlin data class.

Now just a moment ago, when you defined the ***action***, we said that the ***android:defaultValue="@null"*** attribute was useful and you'd soon discover why.

Recall the argument definition:

```
<argument android:name="contactId"
          app:argType="string"
          app:nullable="true"
          android:defaultValue="@null"/>
```

By adding this ***android:defaultValue*** attribute, the plugin generates a constructor like this

```
ContactDetailFragmentArgs(val contactId: String? = null)
```

This constructor allows callers to instantiate an instance of ContactDetailFragmentArgs as:

```
ContactDetailFragmentArgs()
```

If you omit the defaultValue attribute, the generated result is:

```
ContactDetailFragmentArgs(val contactId: String?)
```

Which means that all callers must explicitly provide a value, even if that value is *null* like so:

```
ContactDetailFragmentArgs(null)
```

The NavArgs class is most useful on the destination side of the equation. It allows you to wrap the fragment's Bundle *arguments* and pull them out as type safe values. Of course none of that is very useful if the Bundle doesn't contain the values or the values aren't of the type which the NavArgs class now expects. This is where the other type of generated class, *NavDirections*, come into play.

NavDirecitons

NavDirections are generated for every action declaring destination in your graph. Let's take a look to see what has been generated for this app.

Recall that in *contacts_graph.xml*, you declared an action inside of the *contacts_list* destination that takes the user to the contact_detail destination.

File (contacts_graph.xml) -- Current Setup

```
<?xml version="1.0" encoding="utf-8"?>
<navigation ...>

   <fragment android:id="@+id/contacts_list"
             android:name="com.acme.contacts.list.ContactsListFragment"
             tools:layout="@layout/fragment_contacts_list"
             android:label="Contacts">

       <action android:id="@+id/to_contact_detail"
               app:destination="@id/contact_detail"/>

   </fragment>

   <fragment android:id="@+id/contact_detail"
             android:name="com.acme.contacts.detail.ContactDetailFragment"
             tools:layout="@layout/fragment_contact_detail"
             android:label="Contact Details">

       <argument android:name="contactId"
                 app:argType="string"
                 app:nullable="true"
                 android:defaultValue="@null"/>

   </fragment>
</navigation>
```

Based off of this information, the Safe Args plugin generated a class, ContactsListFragment***Directions***, (*it already happened!*). Similar to *NavArgs*, the naming pattern is *<destinationName>Directions.*

You can explore the course code for ContactsListFragmentDirections just as you did for ContactDetailFragmentArgs.

File (ContactDetailFragmentDirections) [generated class]

```
class ContactsListFragmentDirections private constructor() {
   private data class ToContactDetail(val contactId: String? = null) : NavDirections {
       override fun getActionId(): Int = R.id.to_contact_detail

       override fun getArguments(): Bundle {
           val result = Bundle()
           result.putString("contactId", this.contactId)
           return result
       }
   }

   companion object {
       fun toContactDetail(contactId: String? = null): NavDirections =
                   ToContactDetail(contactId)
   }
}
```

Notice the nested private data class defined inside. You'll find one nested data class for each action you define within the source destination (*ContactsListFragment*). In this specific example, only one action was defined, so there is only one nested data class, ***ToContactDetail***.

ToContactDetail wraps all of the metadata defined by its corresponding action and provides a way to package and send the arguments expected by the target destination. Notice that the constructor for ContactsListFragmentDirections is private. The only way to construct an instance of it is to use the static companion function.

Here again, setting the ***android:defaultValue*** attribute in the target destination's ***argument*** definition gives generated companion function a default value for the contactId argument.

```
fun toContactDetail(contactId: String? = null)
```

If you had not set the ***android:defaultValue*** attribute, the generated companion function would have looked like this.

```
fun toContactDetail(contactId: String?)
```

Take a moment to try this out for yourself. You may need to build the project to force the Safe Args classes to regenerate.

Putting it all together

To put NavArgs to good use, open ***ContactsListFragment*** and update the *NavController.navigate(...)* calls to use *ContactsFragmentListFragmentDirections* instead.

File (ContactsListFragment.kt)

```
import com.acme.contacts.list.ContactsListFragmentDirections.Companion.toContactDetail
...
class ContactsListFragment : Fragment() {
   ...
   override fun onOptionsItemSelected(item: MenuItem): Boolean {
       return when(item.itemId) {
              ...
              R.id.to_contact_detail -> {
                  ~~findNavController().navigate(R.id.to_contact_detail)~~
                  findNavController().navigate(toContactDetail())
                  true
              }
              ...
          }
   }

   private fun onContactItemClick(contact: Contact) {
       ~~findNavController().navigate(R.id.to_contact_detail,~~
           ~~Bundle().apply { putString("contactId", contact.id) })~~
       findNavController().navigate(toContactDetail(contactId = contact.id))
   }
   ...
}
```

Next, modify *ContactDetailFragment* to use *ContactDetailFragmentArgs* in order to receive the contactId argument.

File (ContactDetailFragment.kt)

```
class ContactDetailFragment : Fragment() {
   ...
   ~~var contactId: String? = null~~
   val args by navArgs<ContactDetailFragmentArgs>()
   ...
   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
      ~~contactId = arguments?.getString("contactId")~~
   }
   ...
   override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
       ...
       binding.apply {
            ...
           mode = if (args.contactId.isNullOrEmpty()) Mode.NEW else Mode.EDIT
           ...
       }
   }

   var viewModelFactory = object : ViewModelProvider.Factory {
       override fun <T : ViewModel?> create(modelClass: Class<T>): T {
           return ContactDetailViewModel(args.contactId) as T
       }
   }
}
```

In the code above, you replaced the *contactId* member variable with an args member variable. You may be wondering what this by navArgs bit is though.

```
      by navArgs<ContactDetailFragmentArgs>()
```

The *-ktx* version of the Navigation library provides a Kotlin property delegate to simplify the creation of the NavArgs class. It's just a bit of syntactic sugar. If you were using the Java version of Navigation library, you would do something like this in your *onCreate()* function instead.

Example

```
lateinit var args: ContactDetailFragmentArgs

override fun onCreate(savedInstanceState: Bundle?) {
   ...
   args = ContactDetailFragmentArgs.fromBundle(requireArguments())
}
```

Checkpoint

Run the app again to verify that everything is working.

Navigation UI

Finally, let's utilize NavigationUI to simplify the navigation that takes place when the add contact menu item is clicked. Open *ContactsFragmentList* and make the following changes.

File (ContactsListFragment.kt)

```
class ContactsListFragment : Fragment() {
   ...
   override fun onOptionsItemSelected(item: MenuItem): Boolean {
          return item.onNavDestinationSelected(findNavController()) ||
            when(item.itemId) {
              R.id.toggle_favorites_only -> {
                  ...
              }
              ~~R.id.to_contact_detail -> {~~
                  ~~findNavController().navigate(toContactDetail())~~
                  ~~true~~
              ~~}~~
              else ->  super.onOptionsItemSelected(item)
          }
   }
  ...
}
```

Here, you updated *onOptionsItemSelected* to give the Navigation framework first dibs at trying to handle the event. `onNavDestinationSelected(NavController)` attempts to match the *itemId* of the menu item to an action or destination with the same *id*.

File (contacts_list_option_menu.xml) -- Options Menu

```
<?xml version="1.0" encoding="utf-8"?>
<menu...>
  ...
  <item android:id="@+id/to_contact_detail"
      android:icon="@drawable/ic_contact_add_white"
      android:title="@string/title_contact_add"
      app:showAsAction="ifRoom|withText" />
</menu>
```

File (contacts_graph.xml) -- Navigation Graph

```
<?xml version="1.0" encoding="utf-8"?>
<navigation...>
  <fragment android:id="@+id/contacts_list"
        android:name="com.acme.contacts.list.ContactsListFragment"
        tools:layout="@layout/fragment_contacts_list"
        android:label="Contacts">

    <action android:id="@+id/to_contact_detail"
         app:destination="@id/contact_detail"/>
  </fragment>
   ...
</navigation>
```

If a match is found, it navigates the user to the target destination and `NavigationUI.onNavDestinationSelected(MenuItem, NavController)`returns *true*. Otherwise, it returns *false* and gives you a chance to handle the menu item click manually.

This is a small example, but in a larger app with more menu items, this can save you from writing a lot of boilerplate code to wire up menu item clicks to navigation calls. It should be noted however, that this method is only suited to use cases where you do not need to pass arguments to the target destination.

NavigationUI vs NavigationUI *-ktx*

You may be wondering what this last section had to do with NavigationUI, when it wasn't mentioned even once in code. You may have also been surprised to see the MenuItem class with this new function suddenly `onNavDestinationSelected(NavController)`.

You're not going crazy. The *-ktx* version of the NavigationUI library adds a bunch of extension functions to existing Android APIs, to sweeten the syntax a bit.

> Calling
> `item.onNavDestinationSelected(findNavController())`
>
> Has the same effect as calling
> `NavigationUI.onNavDestinationSelected(item, findNavController())`

The latter of which, is what you would call from Java, if you were using the non -ktx version of the library.

Lab 2: Top Level Navigation & Testing

In this lab you'll be exploring top level navigation. You'll add a BottomNavigationView to your apps MainActivity and learn how to use NavigationUI to trigger navigation in response to navigation item selections. You'll also clean up a few things, adding Up navigation to the contact details screen and hiding the BottomNavigation menu on the contact details screen.

Lab Setup:

If you completed Lab 1, you can continue forward. If you didn't complete Lab 1, you can checkout the Lab 2 starter branch to catch up.

Checking out Lab 2 Starter

You only need to do this if you didn't finish lab 1.

Revert anything you've done to this point, by running *git reset* and *checkout* from the root of the project.

```
git reset --hard head
git checkout .
(you may also have to remove any resources you added)
git checkout starter/lab_2
```

Getting Started

There are a few things you just need to add to the project before moving forward. They don't have to do with Navigation, but are required for the lab. Steps 5 and 6 are optional but suggested for consistency with the lab instruction.

1. Add a few strings to your ***strings.xml*** file.

   File (strings.xml)

```
<resources>
  ...
  <string name="toggle_fav">Toggle Favorite</string>
  <string name="title_home">All Contacts</string>
  <string name="title_settings">Settings</string>
  <string name="title_contact_add">Add Contact</string>
  ....
</resources>
```

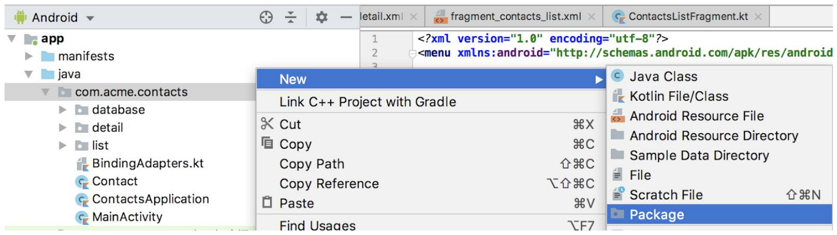
2. Create a new menu resource file named ***bottom_nav_menu.xml***. This should be a peer to the *contacts_list_option_menu.xml*. Replace the contents of bottom_nav_menu.xml with the following:

   File (bottom_nav_menu.xml)

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

   <item
           android:id="@+id/contacts_list"
           android:icon="@drawable/ic_contact"
           android:title="@string/title_home"/>

   <item
           android:id="@+id/settings"
           android:icon="@drawable/ic_settings"
           android:title="@string/title_settings"/>

</menu>
```

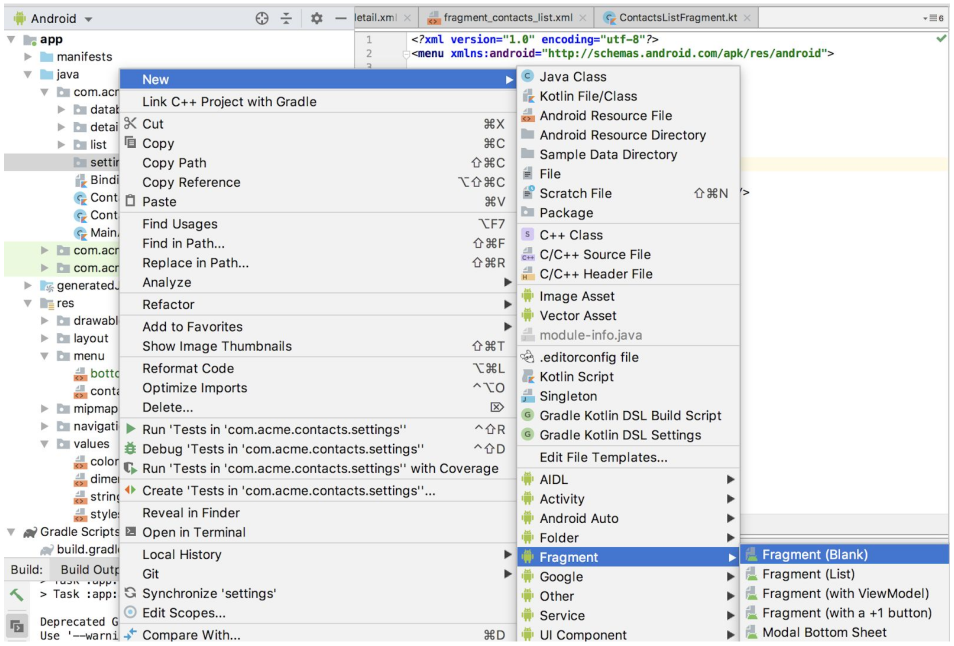
   Note: *drawables (ic_settings, ic_contact) were already added to the project for you.*

3. Add a new package, ***settings***, as a peer to *detail* and *list*.

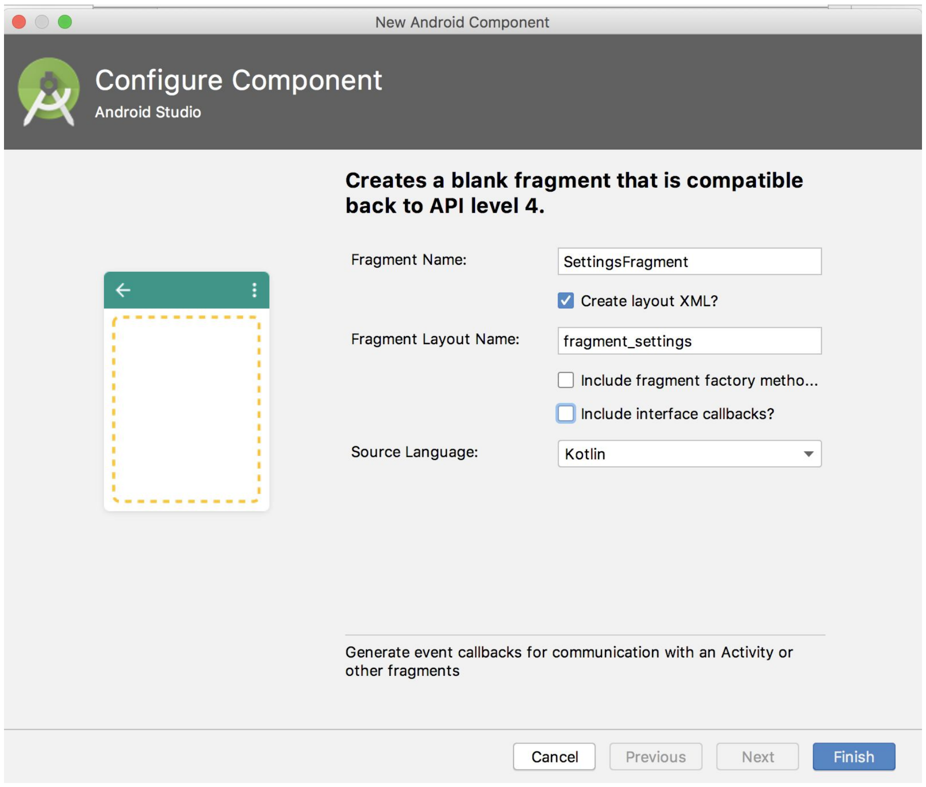


```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android
```

4. Right click on the settings package and add a new fragment, ***SettingsFragment***, using the **New => *Fragment (Blank) template*** as shown below.



Set the values exactly as shown in the screenshot below and click ***okay***.



Creates a blank fragment that is compatible back to API level 4.

Fragment Name: SettingsFragment

[✓] Create layout XML?

Fragment Layout Name: fragment_settings

[ ] Include fragment factory metho...

[ ] Include interface callbacks?

Source Language: Kotlin

Generate event callbacks for communication with an Activity or other fragments

The important thing here is that the names are consistent and the “Create layout XML checkbox is checked”.

5. (Optional) Open ***fragment_settings.xml*** and make the following changes.

File (fragment_settings.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout 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"
             tools:context=".settings.SettingsFragment">

   ~~<!-- TODO: Update blank fragment layout -->~~
   <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:gravity="center"
            ~~android:text="@string/hello_blank_fragment" />~~
            android:text="@string/title_settings"/>

</FrameLayout>
```

6. (Optional) Open ***SettingsFragment***. If the text "*TODO: Rename params…"* is present with 2 constants defined below it, go ahead and delete those lines as shown below.

File (SettingsFragment.kt)

```
~~// TODO: Rename parameter arguments, choose names that match~~
~~// the fragment initialization parameters, e.g. ARG_ITEM_NUMBER~~
~~private const val ARG_PARAM1 = "param1"~~
~~private const val ARG_PARAM2 = "param2"~~

...
class SettingsFragment : Fragment() {
   ...
}
```

With the basic building blocks in place, you're ready to add top level navigation to the app.

Adding Top Level Navigation

Open the activity layout (***activity_main.xml***) and add a ***BottomNavigationView*** that uses the *bottom_nav_menu* as it's *app:menu*.

File (activity_main.xml)

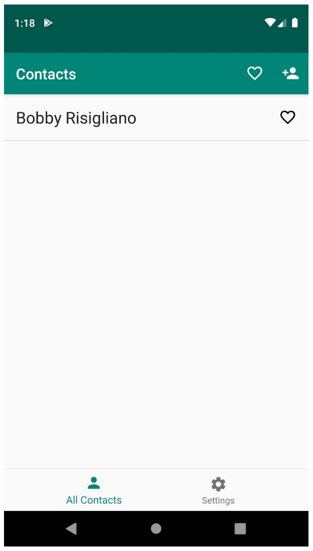
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    ...>
```

```
  <fragment
       android:id="@+id/nav_host"
       android:name="androidx.navigation.fragment.NavHostFragment"
      ... />

  <com.google.android.material.bottomnavigation.BottomNavigationView
       android:id="@+id/bottom_nav_view"
       android:layout_width="0dp"
       android:layout_height="wrap_content"
       android:layout_marginEnd="0dp"
       android:layout_marginStart="0dp"
       android:background="?android:attr/windowBackground"
       app:layout_constraintBottom_toBottomOf="parent"
       app:layout_constraintLeft_toLeftOf="parent"
       app:layout_constraintRight_toRightOf="parent"
       app:menu="@menu/bottom_nav_menu"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

There are a few additional style and layout related attributes too, the details of which are not important for the purposes of Navigation.

Checkpoint

Run the app to make sure everything is setup correctly. You should see something similar to what you see in this screenshot. If the project doesn't build or you see something different, double check your setup.



Note that clicking on the menu item does not change the destination yet. Let's fix that!

Adding the Settings Destination

First you need to add the SettingsFragment as a destination in your navigation graph. Open ***contacts_graph.xml*** and add a new fragment destination as shown in the diagram below.

File (contacts_graph.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<navigation 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/contacts_graph"
           app:startDestination="@id/contacts_list">

   <fragment android:id="@+id/contacts_list"...>
       ...
   </fragment>

   <fragment android:id="@+id/contact_detail"...>
       ...
   </fragment>

   <fragment android:id="@+id/settings"
             android:name="com.acme.contacts.settings.SettingsFragment"
             android:label="Settings"
             tools:layout="@layout/fragment_settings" />

</navigation>
```

You will not be adding an *action* to take the user to settings. Instead, you will be relying on *NavigationUI* to handle navigation to this destination in response to the user clicking on one of the items in your *BottomNavigationView*.

Just like you did in Lab 1, you set the ***android:id*** attribute value to match the value of the menu item.

File (bottom_nav_menu.xml) -- Abbreviated Snippet

(nothing to do here, but recall that the menu ids, match the destination ids)

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
   <item android:id="@+id/contacts_list"...>
   <item android:id="@+id/settings"...>
</menu>
```

Now, you're ready to wire this up in your MainActivity.

Adding Navigation Using NavigationUI

Open the ***MainActivity*** class and make the following updates.

File (MainActivity.kt)

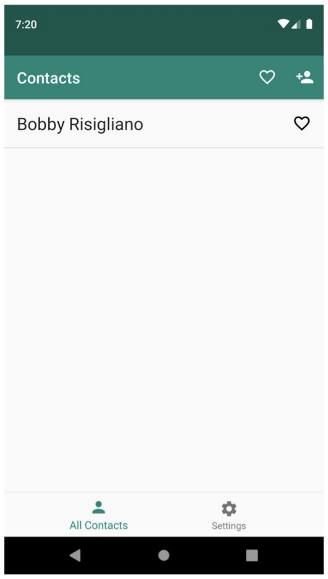
```
class MainActivity : AppCompatActivity() {

   lateinit var navController: NavController
   lateinit var bottomNavigationView: BottomNavigationView

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContentView(R.layout.activity_main)

       navController = findNavController(R.id.nav_host)
       bottomNavigationView = findViewById(R.id.bottom_nav_view)
       bottomNavigationView.setupWithNavController(navController)
   }

}
```

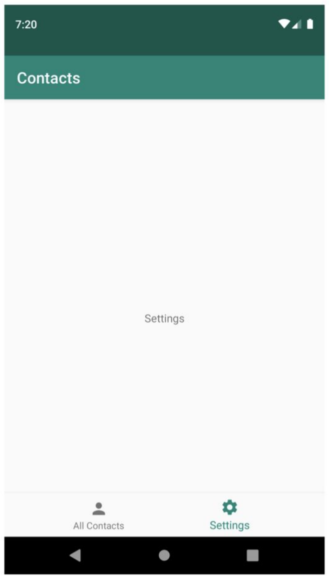
Inside, you added a couple of class member variables to store references to the *BottomNavigationView* and *NavController*. In *onCreate(...)*, you set up the association between the two so that the framework can handle navigation for you. Similar to what you did in lab 1 with the options MenuItem, here you take advantage of the NavigationUI *-ktx* extension function that was added to *BottomNavigationView*.

Note: *You could have used data binding in the example above, to avoid the single `findViewById` call. I opted not to in this example because it's only saving one findViewById lookup and thought it made the example a bit more straightforward.*

Checkpoint

Run your app! You should be able to see that the app navigates between the List and Settings destinations.

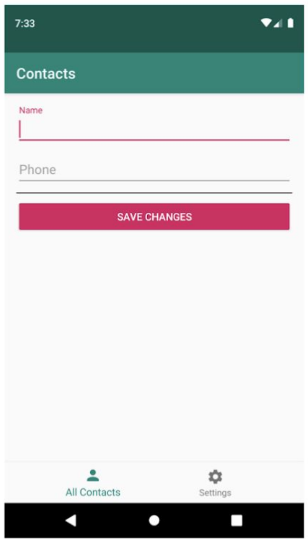




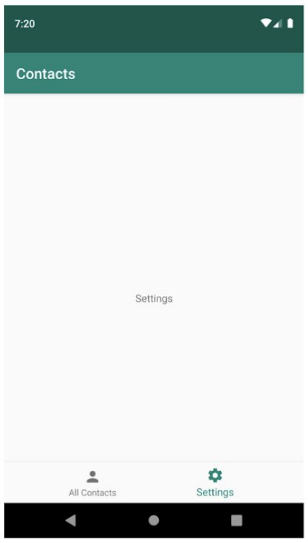
Woot!

There are a couple of issues though. Try adding a contact by clicking on the add contacts ( ) button. Notice the bottom nav menu is still present.

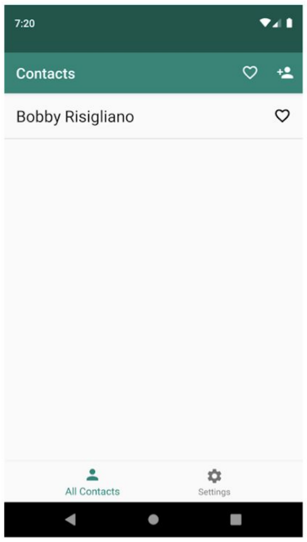
What happens if you click ***settings*** and then hit the ***back*** button to go back?



1. Click Settings



2. Click Back



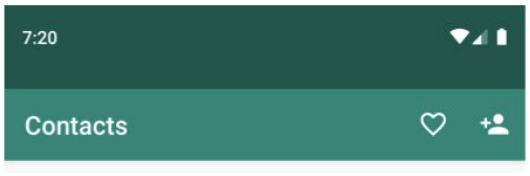
3. Yikes, where is my contact?

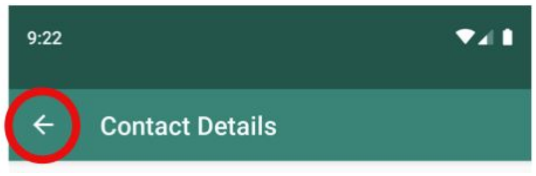
😱What happened to the contact details screen?

The root cause of this issue is that the Navigation framework does not maintain separate destination stacks for each of the top level navigation destinations. When you click on *settings*, the NavController pops all destinations off the stack until it reaches the start destination and then pushes the *settings* destination on top. This is why, when you click *back***,** you end up back at the start destination again (*contacts_list*).

Navigation does not support having separate stacks of navigation inside each top level destination, however there is an Advanced Navigation Google Sample that shows how you can do this with a little extra work in your app. We're going to solve this problem a different way though, by making a few adjustments to the way we present the contact details screen.

If you think about it, the contact details screen is really a child of the contacts list screen. This hierarchical relationship carries with it the implication that the user should be able to navigate up to the contacts lists screen using an ***Up*** button as shown in the image below.





The Up button is part of Material Design navigation patterns and should behave in a very specific way. See the Principles of Navigations for more detail. The really nice thing about the Navigation Architecture Component is that it can implement the proper behavior for you.

Adding Up Navigation

To add an Up button to the toolbar, open ***MainActivity*** and make the following changes.

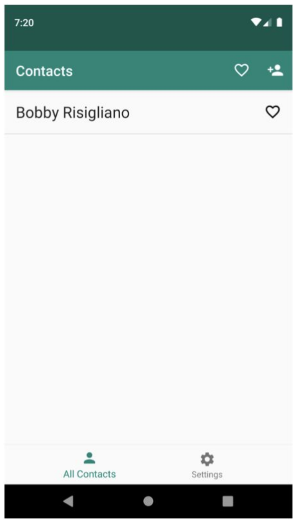
File (MainActivity.kt)

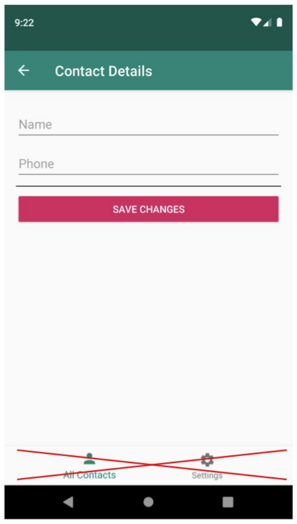
```
class MainActivity : AppCompatActivity() {
   ...
   override fun onCreate(savedInstanceState: Bundle?) {
       ...
       bottomNavigationView.setupWithNavController(navController)

       // Adds the Up Button to the Toolbar
       setupActionBarWithNavController(navController)
   }

    // Provides up navigation when the up button is clicked
   override fun onSupportNavigateUp(): Boolean {
      return navController.navigateUp() || super.onSupportNavigateUp()
   }
}
```

You call *setupActionBarWithNavController(...)* call in *onCreate()* to configure the Toolbar with the Up button. The Up button will appear on any **non** top-level destination, which is currently just the *contacts_list* destination.

You then override the *onSupportNavigateUp()* function, giving a chance for the NavController to handle the navigation up. If the NavController successfully navigated Up, then it returns *true* and you return that value, otherwise you call up to the activity's super implementation. If you forget to override this function, you will see the Up button, but clicking on it will have no affect.

Adding an Up button is key to providing a proper Material UI/UX, but it won't solve the issue of navigating away from the contact details screen when the user clicks settings. For that, we're going to hide the BottomNavigationMenu on the contact details screen.





Hiding the BottomNavigationView

Inside ***MainActivity.onCreate(...)***, add a *Navigation.OnDestinationChangedListener* to the NavController.

File (MainActivity.kt)

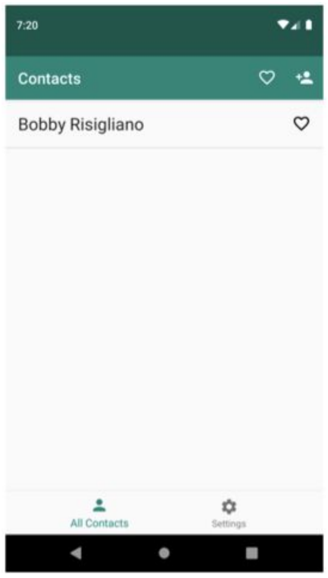
```
class MainActivity : AppCompatActivity() {
   ...
   override fun onCreate(savedInstanceState: Bundle?) {
       ...
       setupActionBarWithNavController(navController)

       navController.addOnDestinationChangedListener {
                                navController, newDestination, arguments ->
           when (newDestination.id) {
               R.id.contact_detail -> bottomNavigationView.visibility = View.GONE
               else -> bottomNavigationView.visibility = View.VISIBLE
           }
       }
   }
   ...
}
```

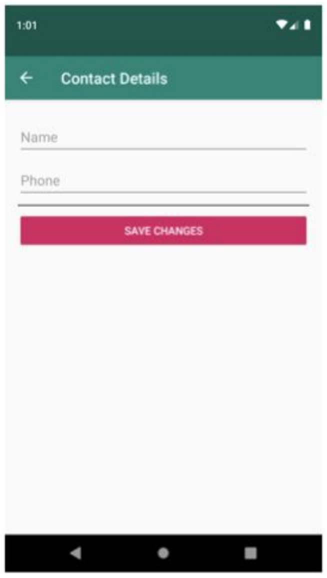
This code adds a listener that will be notified anytime a destination change occurs in your app. This listener fires after the change has occurred.

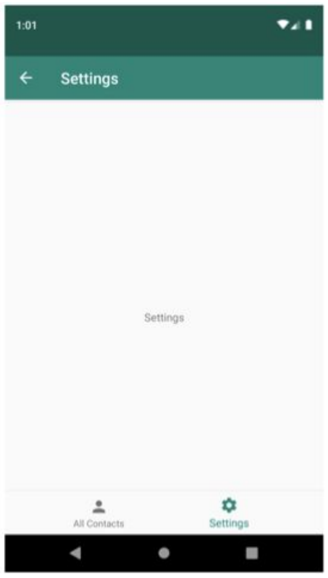
You call *navController.addOnDestinationChangedListener* and pass a lambda that matches the Single Abstract Method (SAM) interface of ***Navigation.OnDestinationChangedListener***. The lambda receives a reference to the NavController, the new destination to which the user just navigated, and the arguments supplied to that destination as an *android.os.Bundle*.

When the id of the new destination is equal to *contact_detail* you hide the *BottomNavigationView*, otherwise you show it.

Run the app now and you'll see the bar goes away when you go to the contact details screen, but otherwise shows up.







There's one last thing to tidy up. Can you see it from the images above?

It's the Up button. We've added it globally in the MainActivity so it will show up for any destination that isn't a top-level destination, including the settings screen. ***This is not actually wrong from a Material UI/UX perspective***. However, if you wanted to consider the settings screen to be a top level destination as well, you can configure it as such by creating and configuring an AppBarConfiguration and passing that along with your call to *setupActionBarWithNavController(...)*.

File (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {

   lateinit var navController: NavController
   lateinit var appBarConfiguration: AppBarConfiguration
   lateinit var bottomNavigationView: BottomNavigationView

   override fun onCreate(savedInstanceState: Bundle?) {
       ...
       ~~setupActionBarWithNavController(navController)~~
       appBarConfiguration =
             AppBarConfiguration(setOf(R.id.contacts_list, R.id.settings))
       setupActionBarWithNavController(navController, appBarConfiguration)

       navController.addOnDestinationChangedListener { ... ->
          ...
       }
   }
   ...
}
```

You pass the top level destination ids as a *Set*, to the *AppBarConfiguration* constructor.

Run the app again, and you'll see that the Up arrow no longer shows on the Settings screen.

More information about top level navigation options and configurations is available at developer.android.com/guide/navigation/navigation-ui.

Challenge: Testing Navigation

This section is optional. You can continue into lab 3 without completing it. This section will cover the basics of testing Navigation within your app.

The recommended approach to testing navigation, as covered at developer.android.com/guide/navigation/navigation-testing, is to use a mocking framework such as Mockito and validate your apps interactions a mock NavController instance. In order to test the fragments in isolation, a fragment testing helper class called FragmentScenario is recommended.

In this section you'll be using the recommended testing approach to verify that

- ***When:*** a contact in the list of contacts on the *contact_list* destination is clicked.
  Then: NavController.navigate(...) is invoked with the corresponding contactId.

- ***When:*** the contact_detail destination is launched, with a valid contactId.
  Then: the *name* and *number* text fields render the proper values
  And: NavController.popBackStack() is invoked after the user clicks the *save* button.

Adding Navigation Testing Dependencies

Open the **app/build.gradle** file and add notice the two dependencies highlighted below

```
dependencies {
  …
  debugImplementation ('androidx.fragment:fragment-testing:1.2.0-alpha01', {
    exclude group: 'androidx.test', module: 'core'
  })
  androidTestImplementation 'org.mockito:mockito-android:2.27.0'
  ...
}
```

The ***androidx.fragment:fragment-testing*** dependency pulls in the FragmentScenario class that you'll be using to initialize your fragments and set the mock NavController instance.

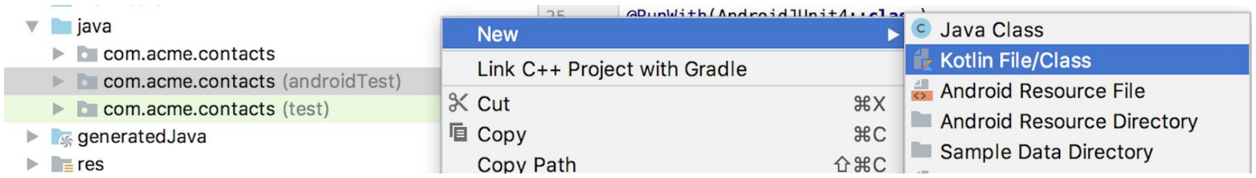
The ***org.mockito:mockito-android*** dependency pulls in the Mockito API, which allows you to create a mock instance of the NavController.

Note: *Not all required testing dependencies are shown, only these two related to testing Navigation. A more complete list is available developer.android.com/training/testing/set-up-project. This project uses an alpha version of fragment-testning. However, FragmentScenario is supported in the latest stable version of fragment-testing.*

With the dependencies in place you're ready to write the tests.

Writing Tests

Create a new instrumented test by ***right clicking on the android test folder*** and selecting ***new => Kotlin Class/File***.



Name the file ***ContactNavigationTest*** and ***replace the contents*** with the following.

File (ContactNavigationTest.kt)

```
package com.acme.contacts

import androidx.test.ext.junit.runners.AndroidJUnit4
import com.acme.contacts.database.ContactsRepository
import org.junit.Before
import org.junit.Test
import org.junit.runner.RunWith

@RunWith(AndroidJUnit4::class)
class ContactNavigationTest {

   val expectedContact = Contact(
       id = "1",
       name = "Test Contact",
       phone = "555-555-5555",
       isFavorite = true
   )

   @Before
   fun initDatabase() {
       ContactsRepository.setupForTesting(ApplicationProvider.getApplicationContext())
      val repository = ContactsRepository.get()
      repository.saveContact(expectedContact)
   }
   @Test
   fun testNavigateToContactDetails() {

   }

   @Test
   fun testNavigateBackOnSave() {

   }
}
```

Inside of the test class, you created an instance of a contact (***expectedContact***) and in *initDatabase()* you saved it to an in memory instance of the database. This code will run before each of the two tests which you stubbed out below that.

Note: *setupForTesting is a custom function added to this test project that will override the default Repository configuration to work against an in memory Room database instance. Refer to the source of the function to see what it is doing, but this is out of scope for learning Navigation. The important thing to know is that these tests are running against an in memory Database that is recreated before each test is run.*

Let's start with our first test. The goal of the test is to verify that:

- ***When:*** a contact in the list of contacts on the *contact_list* destination is clicked.
  Then: NavController.navigate(...) is invoked with the corresponding contactId.

Update the ***testNavigateToContactDetails***() function, following each of the steps 1-6 below. For now just go ahead and type them in. We'll explain after.

File (ContactNavigationTest.kt)

```
@RunWith(AndroidJUnit4::class)
class ContactNavigationTest {

   val expectedContact = Contact(
     ...
   )
   ...
   @Test
   fun testNavigateToContactDetails() {

       // 1. Create a mock instance of NavController
       val mockNavController = mock(NavController::class.java)

       // 2. Create FragmentScenario instance, parameterized as a ContactsListFragment.
       val fragmentScenario = launchFragmentInContainer<ContactsListFragment>()

       // 3. Wait for the fragment to become resumed.
```

```
        fragmentScenario.onFragment { fragment ->

            // 4. Replace the NavController with a mock instance.
            Navigation.setViewNavController(
                fragment.requireView(), mockNavController
            )
        }

        // 5. Click on the first item in the contact list.
        //    It should be the one you inserted in `initDatabase()`
        onView(withId(R.id.rv_contacts_list))
            .perform(actionOnItemAtPosition<ContactViewHolder>(0, click()));

        // 6. Verify that NavController.navigate(...) is called
        // with the contactId you expected.
        verify(mockNavController).navigate(toContactDetail(expectedContact.id))
    }
    ...
}
```

Step 1: You used Mocito to create a “mock” instance of a NavController. Mockito subclasses the NavController class and intercepts calls to any of its methods or properties. By creating a mock instance, you can validate interactions and force it to return fake values to assist with testing.

Step 2: You created a FragmentScenario instance using the *launchFragmentInContainer* function. This function was parameterized to be of type *ContactsListFragment*. FragmentScenario is used in your test to launch *ContactsListFragment* inside a dummy activity and run the fragment through its lifecycle to the RESUMED state. We’re just scratching the surface of using FragmentScenario. For more examples and documentation, check out the docs at developer.android.com/training/basics/fragments/testing.

Step 3: You called *fragmentScenario.onFragment* and passed a lambda. This lambda, known as the *action*, will be invoked on the main thread once the fragment has reached the RESUMED state. The instance of *ContactsListFragment* under test, is passed into the lambda as an argument.

Step 4: Inside the action, you use a utility method, *Navigation.setViewNavController(...)*, to replace the existing NavController instance with your mock version. Note that *fragmentScenario.onFragment* blocks until after your *action* lambda finishes execution. Hence, the rest of this test does not execute until your fragment is RESUMED and the mock NavController instance is set.

Step 5: Using the Espresso API, you click the first item in the list.

Step 6: You verify with the mock NavController, that navigate was called with the expected *NavDirections* instance.

Using a similar approach you can write the second test.

- ***When:*** the contact_detail destination is launched, with a valid contactId.
  Then: the *name* and *number* text fields render the proper values
  And: NavController.popBackStack() is invoked when the user clicks the *save* button.

There are a couple of differences to note however. For this test, you'll need to provide type safe arguments to *ContactDetailFragment* when you launch it and validate that the fragment received it by verifying that the name and phone number fields populate appropriately. Also, the view for ContactDetailFragment uses *TextInputLayout*s and *TextInputEditText*s which require the AppCompat Theme.

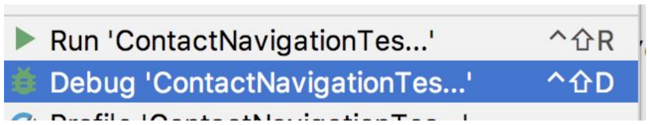
Make the updates noted below to the *testNavigateBackOnSave* function.

File (ContactNavigationTest.kt)

```
@RunWith(AndroidJUnit4::class)
class ContactNavigationTest {

   val expectedContact = Contact(
       id = "1",
       name = "Test Contact",
       phone = "555-555-5555",
       isFavorite = true
   )
   ...
   @Test
   fun testNavigateBackOnSave() {

       val mockNavController = mock(NavController::class.java)

       // android.os.Bundle created using the ContactDetailFragmentArgs class to ensure
type safety.
       val arguments = ContactDetailFragmentArgs(contactId =
expectedContact.id).toBundle()
       // Here you pass to FragmentScenario the arguments along with the theme to use.
       // In this case you pass R.style.AppTheme which is defined
       // in your styles.xml resource folder and is a descendant
       // of the AppCompat theme.
       val fragmentScenario =
           launchFragmentInContainer<ContactDetailFragment>(
              arguments, R.style.AppTheme)

       fragmentScenario.onFragment { fragment ->
```

```
            Navigation.setViewNavController(
                fragment.requireView(), mockNavController)
        }

        onView(withId(R.id.contact_name_tv)).check(
              matches(withText(expectedContact.name)));
        onView(withId(R.id.contact_number_tv)).check(
              matches(withText(expectedContact.phone)));

        onView(withId(R.id.save_contact_btn)).perform(click())
        verify(mockNavController).popBackStack()
    }
}
```

Run the test by right clicking on the file name in the project explorer and selecting the run or debug option from the context menu.


Run 'ContactNavigationTes...' ^⇧R
Debug 'ContactNavigationTes...' ^⇧D

You will be prompted to choose an emulator to run them on. Go ahead and pick the emulator you've been using to run the app.

Important Notes

- You tested two different fragments from this single test class. Ordinarily, you would want to create a separate test class for each fragment under test. We combined them into one here for simplicity.

Lab 3: Deep Linking & Conditional Navigation

In this final lab, you'll learn how to create an explicit deep link into your application and use it to build a Notification. You'll also explore conditional navigation and see how you might put conditional navigation into practice, by securing the contacts app.

Lab Setup:

If you completed the required parts of Lab 2, you can continue forward. If you did not complete Lab 2, you can checkout the Lab 3 starter branch to catch up.

Checking out Lab 3 Starter

You only need to do this if you didn't finish lab 2.

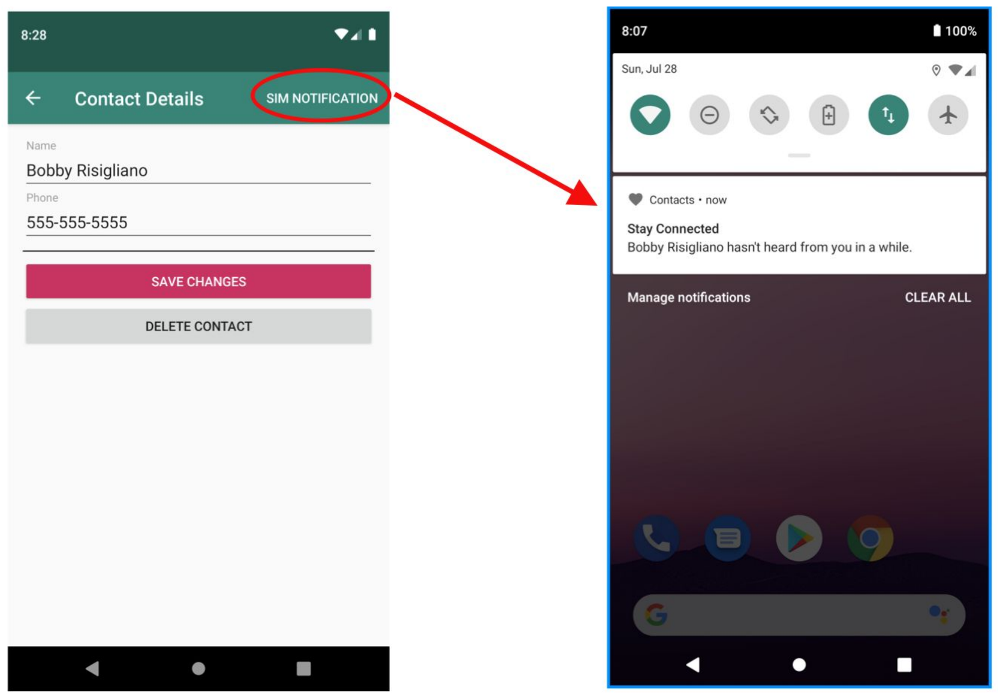
Revert anything you've done to this point, by running *git reset* and *checkout* from the root of the project.

```
git reset --hard head
git checkout .
(you may also have to remove any resources you added)
git checkout starter/lab_3
```

Explicit Deep Linking

In the first part of this lab you will be using the NavController to create a PendingIntent. You'll use that PendingIntent to create a Notification. This Notification will show some engaging message to the user, encouraging them to come back into the app and see the contact details for one of their friends. Clicking on that notification will bring them back into the app on the contact details screen populated with the contact's details.

The final result will look similar to this:



Name
Bobby Risigliano
Phone
555-555-5555

Contacts • now
Stay Connected
Bobby Risigliano hasn't heard from you in a while.

Lab Setup

There are a few things you need to add to the project, before moving forward. They don't have to do with Navigation, but are required for the lab.

1. Add a few strings to your *strings.xml* file. You will be using these throughout the lab.

   File (strings.xml)

```
<resources>
   ...
   <string name="title_contact_save">Save Changes</string>

   <string name="title_simulate_notification">Sim Notification</string>
   <string name="notification_favorites_channel_name">Favorites
Updates</string>
   <string name="encouragement_message_title">Stay Connected</string>
   <string name="encouragement_message_description">
       %1$s hasn\'t heard from you in a while.
   </string>

</resources>
```

2. Add a new resource menu named ***contact_details_option_menu***, and replace the contents with the following XML.

File (contact_details_option_menu.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto">

   <item android:id="@+id/simulate_notification"
         android:title="@string/title_simulate_notification"
         app:showAsAction="ifRoom|withText" />

</menu>
```

3. Add the new menu to the contact details screen as an options menu. Open ***ContactDetailFragment*** and make the following changes:

File (ContactDetailFragment.kt)

```
class ContactDetailFragment : Fragment() {
   ...
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      if(args.contactId != null) {
         setHasOptionsMenu(true)
      }
   }
   ...
   override fun onViewCreated(...) {
      ...
   }

   override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
       super.onCreateOptionsMenu(menu, inflater)
       inflater.inflate(R.menu.contact_details_option_menu, menu)
   }

   override fun onOptionsItemSelected(item: MenuItem): Boolean {
       return item.onNavDestinationSelected(findNavController()) ||
               when(item.itemId) {
                   R.id.simulate_notification -> {
                       simulateNotification()
                       true
                   }
                   else ->  super.onOptionsItemSelected(item)
               }
   }

   override fun onStop() {
      ...
   }
   fun simulateNotification() {
      Toast.makeText(context, "Simulate Deep Link", Toast.LENGTH_SHORT).show()
   }
   ...
}
```

4. Soon you'll be creating Notifications. Notifications require a NotificationChannel (for apps running on Android O and up). Let's go ahead and add that now, so you won't have to later. Open ***ContactsApplication*** and add the snippets below to create a NotificationChannel when the application starts. :

   File (ContactsApplication.kt)

```
package com.acme.contacts

import android.app.Application
import android.app.NotificationChannel
import android.app.NotificationManager
import android.os.Build
import com.acme.contacts.database.ContactsRepository
…
const val FAVORITES_NOTIFICATION_CHANNEL_ID = "favorites_count"

class ContactsApplication : Application() {

   override fun onCreate() {
       super.onCreate()
       ContactsRepository.initialize(this)

       if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
           val name = getString(R.string.notification_favorites_channel_name)
           val importance = NotificationManager.IMPORTANCE_DEFAULT
           val channel =
               NotificationChannel(
                  FAVORITES_NOTIFICATION_CHANNEL_ID, name, importance)
           val notificationManager: NotificationManager =
               getSystemService(NotificationManager::class.java)
           notificationManager.createNotificationChannel(channel)
       }

   }

}
```

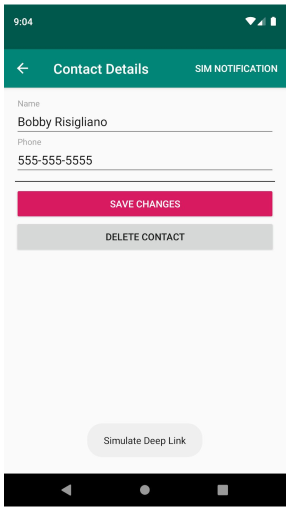
   In the snippet above, you check to see what version of the Android OS the app is running on. If it is on Android O (Api 28) or higher, you create a NotificationChannel.

The channel id is defined as a public constant at the top of the file so that you can reference it later on in your ContactDetailFragment.

Checkpoint

Run the app. Add a contact if one doesn't already exist, then navigate to the contact details screen to edit that contact. You should see an options menu with one option, ***SIM NOTIFICATION***. Click on it and you should see a toast message.

*(**Note:** that this menu only shows up when editing a contact.)*



In a moment, you'll replace this toast message with logic to send a Notification to the user. But first, you'll need to define what action to take when the user clicks the Notification.

This action is wrapped inside of a framework class called a PendingIntent. A PendingIntent is a wrapper for an Intent. The PendingIntent can be handed to other applications so that they can perform the action you described on your behalf at a later time.

Relating this back to the Contacts app. You will need to create an Intent that when triggered will:

- Launch the contacts app (specifically start the MainActivity).
- Navigate directly to the *contact_details* destination with the *contactId* of the person to view/edit passed as an argument.
- Synthesize the backstack (fancy phrase that just means programmatically create a back stack.). The Principles of Navigation stipulates that when the user enters your app via a deep link, the back stack should be similar to that which the user could have organically navigated.

- This Intent needs to be wrapped in a PendingIntent is given to the NotificationManager when your notification is sent. This is so that the Intent can later be triggered to run on your apps behalf.

If this feels like a lot of work, it is!! *...or at least it could be.* Thankfully there is a Navigation class that makes this easy. It's called the ***NavDeepLinkBuilder***.

Using NavDeepLinkBuilder

The NavDeepLinkBuilder can create a PendingIntent to do everything you need to do in just a few lines of code.

Example

```
val pendingIntent = NavDeepLinkBuilder(requireContext())
    .setGraph(R.navigation.contacts_graph)
    .setDestination(R.id.contact_detail)
    .setArguments(arguments)
    .createPendingIntent()
```

You create an instance of the builder passing a Context, set the navigation graph, the target destination, any arguments needed by that destination as an android.os.Bundle, and then you invoke *createPendingIntent()*.

If you are already working with a NavController, you can simplify this even more:

Example

```
val pendingIntent = findNavController().createDeepLink()
    .setDestination(R.id.contact_detail)
    .setArguments(arguments)
    .createPendingIntent()
```

NavController has a function, *createDeepLink()*, that creates a NavDeepLink builder with the context and graph already set. Then you can build onto it setting your target destination, arguments, etc.

You will use this in *ContactDetailFragment* to create a PendingIntent and then send a Notification with it.

Creating a Notification

Inside ***ContactDetailFragment*** create a PendingIntent using a *NavDeepLinkBuilder*, create a Notification configured to use that PendingIntent as the ***contentIntent*** of the Notification, and send it with ***NotificationManagerCompat***. Finally, remove the Toast message.

File (ContactDetailFragment.kt)

```
class ContactDetailFragment : Fragment() {
   ...
  fun simulateNotification() {
      ~~Toast.makeText(context, "Simulate Deep Link", Toast.LENGTH_SHORT).show()~~
      val pendingIntent = findNavController().createDeepLink()
          .setDestination(R.id.contact_detail)
          .setArguments(arguments)
          .createPendingIntent()

      val notification = NotificationCompat
          .Builder(requireContext(), FAVORITES_NOTIFICATION_CHANNEL_ID)
          .setContentTitle(getString(R.string.encouragement_message_title))
          .setSmallIcon(R.drawable.ic_contact_favorite_selected)
          .setContentText(getString(R.string.encouragement_message_description,
                          contactDetailVm.contact.value?.name ?: ""))
          .setContentIntent(pendingIntent)
          .setAutoCancel(true)
          .build()

      val notificationManager = NotificationManagerCompat.from(requireContext())
      notificationManager.notify(0, notification)
  }
  ...
}
```

After making these changes, run the app again, and you should see a Notification when you click on the ***SIM NOTIFICATION*** menu item.

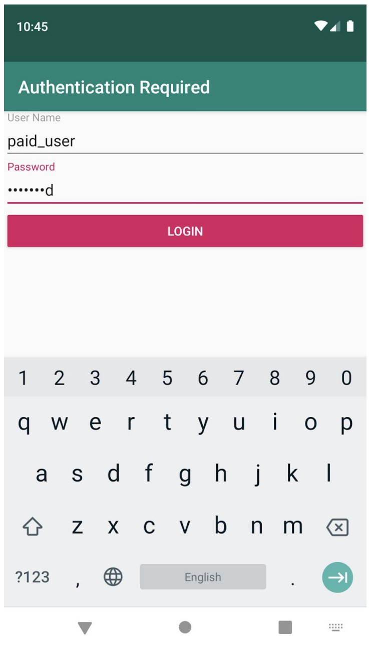
Try backing completely out of the app (pressing the back button repeatedly until you are back on the home screen). Then, pull down the notification bar and click on the notification.

What happens when you press back after following the notification?
(Are you taken back to the home screen or to the Contact List screen?)

If you want to practice Implicit Deep Linking, try the Implicit Deep Linking challenge at the end of this lab.

Conditional Navigation

You've put in a lot of work creating this app and it's really starting to come together. You shouldn't let this work be wasted by giving away your app for free. In the second half of this lab, we're going to update the Contacts app to require a subscription. We'll worry about the subscription and sign up. You just worry about making sure each destination in the app is secured. None of the destinations in the app (*settings, contacts_list, contact_detail*) should be accessible unless the user is authenticated. After we're finished, the app will look something like this it is first launched.



Setup

There are a few things you need to add to the project, before moving forward. They don't have to do with Navigation, but are required for the remainder of the lab.

1. Add a few strings to your *strings.xml* file.

   File (strings.xml)

```
<resources>
   <string name="encouragement_message_description">
       %1$s hasn\'t heard from you in a while.
   </string>

   <string name="username">Username</string>
   <string name="login">Login</string>
   <string name="password">Password</string>
   <string name="login_button_label">Login</string>
   <string name="password_screen_label">Password</string>
   <string name="user_name_label">User Name</string>
   <string name="logout">Logout</string>

</resources>
```

2. Create a new layout resource file called ***fragment_enter_credentials***. This will be the UI for the login screen. Replace the contents with the following:

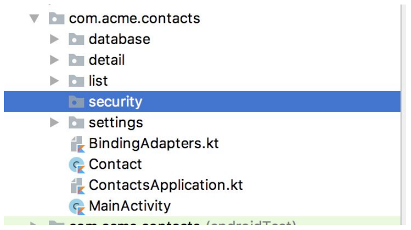
   File (fragment_enter_credentials.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">
   <data />
   <LinearLayout
           android:orientation="vertical"
           android:layout_width="match_parent"
           android:layout_height="wrap_content">

       <com.google.android.material.textfield.TextInputLayout
               android:layout_width="match_parent"
               android:layout_height="match_parent">

           <com.google.android.material.textfield.TextInputEditText
                   android:id="@+id/username_tv"
                   android:layout_width="match_parent"
                   android:layout_height="wrap_content"
                   android:hint="@string/user_name_label"/>
       </com.google.android.material.textfield.TextInputLayout>

       <com.google.android.material.textfield.TextInputLayout
               android:layout_width="match_parent"
               android:layout_height="wrap_content">

           <com.google.android.material.textfield.TextInputEditText
                   android:id="@+id/password_tv"
                   android:layout_width="match_parent"
                   android:layout_height="wrap_content"
                   android:hint="@string/password_screen_label"
                   android:inputType="textPassword"/>
       </com.google.android.material.textfield.TextInputLayout>

       <Button android:id="@+id/login_btn"
               style="@style/Widget.AppCompat.Button.Colored"
               android:text="@string/login_button_label"
               android:layout_width="match_parent"
               android:layout_height="wrap_content" />
   </LinearLayout>
</layout>
```

Securing Destinations

In order to manage the authentication status of a user at any given time you're going to create a ViewModel (Architecture Component) that is scoped to the Activity level.

First, create a new package called ***security*** as a peer to *list*, *details* and *settings*.



Inside of that package, create a new Kotlin Class/File called **AuthenticationViewModel** as a subclass of ViewModel.

Inside AuthenticationViewModel create an enum, ***AuthenticationStatus***, that encapsulates the possible states of authentication.

- **Unauthenticated** - The user is not authenticated
- **Authenticated** - The user is authenticated
- **User Declined** - The user has declined to authenticate, perhaps by pressing the back button while on the login screen.

File (AuthenticationViewModel.kt)

```
class AuthenticationViewModel : ViewModel() {

   enum class AuthenticationStatus {
       UNAUTHENTICATED, // user needs to authenticate state
       AUTHENTICATED, // user is in authenticated state
       USER_DECLINED, // user has declined to authenticate
   }

}
```

The state of authentication at any given moment can be represented by a state variable of type *MutableLiveData*. Create a final variable ***authenticationStatus*** to do this and set the initial value to ***UNAUTHENTICATED***.

File (fragment_enter_credentials.xml)

```
class AuthenticationViewModel : ViewModel() {

   enum class AuthenticationStatus {
       UNAUTHENTICATED, // user needs to authenticate state
       AUTHENTICATED, // user is in authenticated state
       USER_DECLINED, // user has declined to authenticate
   }

   val authenticationStatus =
MutableLiveData<AuthenticationStatus>(AuthenticationStatus.UNAUTHENTICATED)
}
```

Finally, create public functions to support logging in, logging out, and declining authentication. *For the purposes of this example, just update the status. Don't bother with validating the username, password, etc.*

File (fragment_enter_credentials.xml)

```
class AuthenticationViewModel : ViewModel() {
   ...
   val authenticationStatus = MutableLiveData<AuthenticationStatus>(
                                 AuthenticationStatus.UNAUTHENTICATED)

   fun authenticate(username: String, password: String) {
       // disregard credentials, just set auth status to true
       authenticationStatus.value = AuthenticationStatus.AUTHENTICATED
   }

   fun logout() {
       authenticationStatus.value = AuthenticationStatus.UNAUTHENTICATED
   }

   fun declineAuthentication() {
       authenticationStatus.value = AuthenticationStatus.USER_DECLINED
   }

}
```

Next, still inside of the *security* package, create a new Kotlin Class/File called ***EnterCredentialsFragment***. Make this a subclass of Fragment. This will be the backing fragment for the *fragment_enter_credentials.xml* layout file you created a moment ago.

File (EnterCredentialsFragment.kt)

```
class EnterCredentialsFragment : Fragment() {
}
```

Inside EnterCredentialsFragment, implement *onCreateView(...)* and *onViewCreated(...)*. In *onCreateView*, inflate the view using the *DataBindingUtil.inflate* method and keep a reference to the binding. In *onViewCreated(...)*, add a clickListener to the loginBtn.

File (EnterCredentialsFragment.kt)

```
class EnterCredentialsFragment : Fragment() {

   lateinit var binding: FragmentEnterCredentialsBinding

   override fun onCreateView(
       inflater: LayoutInflater,
       container: ViewGroup?,
       savedInstanceState: Bundle?
   ): View? {
       binding = DataBindingUtil.inflate(
                   inflater, R.layout.fragment_enter_credentials, container, false)
       return binding.root
   }

   override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
       super.onViewCreated(view, savedInstanceState)
       binding.apply {
           loginBtn.setOnClickListener {

           }
       }
   }
}
```

Note: *You may have to build your project for Data Binding to generate the required bindings in order to make this code compile.*

Finally, add a reference to the *AuthenticationViewModel* that you created a moment ago, scoped to the activity hosting activity. When the user clicks the login button, invoke *authenticate* on the viewModel, passing the values of the *username* and *password* inputs.

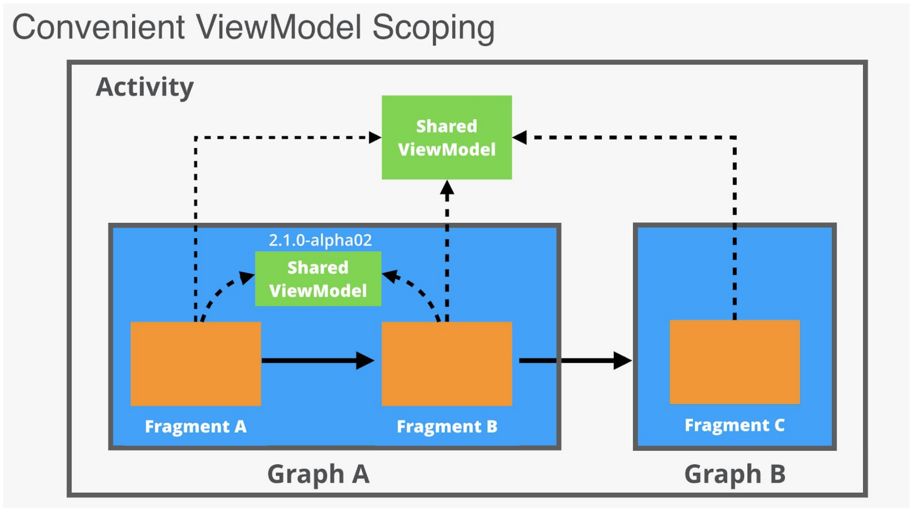
File (EnterCredentialsFragment.kt)

```
class EnterCredentialsFragment : Fragment() {

   val authenticationViewModel by activityViewModels<AuthenticationViewModel>()
   lateinit var binding: FragmentEnterCredentialsBinding
   ...
   override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
       binding.apply {
           loginBtn.setOnClickListener {
                authenticationViewModel.authenticate(
                    usernameTv.text.toString(), passwordTv.text.toString())
            }
        }
    }
}
```

ViewModel Scoping

While it's not related to Navigation, it bears explaining the ***activityViewModels*** property delegate. This is similar to (*by viewModels<>()*) except that the same view model will be retained and provided across any fragment hosted within the same activity, (*MainActivity* in this case). You can also scope ViewModels to a specific Navigation Graph, which is a convenient in between scope, but you will not be using it here in this lab.

This graphic illustrates the difference between Activity and Navigation Graph scoping.



Adding a Login Graph and Global Action

You're almost finished with the Login Screen. The last piece is to add it as a navigable destination in the navigation graph. The Login Screen is part of a separate flow. It's part of a [First Time User Experience](). Currently that flow consists of a single login destination, but imagine that this could eventually grow into a series of screens. Perhaps you'd want to introduce a signup process as part of this flow. You may want to use ViewModels scoped specifically within that flow. Because of this, you're going to create a separate graph to encapsulate this flow.

Create a new *navigation* resource file called ***login_graph.xml*** with the following configuration.

File (login_graph.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<navigation 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/login_graph"
   app:startDestination="@id/enter_credentials">

   <fragment
       android:id="@+id/enter_credentials"
       android:name="com.acme.contacts.security.EnterCredentialsFragment"
       android:label="Authentication Required"
       tools:layout="@layout/fragment_enter_credentials">
      <action
          android:id="@+id/action_login_pop"
          app:popUpTo="@id/login_graph"
          app:popUpToInclusive="true" />
</fragment>

</navigation>
```

Here you added a single destination with the id, ***enter_credentials***, and set this as the graphs starting destination. You also added an id to the graph itself, ***login_graph***. This is important. You will see why in a second.

Inside the *enter_credentials* destination you defined a single action, ***action_login_pop***. This action is configured a bit differently than you've seen before. When this action is followed, instead of pushing a new destination on the navigation stack, it will pop destinations off the stack. Following this action from code using `NavController.navigate(...)` has the same effect as calling `NavController.popBackStack(destinationId: Int, inclusive: Boolean)`. Declaring it here in the XML configuration though makes it arguably more explicit of what you're trying to do and easier to change in the future.

In order to reach this destination from any destination in your *contacts_graph*, you will need to add a reference to *login_graph* and a global action to send users there.

Open ***contacts_graph.xml*** and add these items.

File (contacts_graph.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<navigation 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/contacts_graph"
           app:startDestination="@id/contacts_list">
```

```
    <include android:id="@+id/login_graph" app:graph="@navigation/login_graph" />

    <action
            android:id="@+id/action_global_login"
            app:destination="@id/login_graph"
            app:launchSingleTop="true" />
     ...
     <fragment android:id="@+id/contacts_list" ...>
     ...
     <fragment android:id="@+id/contact_detail" ...>
     ...
     <fragment android:id="@+id/settings" ...>

</navigation>
```

The ***include*** element pulls in the *login_graph* for reference in *contacts_graph*. It is given an id to reference here, ***login_graph***. This id does not have to reference the id given to the graph inside *login_graph.xml*. In fact, it's best that this outer graph has no knowledge of the inner workings of the *login_graph* and vice versa. This degree of separation, keeps the graphs decoupled and easier to change with minimal effects.

You added a global action, that is an action placed as a direct child to the root ***navigation*** element. This will make the action accessible to any destination in the graph.

Now that you can get into the login screen. Finish up ***EnterCredentialsFragment***, by providing a way to leave the login screen, either when the user is authenticated or they press the back button.

File (EnterCredentialsFragment.kt)

```
...
import com.acme.contacts.security.AuthenticationViewModel.AuthenticationStatus.*
import com.acme.contacts.security.EnterCredentialsFragmentDirections.Companion.actionLoginPop

class EnterCredentialsFragment : Fragment() {

    val authenticationViewModel by activityViewModels<AuthenticationViewModel>()
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        binding.apply {
            loginBtn.setOnClickListener {
                authenticationViewModel.authenticate(
                     usernameTv.text.toString(), passwordTv.text.toString())
            }
        }
        authenticationViewModel.authenticationStatus.observe(viewLifecycleOwner,
                Observer { authState ->
                    when (authState) {
                        AUTHENTICATED -> findNavController().navigate(actionLoginPop())
                        USER_DECLINED -> findNavController().navigate(actionLoginPop())
                        else -> Unit
                    }
                }
        )

        requireActivity().onBackPressedDispatcher.addCallback(viewLifecycleOwner) {
            authenticationViewModel.declineAuthentication()
        }
    }
}
```

There are 2 possible paths here. Either the user becomes authenticated (AUTHENTICATED) or they decline authentication (USER_DECLINED).

When the user clicks the login button and the click listener calls *authenticate* on the viewModel it will either result in updating the authenticationStatus to AUTHENTICATED or stay the same. You observe the authentication status LiveData provided by the viewModel so that as soon as the user is Authenticated, you follow the ***action_login_pop*** action, that pops the user to the root nav graph element of this graph and inclusively pops that destination as well. This will put the user back to where they were and they'll be authenticated.

If instead of authenticating, the user presses the back button, you need to tell the viewModel that the user declined. You do that by attaching a lifecycle aware callback, ***`onBackPressedCallback`***. This overrides the default back button behavior. Instead of taking the user back, you call *declineAuthentication()* to tell the viewModel about it. The viewModel changes the authentication state to USER_DECLINED and when it changes, the listener you setup reacts to it in the same way.

Even though the navigation results in the same path, the outcomes of AUTHENTICATED vs USER_DECLINED are different. Any secured fragment monitoring this authentication state will react to these states differently.

Monitoring Authentication State

Now that you have a login screen and a viewModel to track the authentication status, you are going to create an abstract base fragment, ***SecureFragment***, that subscribes to *AuthenticationViewModel.AuthenticationStatus* and conditionally navigates the user appropriately based on the state.

- If at any time the user becomes unauthenticated, *SecureFragment* will send the user directly to the login fragment.

- If the user has declined authentication, *SecureFragment*, will pop back (or finish the activity if already at the start destination).

- If the user is authenticated, it will do nothing.

This fragment will serve as a base fragment for any destination that we wish to secure in the app… *such as... all of them!*

Inside of the *security* package, create a new abstract Kotlin Class/File named, ***SecureFragment***., that extends Fragment. Inside add an activity scoped reference to the *AuthenticationViewModel*.

File (SecureFragment.kt)

```
abstract class SecureFragment : Fragment() {

   val authenticationViewModel by activityViewModels<AuthenticationViewModel>()

}
```

Override *onViewCreated*(). Inside, subscribe to *authenticationViewModel.authenticationStatus,* reacting to status changes in the users authentication state as appropriate.

File (SecureFragment.kt)

...

```
import com.acme.contacts.ContactsGraphDirections.Companion.actionGlobalLogin
import com.acme.contacts.security.AuthenticationViewModel.AuthenticationStatus.*

abstract class SecureFragment : Fragment() {

   val authenticationViewModel by activityViewModels<AuthenticationViewModel>()

   override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
       super.onViewCreated(view, savedInstanceState)

       authenticationViewModel.authenticationStatus.observe(viewLifecycleOwner,
              Observer { authStatus ->
                     when(authStatus) {
                            UNAUTHENTICATED -> findNavController()
                                   .navigate(actionGlobalLogin())
                            USER_DECLINED -> popBackStackOrExit()
                            else -> Unit
                     }
              }
       )
   }
```

```
    private fun popBackStackOrExit() {
        if(!findNavController().popBackStack()) {
            requireActivity().finish()
        }
    }
}
```

Finally, secure each destination by changing the base class for each to ***SecureFragment***.

File (ContactDetailFragment.kt)

```
class ContactDetailFragment : SecureFragment() { ... }
```

File (ContactsListFragment.kt)

```
class ContactsListFragment : SecureFragment() { ... }
```

File (SettingsFragment.kt)

```
class SettingsFragment : SecureFragment() { ... }
```

And, update MainActivity to hide the BottomNavigationView and Up when the user is on the login destination.

File (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {
  ...
  override fun onCreate(savedInstanceState: Bundle?) {
    ...
    appBarConfiguration = AppBarConfiguration(setOf(R.id.contacts_list, R.id.settings, R.id.enter_credentials))
    setupActionBarWithNavController(navController, appBarConfiguration)

    navController.addOnDestinationChangedListener { navController, newDestination, arguments ->
      when (newDestination.id) {
        R.id.contact_detail -> bottomNavigationView.visibility = View.GONE
        R.id.enter_credentials -> bottomNavigationView.visibility = View.GONE
        else -> bottomNavigationView.visibility = View.VISIBLE
      }
    }
  }
  ...
}
```

Checkpoint

Run the app. You should now be prompted with a login screen when the app launches. If you click back, the app should exit. If you instead, click the login button, you should be taken to the starting destination.

Try the deep link notification you set up earlier. What happens when you open the app via the notification?

Notice that you are prompted for authentication. This works because each individual destination is secured. Notice that from the contact details screen, if you decline to authenticate, you are backed completely out of the application.

Challenge : Implicit Deep Linking

Recall in lecture the other type of deep linking is implicit deep linking. This involves adding a uri in your navigation graph that links directly to a specific destination and may provide arguments. Implicit deep linking relies on Implicit Intents and marker that you add in your Android manifest file.

There is no solution for this challenge, but if you would like to give it a shot, here are the high level steps you'll need to perform to create an implicit deep link for the contacts app, that takes the user directly to the contact details screen, where they can add a contact.

- Make decide on a URI (example: http://contacts.acme.com/contact_add).
- Add this uri to your *contect_detail* destination so that, when followed the app launches into that destination. Where they can fill out the name and phone number for a new contact they wish to add.
- For an extra challenge, add an additional, optional, *name* argument to the contact_detail destination and change the URI to (http://contacts.acme.com/contact_add/{name}). When the user follows this URI, make it so that the contact name field is pre-populated in the form.