Getting Started

To get started, download the CodeRAD starter project from here.

Extract the .zip file and then open the project in IntelliJ IDEA

NOTE The CodeRAD starter project is a maven project which can be opened in any Maven-compatible IDE (e.g. NetBeans, Eclipse, IntelliJ, etc...). For this tutorial, I'll be using IntelliJ.

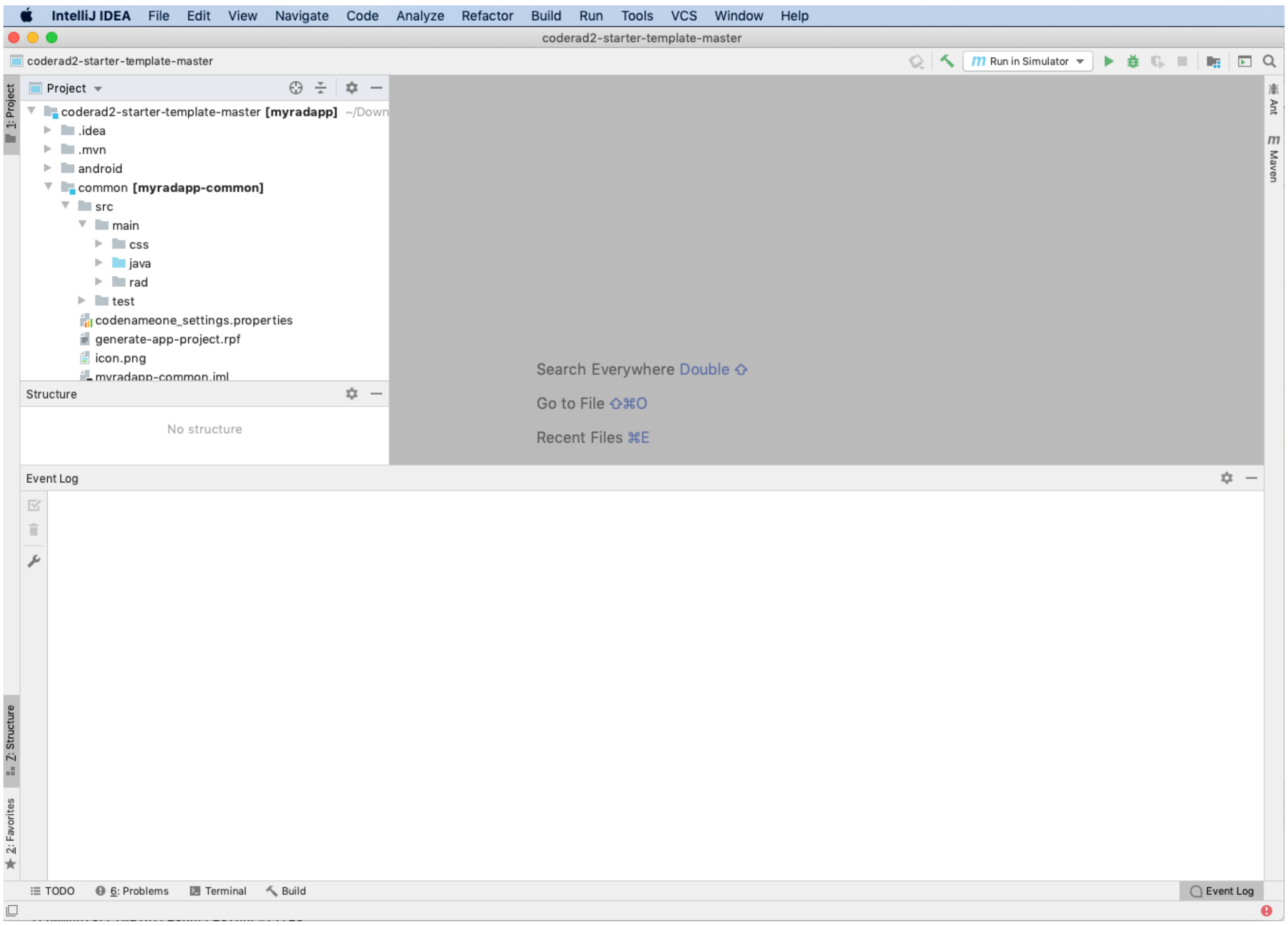


Figure 1. The Code RAD starter project opened in IntelliJ IDEA

Once the project is opened, press the ▶ icon on the toolbar to run the project in the Codename One simulator.

NOTE The first time you run and build the project it will take some time because it needs to download all of the project dependencies. Subsequent builds should only take a few seconds.

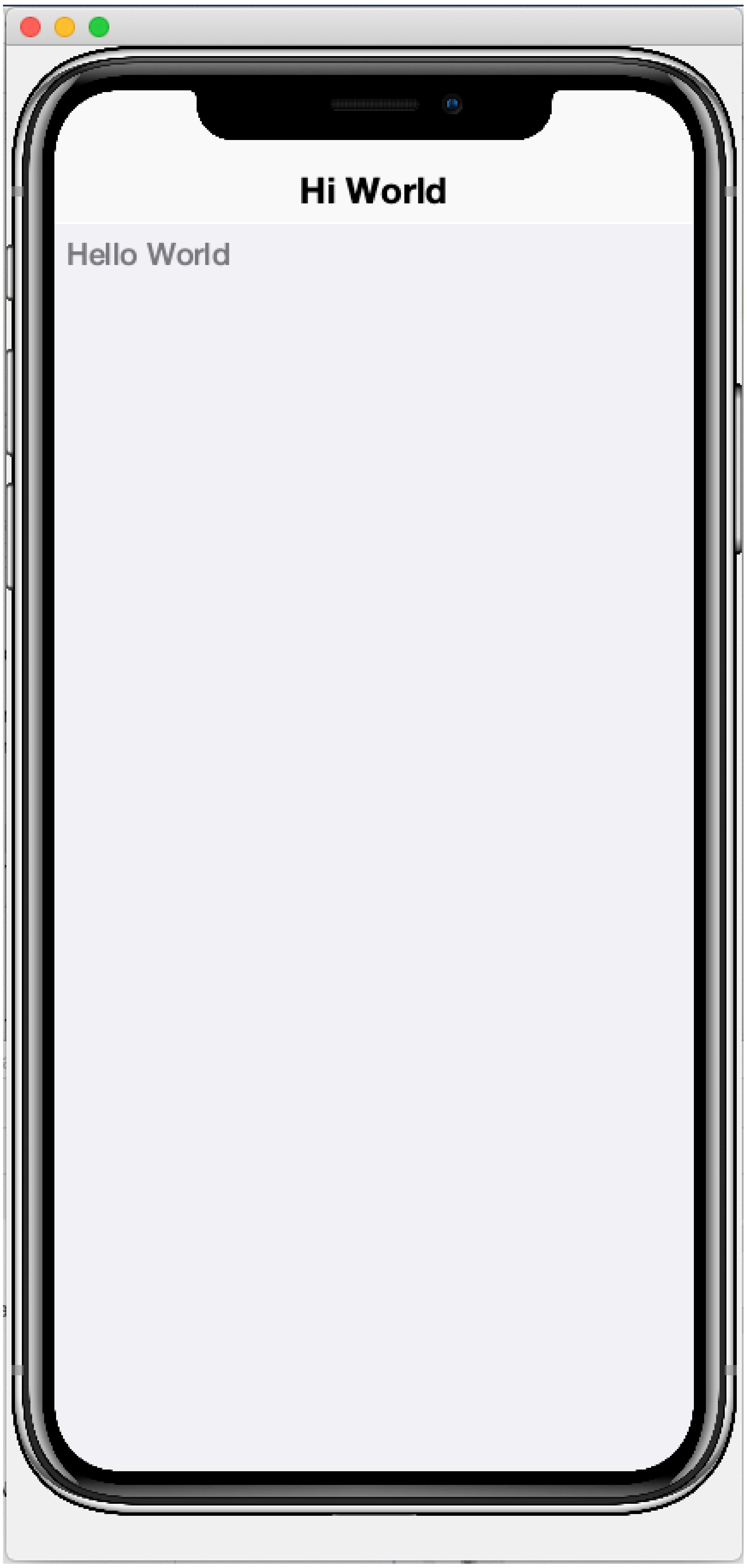
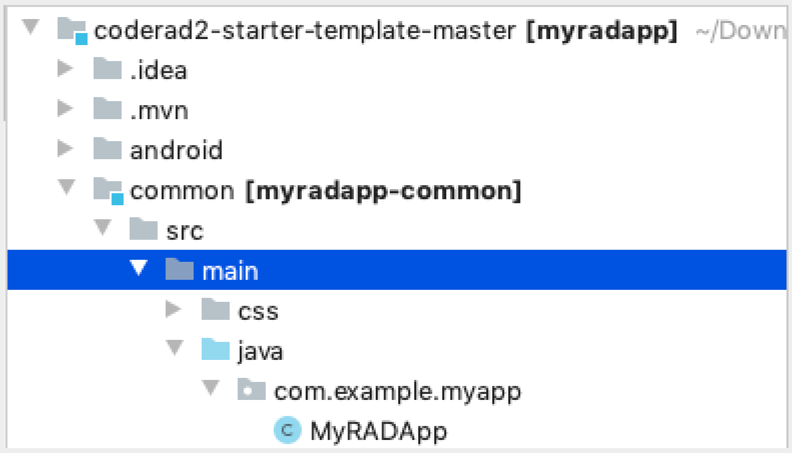


Figure 2. The starter project running inside the Codename One simulator

Under the Hood

Let’s take look under the hood to see how this "Hello World" app is producted.

The *starting point* for your app is the `com.example.myapp.MyRADApp` class, located in the *common/src/main/java* directory.


coderad2-starter-template-master **[myradapp]** ~/Down
.idea
.mvn
android
common **[myradapp-common]**
src
main
css
java
com.example.myapp
MyRADApp

The contents are shown below:

```
package com.example.myapp;

/*.. imports omitted ...*/

public class MyRADApp extends ApplicationController {

    public void actionPerformed(ControllerEvent evt) {
        with(evt, StartEvent.class, startEvent -> {
            startEvent.setShowingForm(true);
            new StartPageController(this).show();
        });
        super.actionPerformed(evt);
    }
}
```

This class overrides ApplicationController, and listens for the *StartEvent*, which is fired when the app starts, or is brought to the foreground.

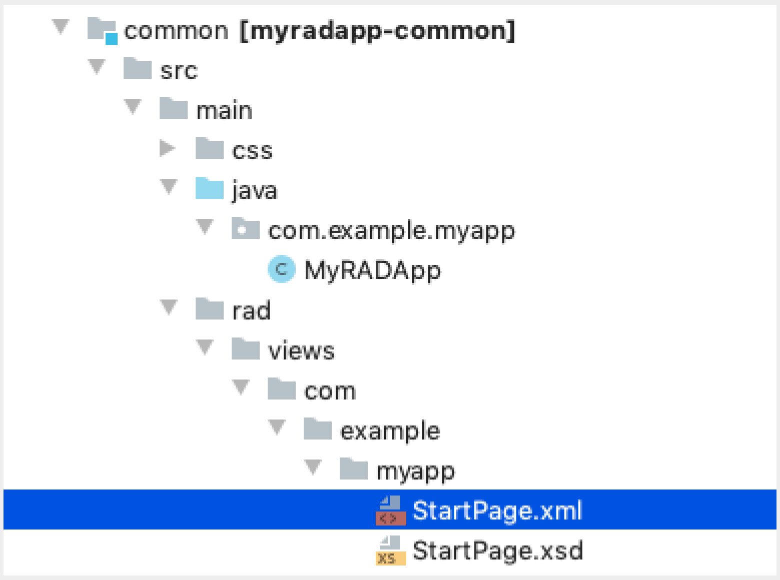
The effective action performed by the event handler is the line:

```
new StartPageController(this).show();
```

This creates a new *StartPageController* and shows it.

The *StartPagePageController* class is a FormController subclass that is generated from the *StartPage* view at compile-time by the CodeRAD annotation processor.

The *StartPage* view is implemented as a RAD View using XML. It is located inside the *common/src/main/rad/views* directory.



```
common [myradapp-common]
  src
    main
      css
      java
        com.example.myapp
          MyRADApp
      rad
        views
          com
            example
              myapp
                StartPage.xml
                StartPage.xsd
```

The contents are shown below:

```
<?xml version="1.0"?>
<y xsi:noNamespaceSchemaLocation="StartPage.xsd" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
    <title>Hi World</title>
    <label>Hello World</label>
</y>
```

TIP The boilerplate attributes `xsi:noNamespaceSchemaLocation="StartPage.xsd"` `xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"` are injected automatically by the annotation processor. You can leave those out when you create your own views.

Let's walk through this file line by line:

`<?xml version="1.0"?>`
: Obligatory XML boilerplate.

`<y>`
: A Container with `BoxLayout.Y` layout. (i.e. a container that lays out its children vertically).

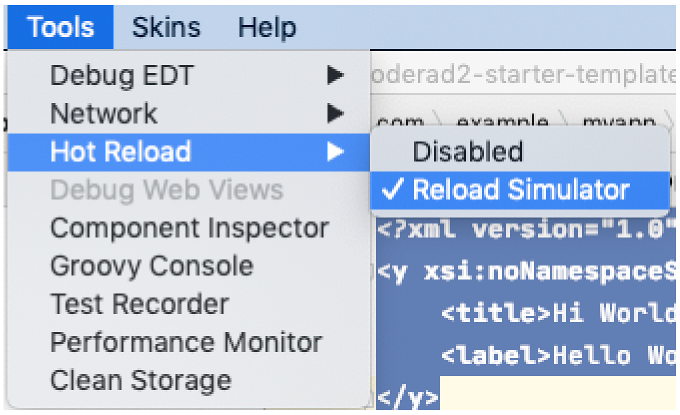
`<title>Hi World</title>`
: Not a child component, rather a "bean" that sets the title of the form to "Hi World"

`<label>Hello World</label>`
: A Label component with the text "Hello World"

Hot Reload

The Codename One simulator supports a "Hot Reload" feature that can dramatically improve productivity. Especially if you're like me, and you like to experiment with the UI by trial and error.

Hot Reload is disabled by default, but you can enable it using the *Tools* > *Hot Reload* > *Reload Simulator* option (From the Simulator's menu bar).



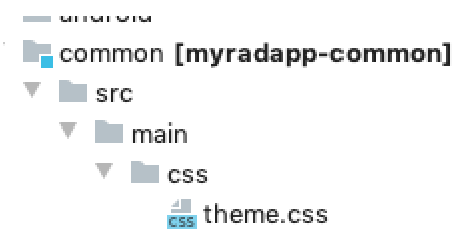
If the *Reload Simulator* option is checked, then the simulator will monitor the project source files for changes, and automatically recompile and reload the simulator as needed.

TIP Technically the *Reload Simulator* option isn't a hot reload, since it actually restarts the simulator - and you will lose your place in the app. True hot reload (where the classes are reloaded transparently without having to restart the simulator) is also available, but it is experimental and requires some additional setup.

The remainder of this tutorial will assume that you have *Hot Reload* enabled

Changing the Styles

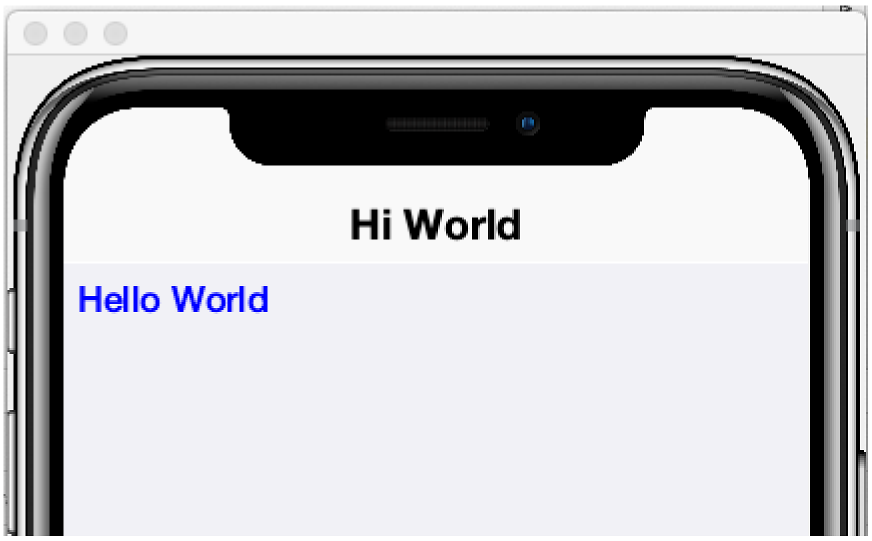
Keep the simulator running, and open the CSS style stylesheet for the project, located at *common/src/main/css/theme.css*.



Add the following snippet to the *theme.css* file:

```
Label {
  color: blue;
}
```

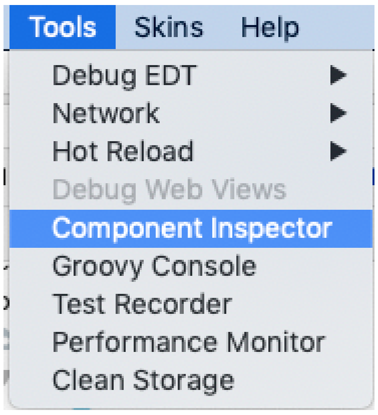
Within a second or two after you save the file, you should notice that the "Hello World" label in the simulator has turned blue.



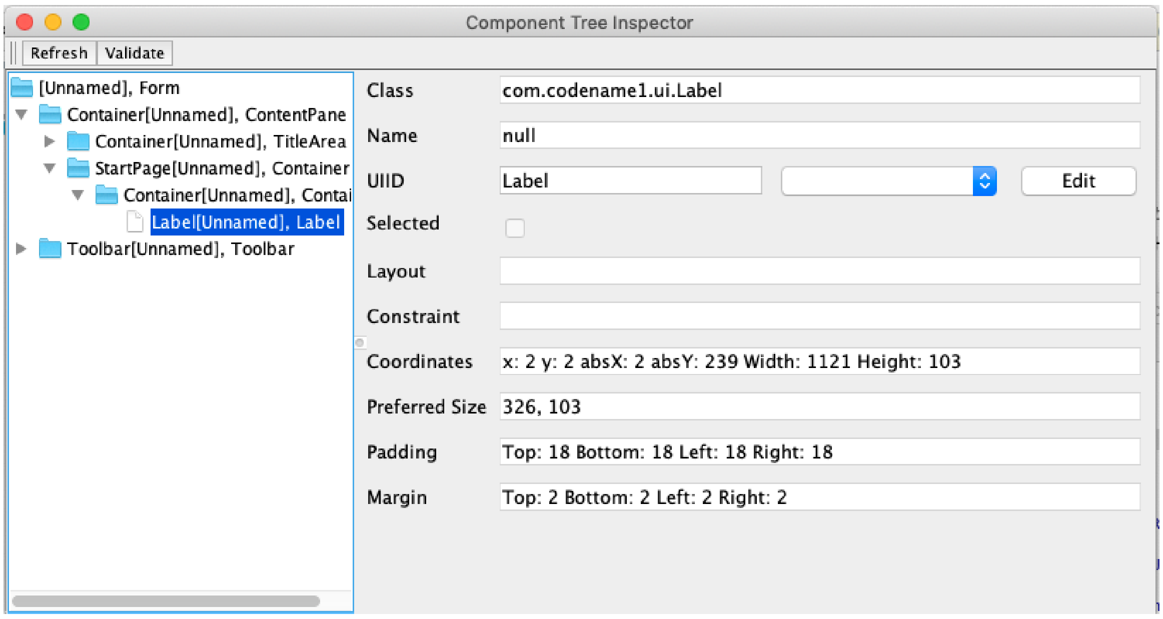
This is because the Label component’s default UIID is "Label", so it adopts styles defined for the selector "Label" in the stylesheet.

TIP

If you are unsure of the UIID of a particular component, you can use the component inspector in the simulator to find out. Select *Tools* > *Component Inspector*



In the *Component Inspector*, you can expand the component tree in the left panel until you reach the component you're looking for. The details of that component will then be shown in the right panel.



- [Unnamed], Form
  - Container[Unnamed], ContentPane
    - Container[Unnamed], TitleArea
    - StartPage[Unnamed], Container
      - Container[Unnamed], Contai
        - Label[Unnamed], Label
  - Toolbar[Unnamed], Toolbar

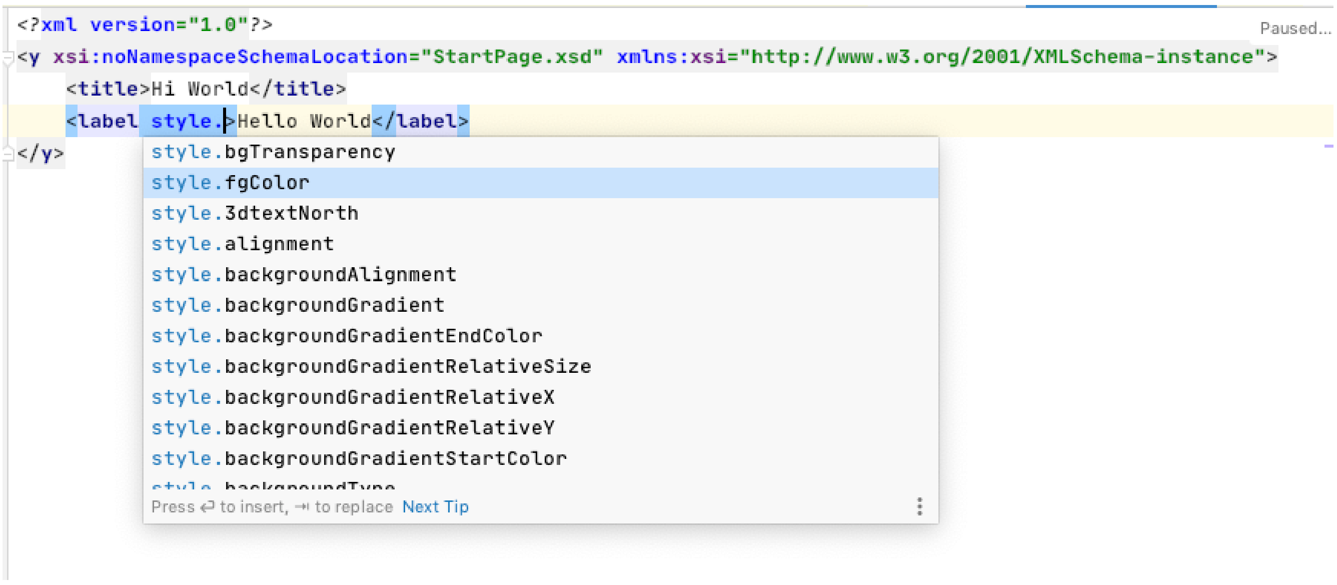
Class	com.codename1.ui.Label
Name	null
UIID	Label
Selected	
Layout	
Constraint	
Coordinates	x: 2 y: 2 absX: 2 absY: 239 Width: 1121 Height: 103
Preferred Size	326, 103
Padding	Top: 18 Bottom: 18 Left: 18 Right: 18
Margin	Top: 2 Bottom: 2 Left: 2 Right: 2

The *UIID* field will show you the UIID of the component that you can use to target the component from the stylesheet.

The above stylesheet change will change the color of *all* labels to *blue*. What if we want to change only the color of *this* label without affecting the other labels in the app? There are two ways to do this. The first way is to override the *fgColor* style inline on the `<label>` tag itself.

Inline Styles

In the *StartPage.xml* file, add the `style.fgColor` attribute to the `<label>` tag with the value "0xff0000".



```
<?xml version="1.0"?>
<y xsi:noNamespaceSchemaLocation="StartPage.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <title>Hi World</title>
    <label style.>Hello World</label>
</y>
```

```
style.bgTransparency
style.fgColor
style.3dtextNorth
style.alignment
style.backgroundAlignment
style.backgroundGradient
style.backgroundGradientEndColor
style.backgroundGradientRelativeSize
style.backgroundGradientRelativeX
style.backgroundGradientRelativeY
style.backgroundGradientStartColor
style.backgroundType
```

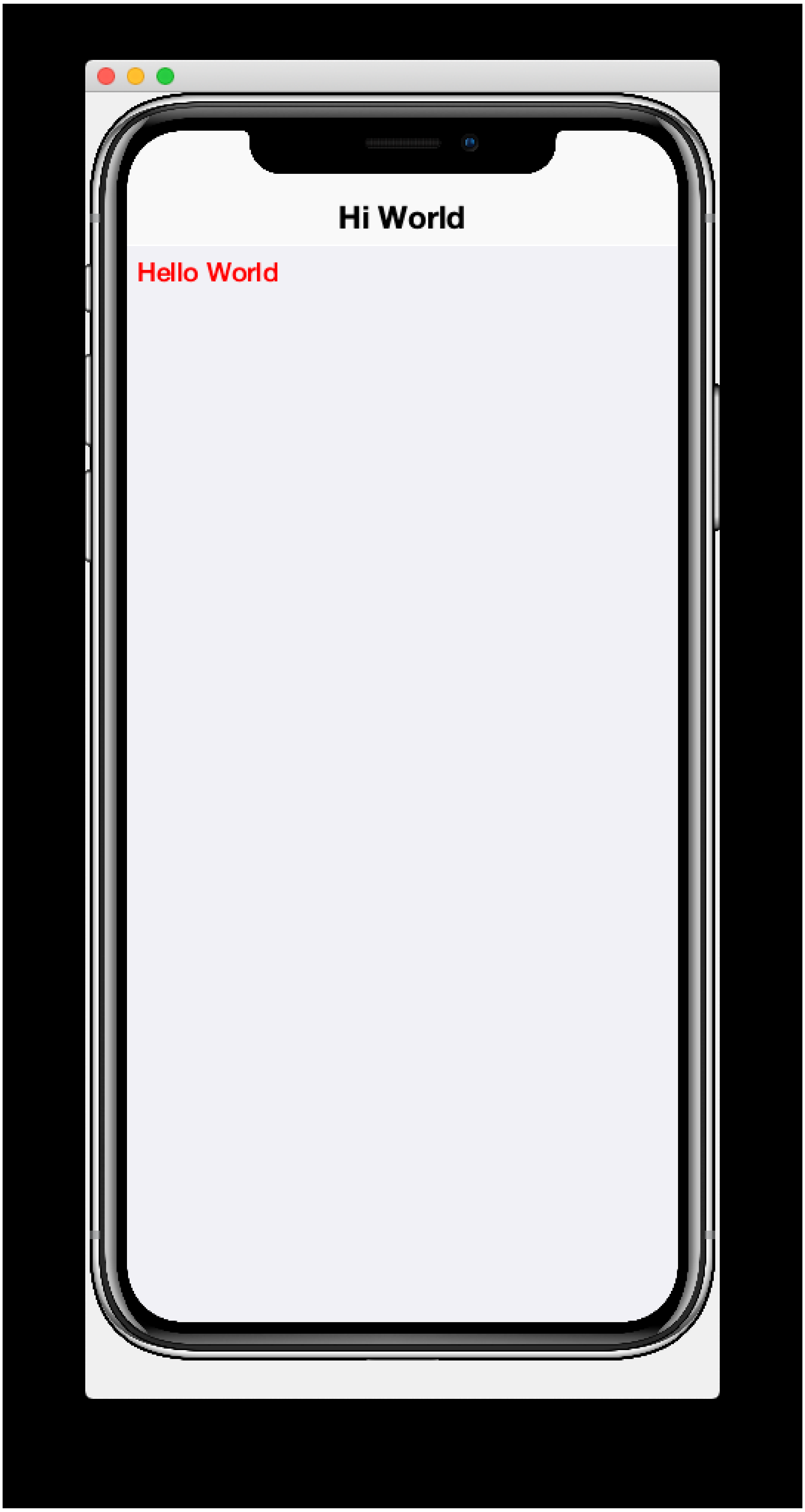
Figure 3. In IntelliJ's XML editor, you'll receive type hints for all tags and attributes as shown here.

Notice that, as soon as you start typing inside the `<label>` tag, the editor presents a drop-down list of options for completion. This is made possible by the schema (StartPage.xsd located in the same directory as your StartPage.xml file) that the CodeRAD annotation processor automatically generates for you. This schema doesn't include *all* of the possible attributes you can use, but it does include most of the common ones.

After making the change, your *StartPage.xml* file should look like:

```
<?xml version="1.0"?>
<y xsi:noNamespaceSchemaLocation="StartPage.xsd" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
    <title>Hi World</title>
    <label style.fgColor="0xff0000">Hello World</label>
</y>
```

And, within a couple of seconds, the simulator should have automatically reloaded your form - this time with "Hello World" in *red* as shown below.



Hi World

Hello World

If it doesn't automatically reload your form, double check that you have *Hot Reload* enabled (See the *Tools* > *Hot Reload* menu). If *Hot Reload* is enabled and it still hasn't updated your form, check the console output for errors. It is likely that your project failed to recompile; probably due to a syntax error in your *StartPage.xml* file.

XML Tag Attributes

In the above example, we added the `style.fgColor` attribute to the `<label>` tag to set its color. This attribute corresponds to the following Java snippet on the label:

```
theLabel.getStyle().setFgColor(0xff0000);
```

In a similar way, you can set any property via attributes that can be accessed via a chain of "getters" from the label, as long as the end of the chain has an appropriate "setter". The *Label* class includes a "setter" method `setPreferredH(int height)`. You could set this via the `preferredH` property e.g.:

```
<label preferredH="100"/>
```

would correspond to the Java:

```
theLabel.setPreferredH(100)
```

In the `style.fgColor` example, the `style` portion corresponded to the `getStyle()` method, and the `fgColor` component corresponded to the `setFgColor()` method of the `Style` class. The `Label` class also has a `getDisabledStyle()` method that returns the style that is to be used when the label is in "Disabled" state. This isn't as relevant for `Label` as it would be for active components like `Button` and `TextField`, but we could set it using attributes. E.g.

```
<label disabledStyle.fgColor="0xff0000">Hello World</label>
```

or All styles (which sets the style for all of the component states at once):

```
<label allStyles.fgColor="0xff0000">Hello World</label>
```

This sidebar is meant to give you an idea of the attributes that are available to you in this XML language, however, we haven't yet discussed the vocabulary that is available to you for the attribute values. So far the examples have been limited to *literal* values (e.g. `0xff0000`), but this is just for simplicity. Attributes values can be any valid Java expression in the context. See the section on "Attribute Values" for a more in-depth discussion on this, as there are a few features and wrinkles to be aware of.

Custom UIIDs

The second (preferred) way to override the style of a particular Label without affecting other labels in the app is to create a custom UIID for the label.

Start by changing the `Label` style in your stylesheet to `CustomLabel` as follows:

```
CustomLabel {
  cn1-derive: Label;  ①
  color: blue;
}
```

① The `cn1-derive` directive indicates that our style should "inherit" all of the styles from the "Label" style.

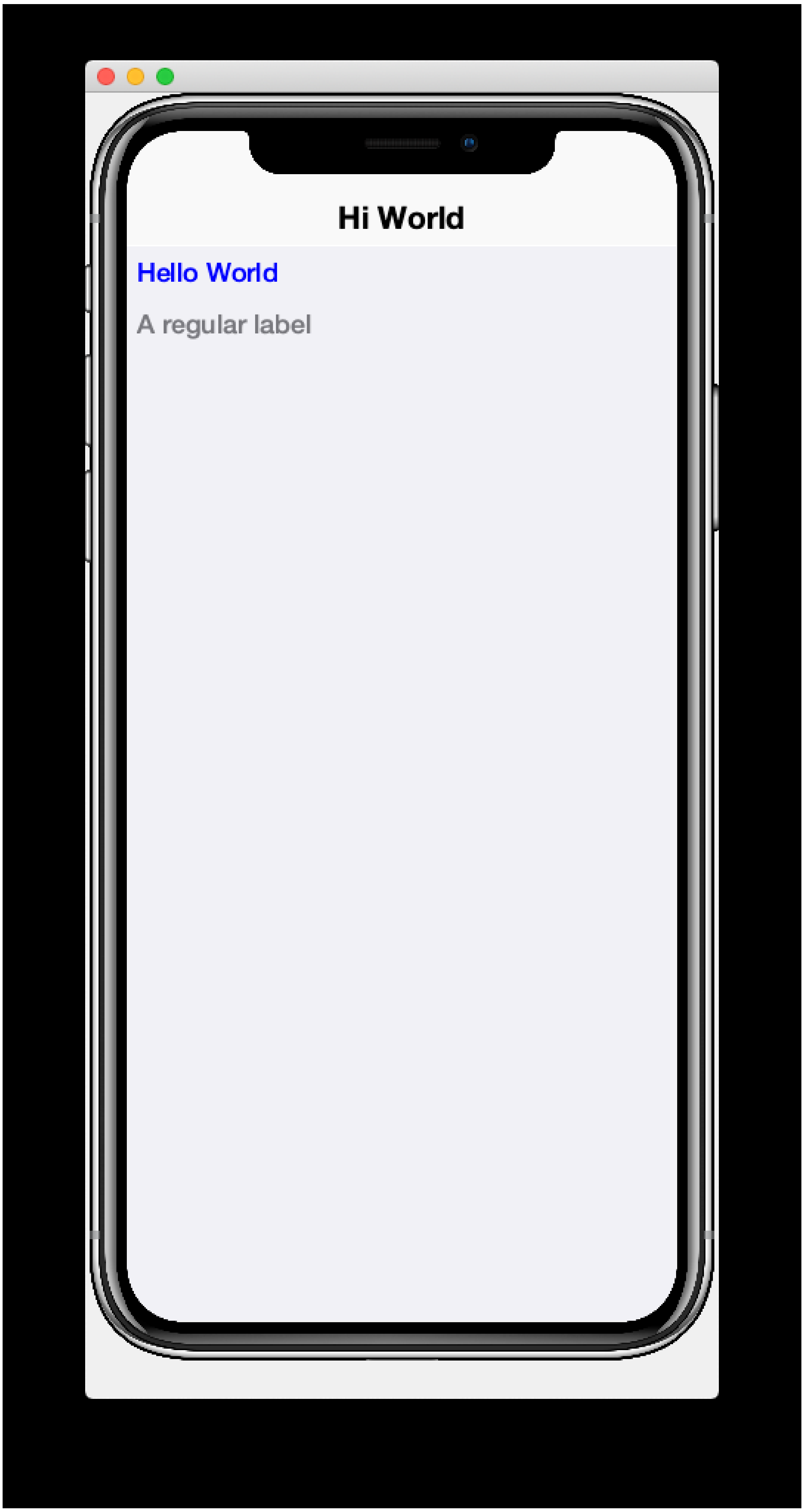
Now return to the *StartPage.xml* file and add `uiid="CustomLabel"` to the `<label>` tag. While we're at it, remove the inline `style.fgColor` attribute:

```
<label uiid="CustomLabel">Hello World</label>
```

Finally, to verify that our style only affects this single label, let's add another label to our form without the `uiid` attribute. When all of these changes are made, the *StartPage.xml* content should look like:

```
<?xml version="1.0"?>
<y xsi:noNamespaceSchemaLocation="StartPage.xsd" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
    <title>Hi World</title>
    <label uiid="CustomLabel">Hello World</label>
    <label>A regular label</label>
</y>
```

After saving both *theme.css* and *StartPage.xml*, the simulator should automatically reload, and you'll see something like the following:


Hi World
Hello World
A regular label

Adding More Components

So far we’ve only used the `<label>` tag, which corresponds to the `Label` component. You are not limited to `<label>`, nor are you limited to any particular subset of "supported" components. You can use *any* Component in your XML files that you could use with Java or Kotlin directly. You can even use your own custom components.

The tag name will be the same as the simple class name of the component you want to use. By convention, the tag names begin with a lowercase letter. E.g. The *TextField* class would correspond to the `<textField>` tag.

XML Tag Namespaces

Since XML tags use only the *simple* name for its corresponding Java class, you may be wondering how we deal with name collisions. For example, what if you have defined your own component class *com.xyz.widgets.TextField*. Then how would you differentiate this class from the *com.codename1.ui.TextField* class in XML. Which one would `<textField>` create?

The mechanism of differentiation here is the same as in Java. Each XML file includes a set of *import* directives which specify the package namespaces that it will search to find components corresponding with an XML tag. It small selection of packages are imported "implicitly", such as *com.codename1.ui*, *com.codename1.components*, *com.codename1.rad.ui.propertyviews*, and a few more. If you want to import *additional* packages or classes, you can use the `<import>` tag, and include regular Java-style import statements as its contents.

E.g.

```
<?xml version="1.0" ?>
<y>
  <import>
  import com.xyz.widgets.TextField;
  </import>

  <!-- This would create an instance of com.xyz.widgets.TextField
       and not com.codename1.ui.TextField -->
  <textField/>
</y>
```

You can include any valid Java import statement inside the `<import>` tag.

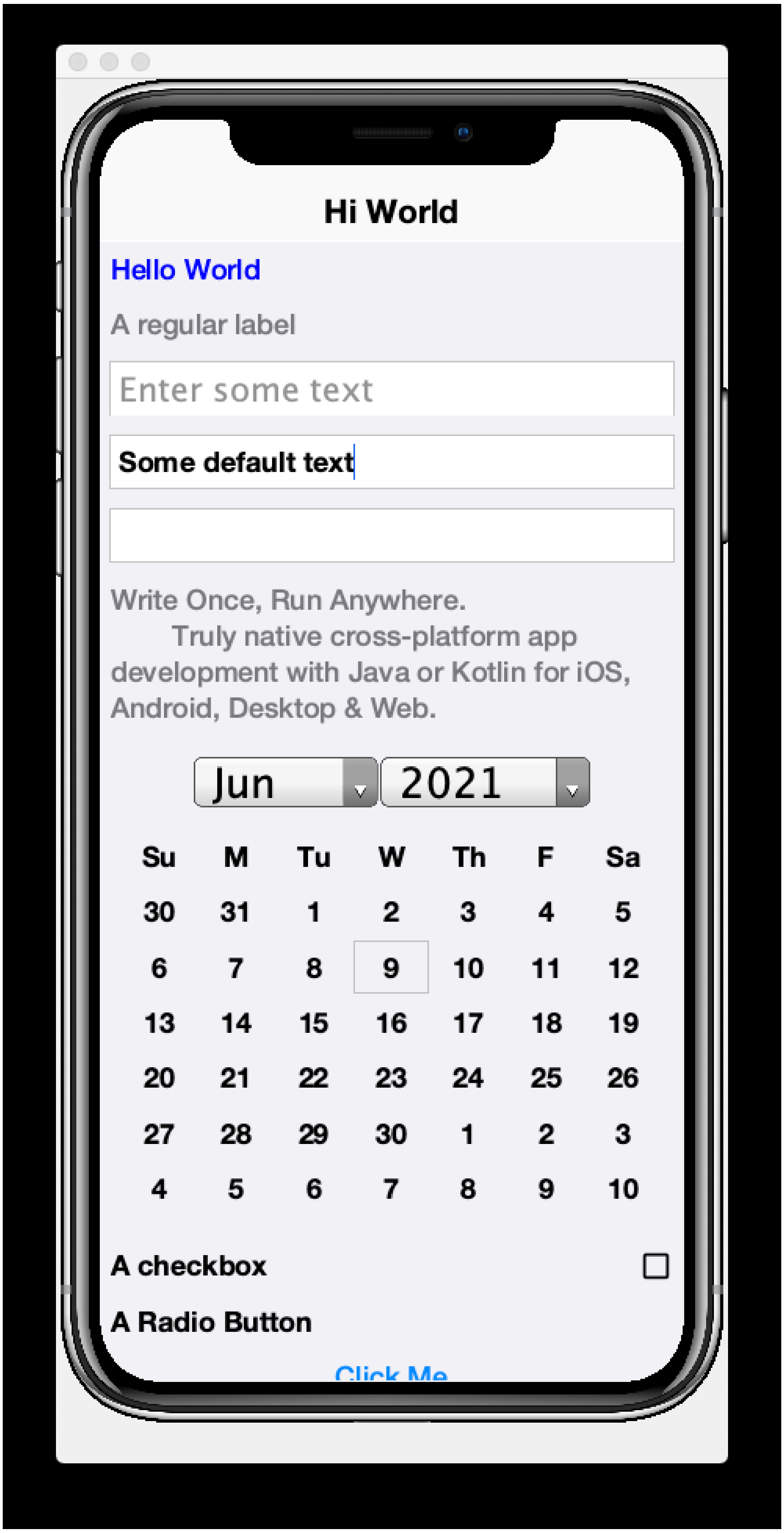
E.g. the following mix of package and class imports is also fine:

```
<import>
import com.xyz.widgets.TextField;
import com.xyz.otherwidgets.*;
</import>
```

For fun, let’s try adding a few of the core Codename One components to our form to spice it up a bit.

```xml
<?xml version="1.0"?>
<y scrollableY="true" xsi:noNamespaceSchemaLocation="StartPage.xsd" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
    <title>Hi World</title>
    <label uiid="CustomLabel">Hello World</label>
    <label>A regular label</label>

    <!-- A text field with a hint -->
    <textField hint="Enter some text"></textField>

    <!-- A text field default text already inserted -->
    <textField>Some default text</textField>

    <!-- A password field -->
    <textArea constraint="TextArea.PASSWORD"/>

    <!-- Multiline text -->
    <spanLabel>Write Once, Run Anywhere.
        Truly native cross-platform app development with Java or Kotlin for iOS,
Android, Desktop &amp; Web.
    </spanLabel>

    <!-- A Calendar -->
    <calendar/>

    <checkBox>A checkbox</checkBox>

    <radioButton>A Radio Button</radioButton>

    <button>Click Me</button>

    <spanButton>Click
    Me</spanButton>

    <multiButton textLine1="Click Me"
        textLine2="A description"
                 textLine3="A subdesc"
                 textLine4="Line 4"
    />



</y>
```

After changing the contents of your *StartPage.xml* file to the above, and saving it, you should see the following result in the simulator:


Hi World

Hello World

A regular label

Enter some text

Some default text

Write Once, Run Anywhere.
        Truly native cross-platform app development with Java or Kotlin for iOS, Android, Desktop & Web.

Su	M	Tu	W	Th	F	Sa
30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	1	2	3
4	5	6	7	8	9	10

Adding Actions

CodeRAD is built around the Model-View-Controller (MVC) philosophy which asserts that the *View* logic (i.e. how the app looks) should be separated from the *Controller* logic (i.e. what the app does with user input). *Actions* form the cornerstone of how CodeRAD keeps these concerns separate. They provide a sort of communication channel between the controller and the view, kind of like a set of Walkie-talkies.

To go with the Walkie-talkie metaphor for a bit, A View will broadcast on a few frequencies that are predefined by the View. It might broadcast on 96.9MHz when the "Help" button is pressed, and 92.3MHz when text is entered into its *username* text field.

Before displaying a View, the Controller will prepare a set of one-way Walkie-talkies at a particular frequency. It passes one of the handset's to the view - the one that *sends*. It retains the other handset for itself - the one that receives.

When the view is instantiated, it will look through all of the walkie-talkie handsets that were provided and see if any are set to a frequency that it wants to broadcast on. If it finds a match, it will use it to broadcast relevant events. To continue with the example, if finds a handset that is tuned to 96.9MHz, it will send a message to this handset whenever the "Help" button is pressed.

When the controller receives the message in the corresponding hand-set of this walkie-talkie, it can respond by performing some action.

The view can also use the set of Walkie-talkies that it receives to affect how it renders itself. For example, if, when it is instantiated, it doesn't find any handsets tuned to 96.9MHz, it may "choose" just to not render the "Help" button at all, since nobody is listening.

Additionally, the Controller might attach some additional instructions to the handset that it provides to the view. The view can then use these instructions to customize how it renders itself, or how to use the handset. For example, the handset might come with a note attached that says "Please use *this* icon if you attach the handset to a button", or "Please use *this* text for the label", or "Please disable the button under this condition".

In the above metaphor, the *frequency* represents an instance of the `ActionNode.Category` class, and the walkie-talkies represent an instance of the `ActionNode` class. The *View* declares which *Categories* it supports, how it will interpret them. The *Controller* defines *Actions* and registers them with the view in the prescribed categories. When the *View* is instantiated, it looks for these actions, and will use them to affect how it renders itself. Typically actions are manifested in the View as a button or menu item, but not necessarily. `EntityListView`, for example, supports the `LIST_REFRESH_ACTION` and `LIST_LOAD_MORE_ACTION` categories which will broadcast events when the list model should be refreshed, or when more entries should be loaded at the end of the list. They don't manifest in any particular button or menu.

Adding our first action

Let's begin by restoring the *StartPage.xml* template to its initial state:

```
<?xml version="1.0"?>
<y scrollableY="true" xsi:noNamespaceSchemaLocation="StartPage.xsd" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
    <title>Hi World</title>
    <label>Hello World</label>
</y>
```

Now, let’s define an action category using the `<define-category>` tag.

```
<?xml version="1.0"?>
<y scrollableY="true" xsi:noNamespaceSchemaLocation="StartPage.xsd" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
    <define-category name="HELLO_CLICKED" />
    <title>Hi World</title>
    <label>Hello World</label>
</y>
```

And then change the `<label>` to a `<button>`, and "bind" the button to the "HELLO_CLICKED" category using the `<bind-action>` tag:

```
<?xml version="1.0"?>
<y scrollableY="true" xsi:noNamespaceSchemaLocation="StartPage.xsd" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
    <define-category name="HELLO_CLICKED" /> ①
    <title>Hi World</title>
    <button>Hello World
        <bind-action category="HELLO_CLICKED"/>
    </button>

</y>
```

① The `define-category` tag will define an `ActionNode.Category` in the resulting Java View class with the given name.

When the simulator reloads after this last change you will notice that the "Hello World" button is not displayed. You do not need to adjust your lenses. This is *expected* behaviour. Since the button is bound to the "HELLO_CLICKED" category, and the controller hasn’t supplied any actions in this category, the button will not be rendered.

Let’s now define an action in the Controller with this category. Open the *com.example.myapp.MyRadApp* class and add the following method:

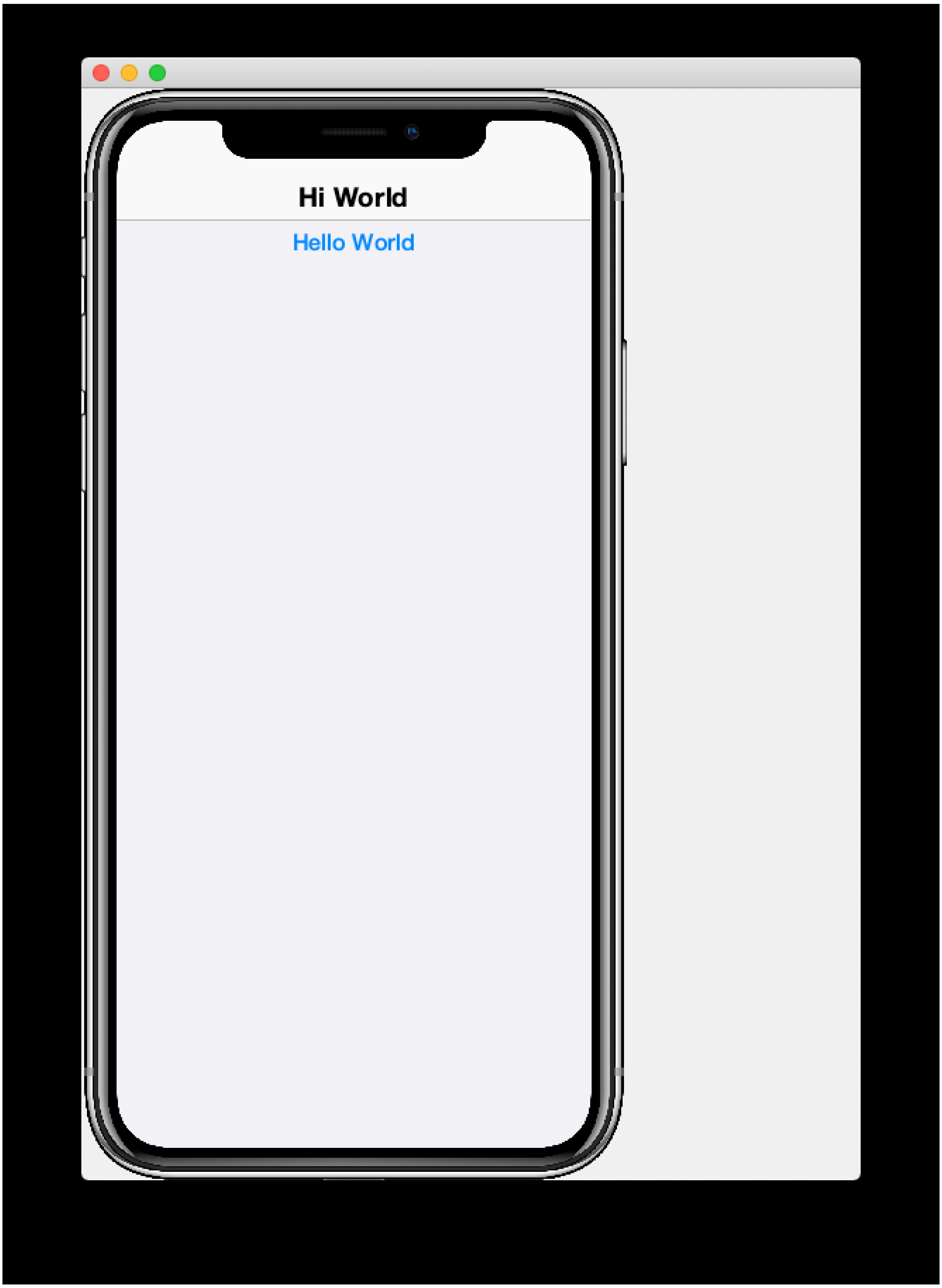
```
@Override
protected void initControllerActions() {
    super.initControllerActions();
    addAction(StartPage.HELLO_CLICKED, evt-> {
        evt.consume();
        Dialog.show("Hello", "You clicked me", "OK", null);
    });
}
```

The `initControllerActions()` method is where all actions should be defined in a controller. This method is guaranteed to be executed before views are instantiated. The `addAction()` method comes in multiple flavours, the simplest of which is demonstrated here. The first parameter takes the `HELLO_CLICKED` action category that we defined in our view, and it registered an `ActionListener` to be called when that action is fired.

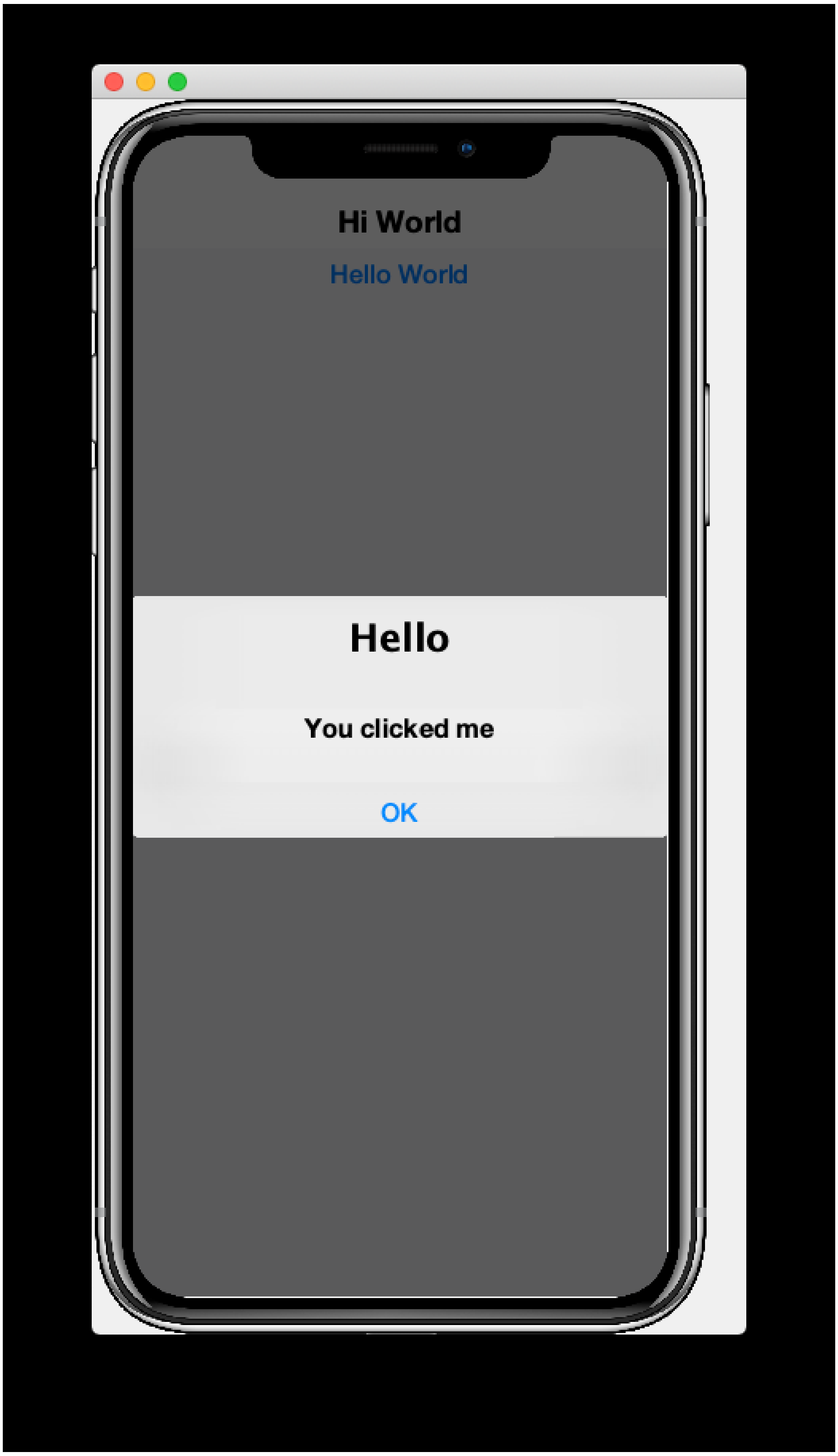
Calling `evt.consume()` is good practice as it signals to other interested parties that the event has been handled. This will prevent it from propagating any further to any other listeners to the `HELLO_CLICKED` action.

The `Dialog.show()` method shows a dialog on the screen.

If you save this change, you should see the simulator reload with the "Hello World" button now rendered as shown below:



And if you click on the button, it will display a dialog as shown here:

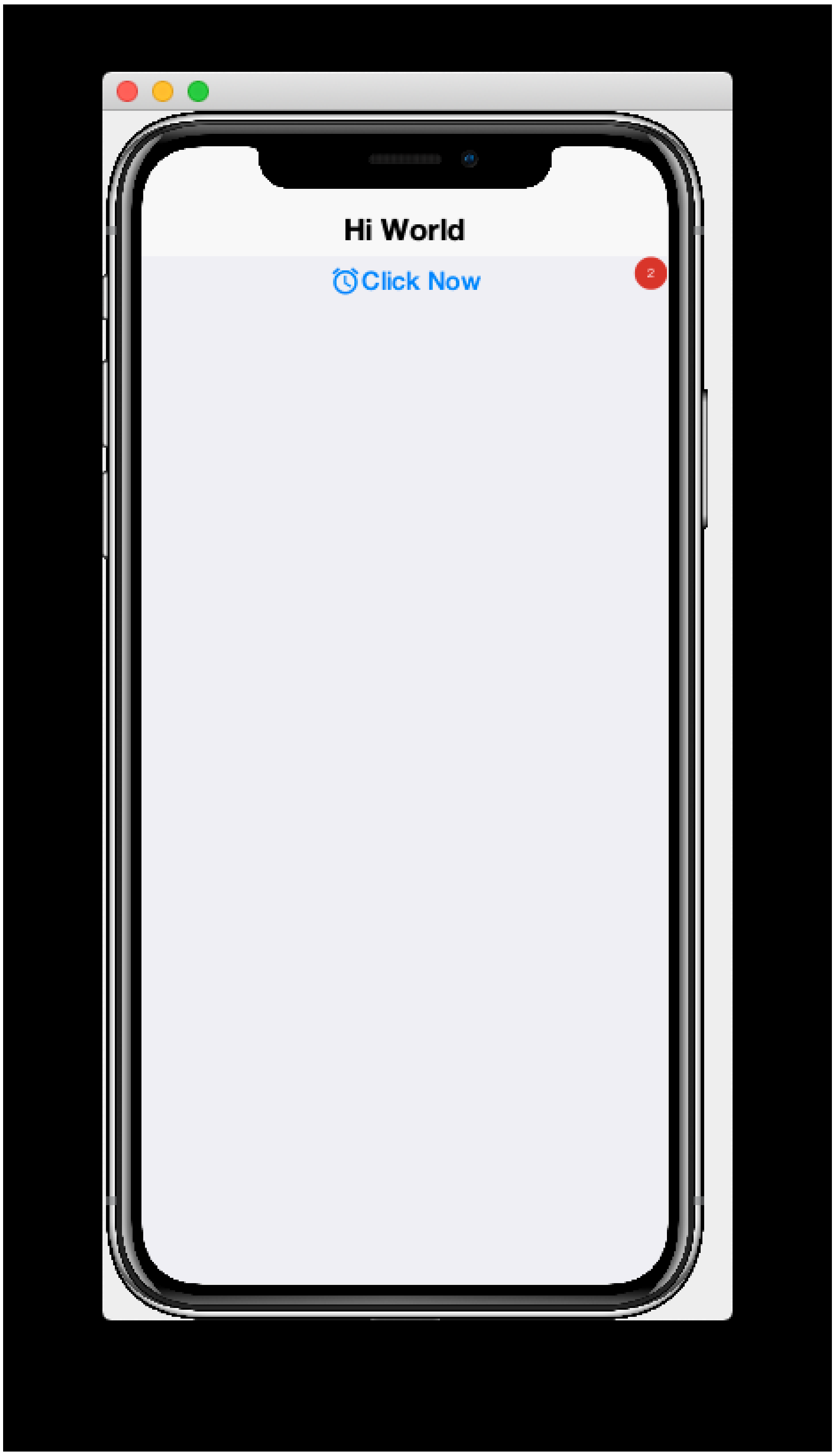


Customizing Action Rendering

In the previous example, the controller didn't make any recommendations to the view over how it wanted its *HELLO_CLICKED* action to be rendered. It simply registered an `ActionListener` and waited to be notified when it is "triggered". Let’s go a step further now, and specify an icon and label to use for the action. We will use the `ActionNode.Builder` class to build an action with the icon and label that we desire, and add it to the controller using the `addToController()` method of `ActionNode.Builder`.

Change your `initControllerActions()` method to the following and see how the action’s button changes in the simulator:

```
@Override
protected void initControllerActions() {
    super.initControllerActions();
    ActionNode.builder().
        icon(FontImage.MATERIAL_ALARM).
        label("Click Now").
        badge("2").
        addToController(this, StartPage.HELLO_CLICKED, evt -> {
            evt.consume();
            Dialog.show("Hello", "You clicked me", "OK", null);
        });
}
```



There’s quite a bit more that you can do with actions, but this small bit of foundation will suffice for our purposes for now.

Creating Menus

Whereas the `<button>` tag will create a single button, which can be optionally "bound" to a single action, the `<buttons>` renders multiple buttons to the view according to the actions that it finds in a given category. Let’s change the example from the previous section display a menu of buttons. We will:

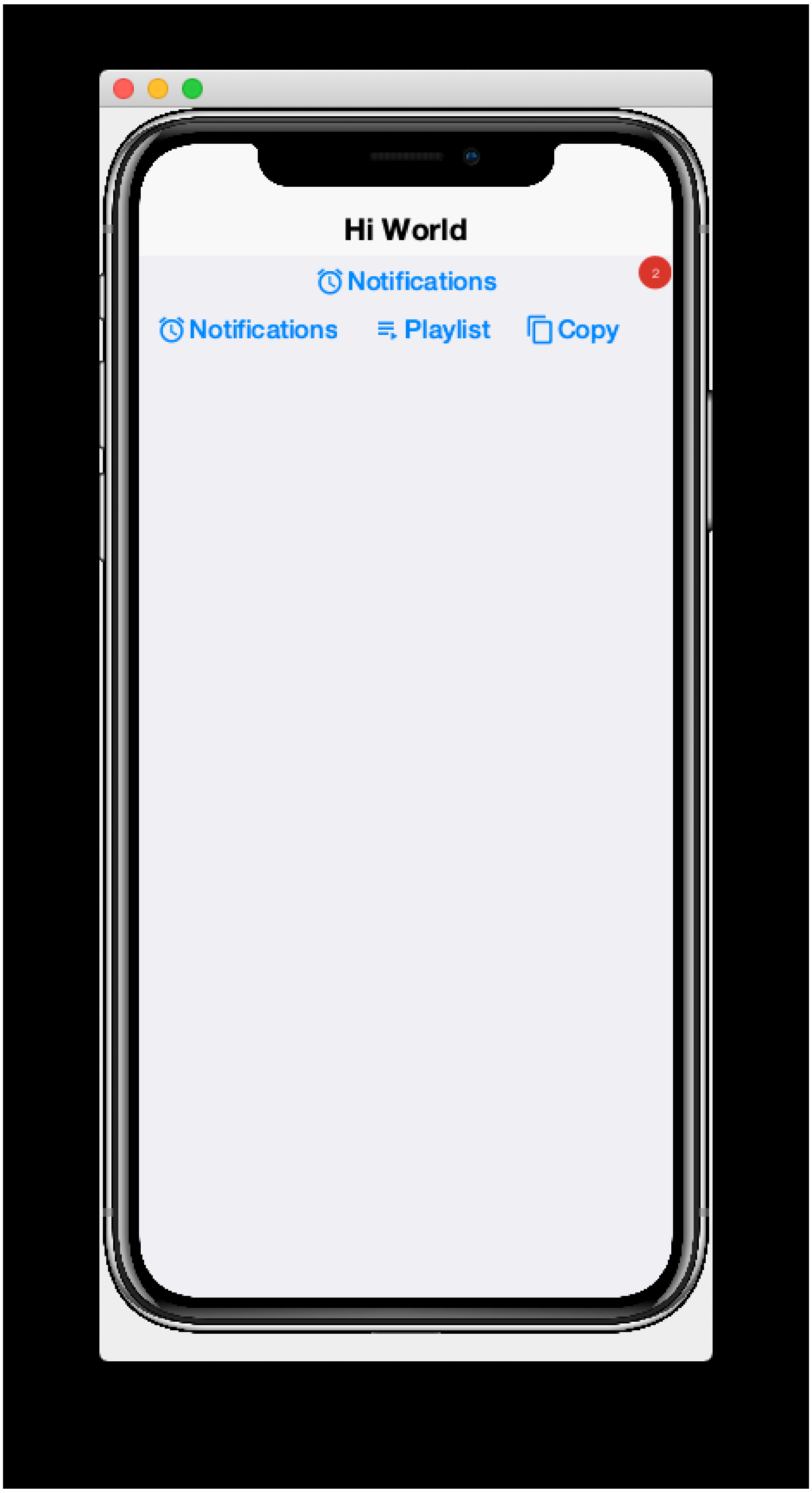
1. Define a new category called `MAIN_MENU`.
2. Add a `<buttons>` component to our view with `actionCategory="MAIN_MENU"`.
3. Define some actions in the controller, and register them with the new `MAIN_MENU` category.

```xml
<?xml version="1.0"?>
<y scrollableY="true" xsi:noNamespaceSchemaLocation="StartPage.xsd" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
    <define-category name="HELLO_CLICKED"/>

    <define-category name="MAIN_MENU" />
    <title>Hi World</title>
    <button text="Hello World">
        <bind-action category="HELLO_CLICKED"/>
    </button>
    <buttons actionCategory="MAIN_MENU"/>

</y>
```

And add the following to the `initControllerActions()` method of your controller class:

```java
ActionNode.builder().
        icon(FontImage.MATERIAL_ALARM).
        label("Notifications").
        addToController(this, StartPage.MAIN_MENU, evt -> {
            System.out.println("Notifications was clicked");
        });

ActionNode.builder().
        icon(FontImage.MATERIAL_PLAYLIST_PLAY).
        label("Playlist").
        addToController(this, StartPage.MAIN_MENU, evt -> {
            System.out.println("Playlist was clicked");
        });

ActionNode.builder().
        icon(FontImage.MATERIAL_CONTENT_COPY).
        label("Copy").
        addToController(this, StartPage.MAIN_MENU, evt -> {
            System.out.println("Copy was clicked");
        });
```

If all goes well, the simulator should reload to resemble the following screenshot:

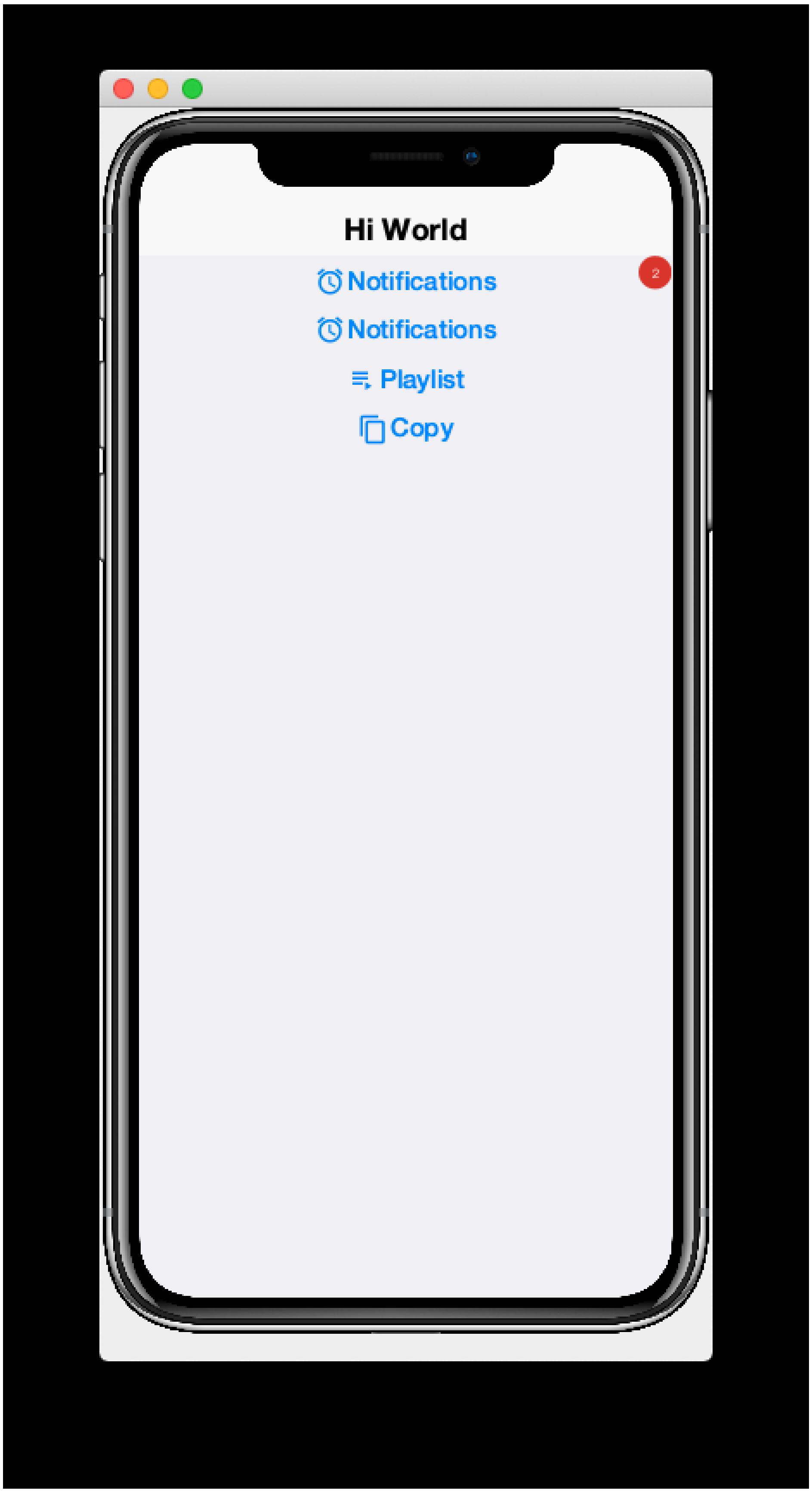


Buttons Layout

The `<buttons>` tag laid out all of the buttons in its specific action category. Currently they are all laid out on a single line. The default layout manager for the "Buttons" component is `FlowLayout`, which means that it will lay out actions horizontally from left to right (or right to left for RTL locales), and wrap to the next line when it runs out of space. It gives you quite a bit of flexibility for how the buttons are arranged and rendered, though. You can set the layout of `Buttons` to any layout manager that doesn't require a layout constraint. E.g. *BoxLayout*, *GridLayout*, and *FlowLayout*.

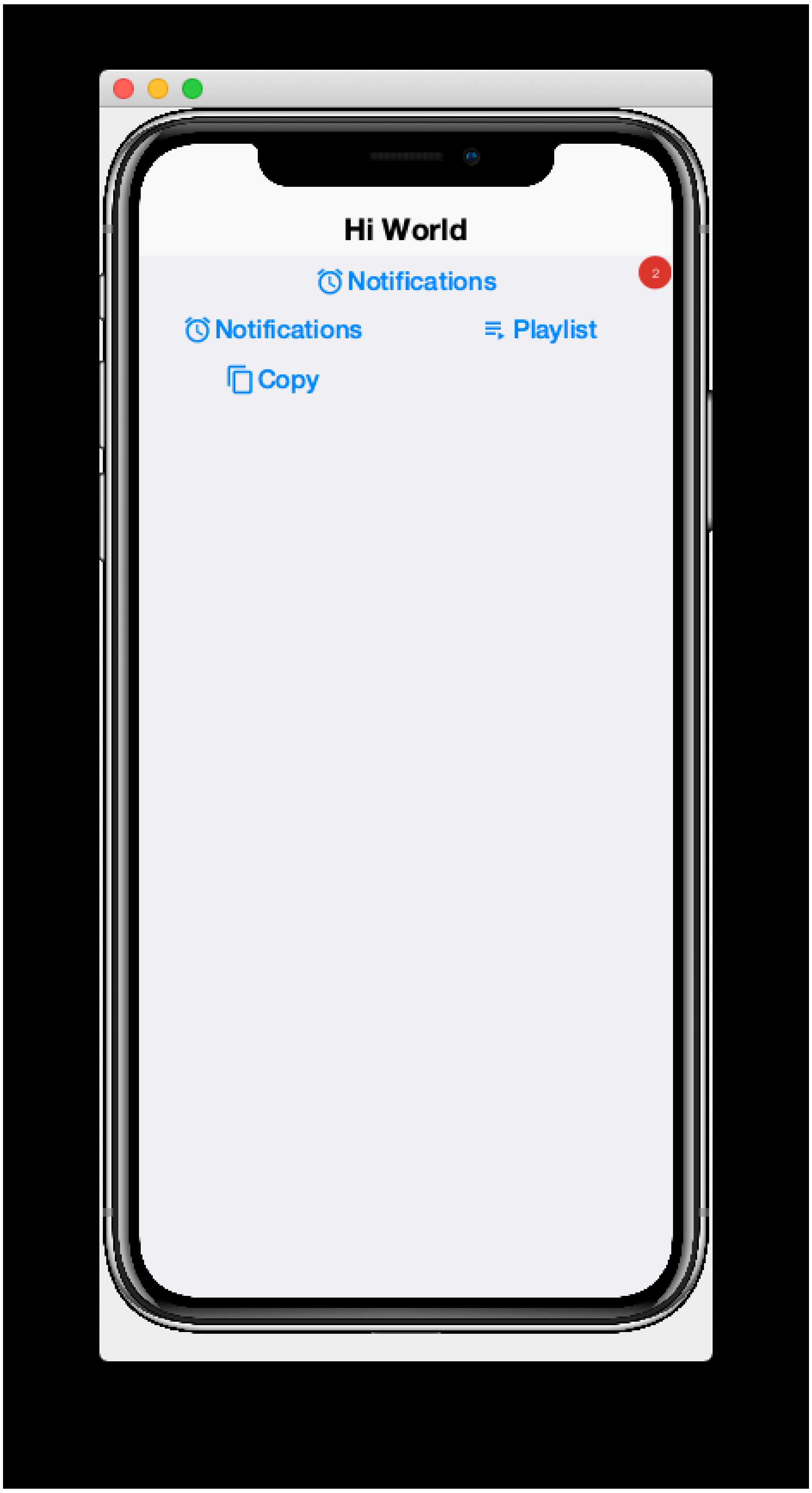
E.g. We can change the layout to *BoxLayout.Y* by setting the `layout=BoxLayout.y()` attribute:

```
<buttons layout="BoxLayout.y()" actionCategory="MAIN_MENU"/>
```



Or GridLayout using `layout="new GridLayout(2)"`:

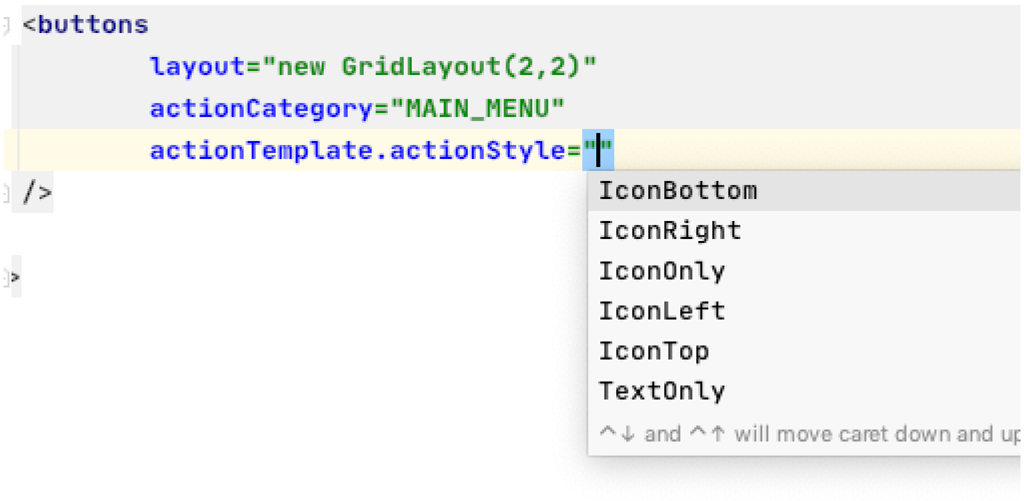
```
<buttons layout="new GridLayout(2,2)" actionCategory="MAIN_MENU"/>
```



Action Styles

Actions may include many preferences about how they should be rendered. The view is not obligated to abide by these preferences, but it usually at least considers them. We've already seen how actions can specify their preferred icons, labels, and badges, but there are several other properties available as well. One simple, but useful property is the *action style* which indicates whether the action should be rendered with both its icon and text, only its icon, or only its text. This is often overridden by the view based on the context. E.g. The view may include a menu of actions, and it only wants to display the action icons.

The `<buttons>` tag has an action template that defines "fallback" properties for its actions. These can be set using the `actionTemplate.*` attributes. For example, try adding the `actionTemplate.actionStyle` attribute to your `<buttons>` tag. You should notice that the editor gives you a drop-down list of options for the value of this attribute as shown below:



```
<buttons
        layout="new GridLayout(2,2)"
        actionCategory="MAIN_MENU"
        actionTemplate.actionStyle=""
/>                                    IconBottom
                                      IconRight
>                                     IconOnly
                                      IconLeft
                                      IconTop
                                      TextOnly
                                      ^↓ and ^↑ will move caret down and up
```

Try selecting different values for this attribute and save the file after each change to see the result in the simulator. You should see something similar to the following:

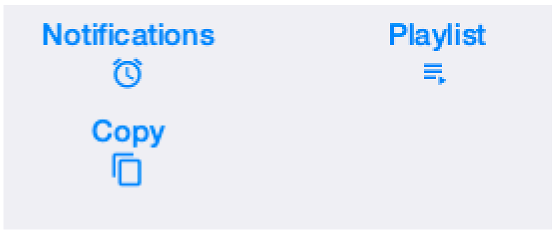


Figure 4. IconBottom

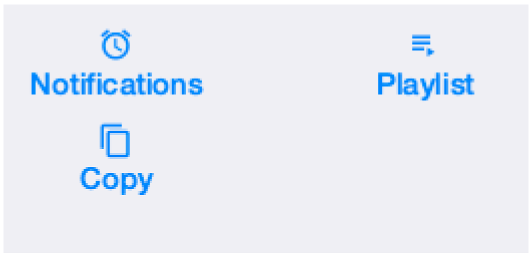


Figure 5. IconTop

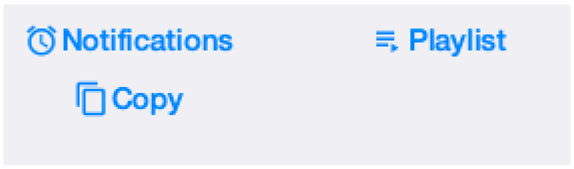


Figure 6. IconLeft

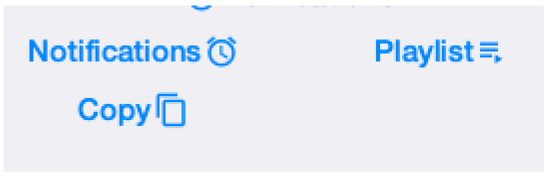


Figure 7. IconRight

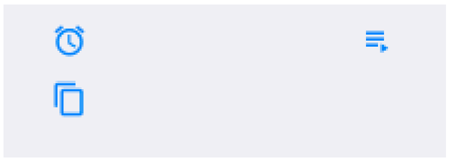


Figure 8. IconOnly

You can also specify UIIDs for the actions to customize things like font, color, borders, padding, etc... To learn more about the various options available, see the Actions chapter of the manual. (TODO: Create actions section of manual).

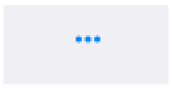
Overflow Menus

In some cases, your view may only have room for one or two buttons in the space provided, but you want to be able to support more actions than that. You can use the *limit* attribute to specify the maximum number of buttons to render. If the number of actions in the action category is greater than this limit, it will automatically add an overflow menu for the remainder of the actions.

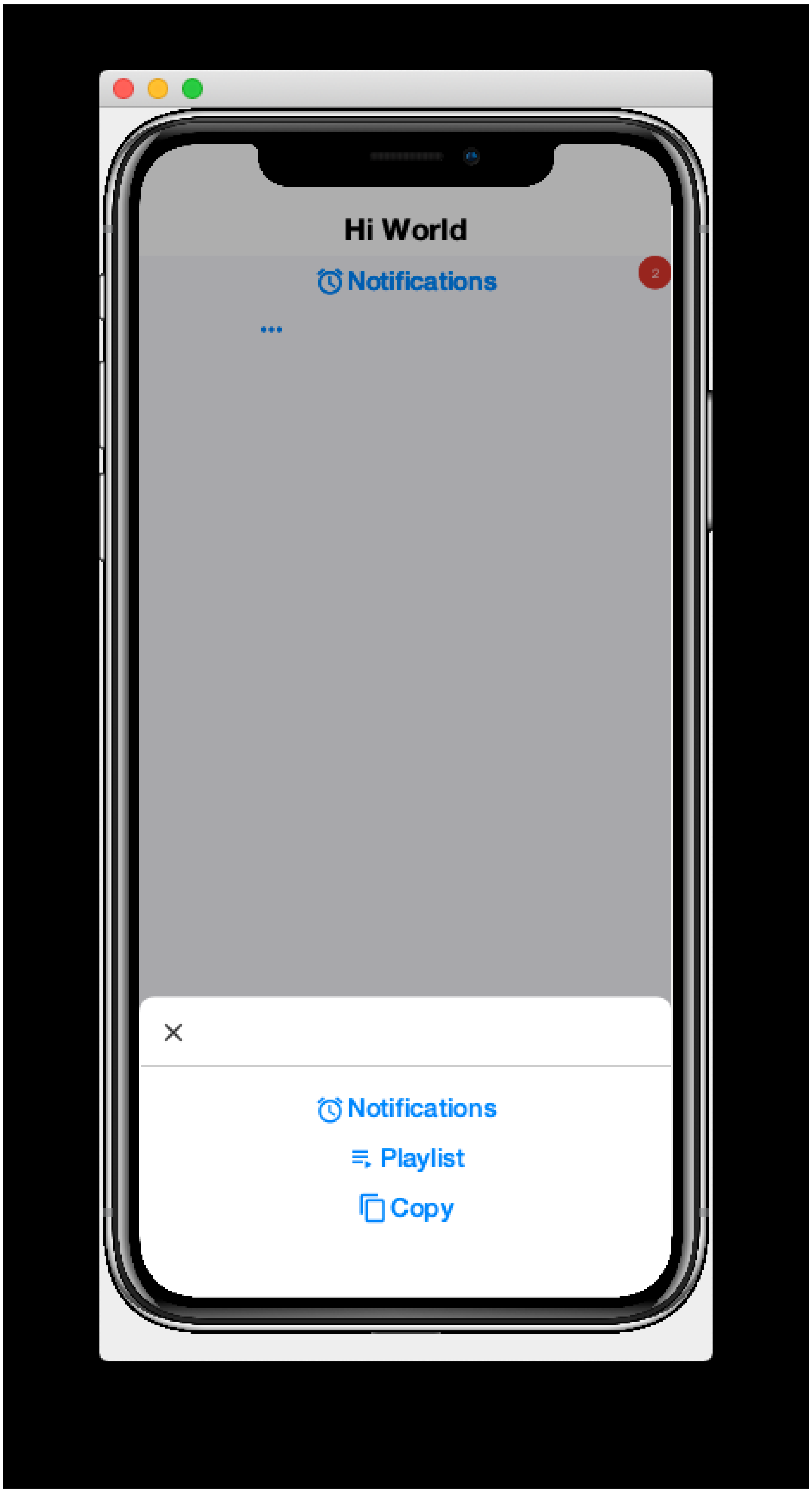
Try adding `limit=1` to the `<buttons>` tag and see what happens:

```
<buttons
        layout="new GridLayout(2,2)"
        actionCategory="MAIN_MENU"
        actionTemplate.actionStyle="IconOnly"
        limit="1"
/>
```

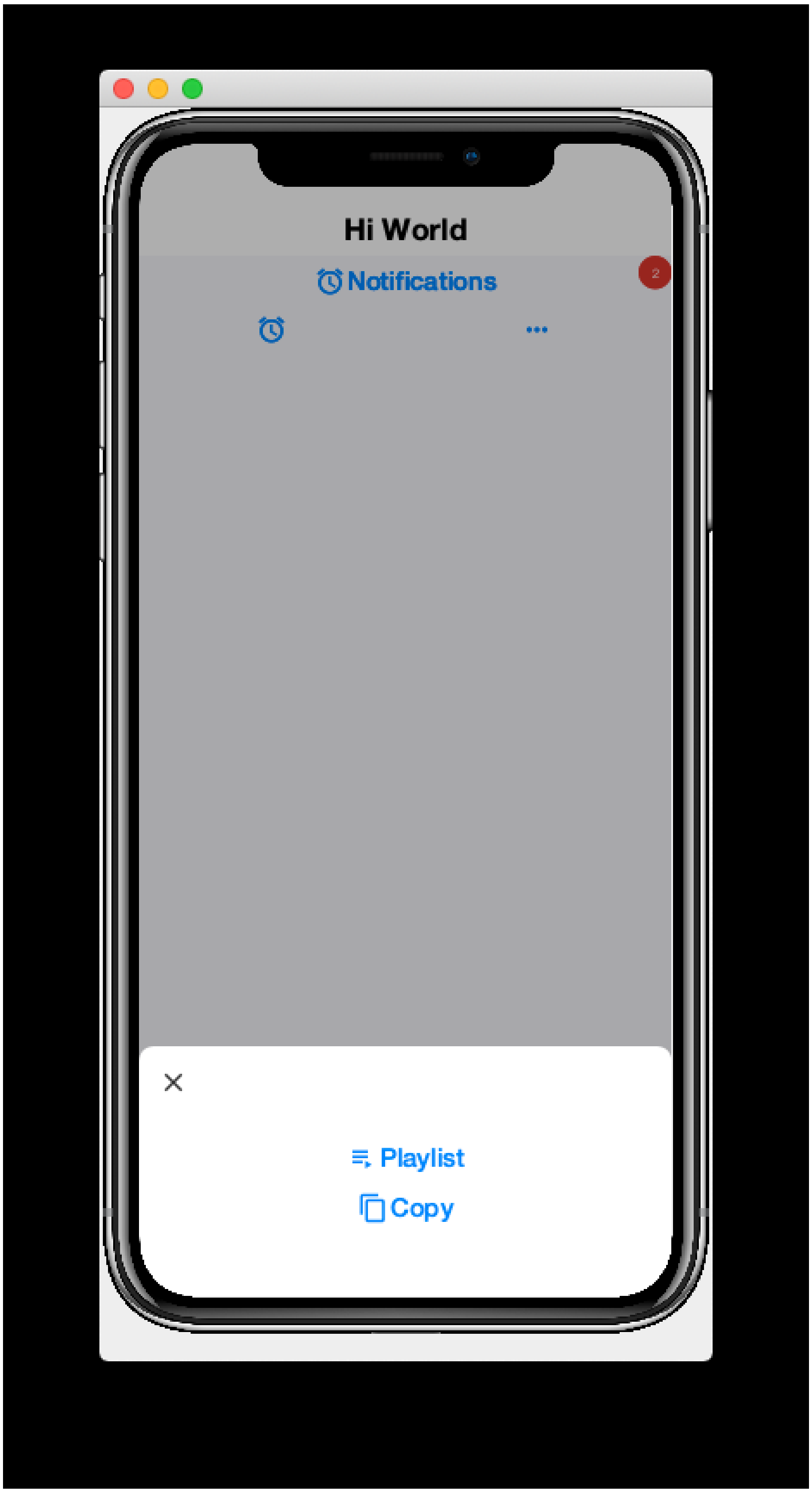
When the simulator reloads you will see only a "More" button where the menu items once were:



If you press this button, you will be presented with an Action Sheet with the actions.

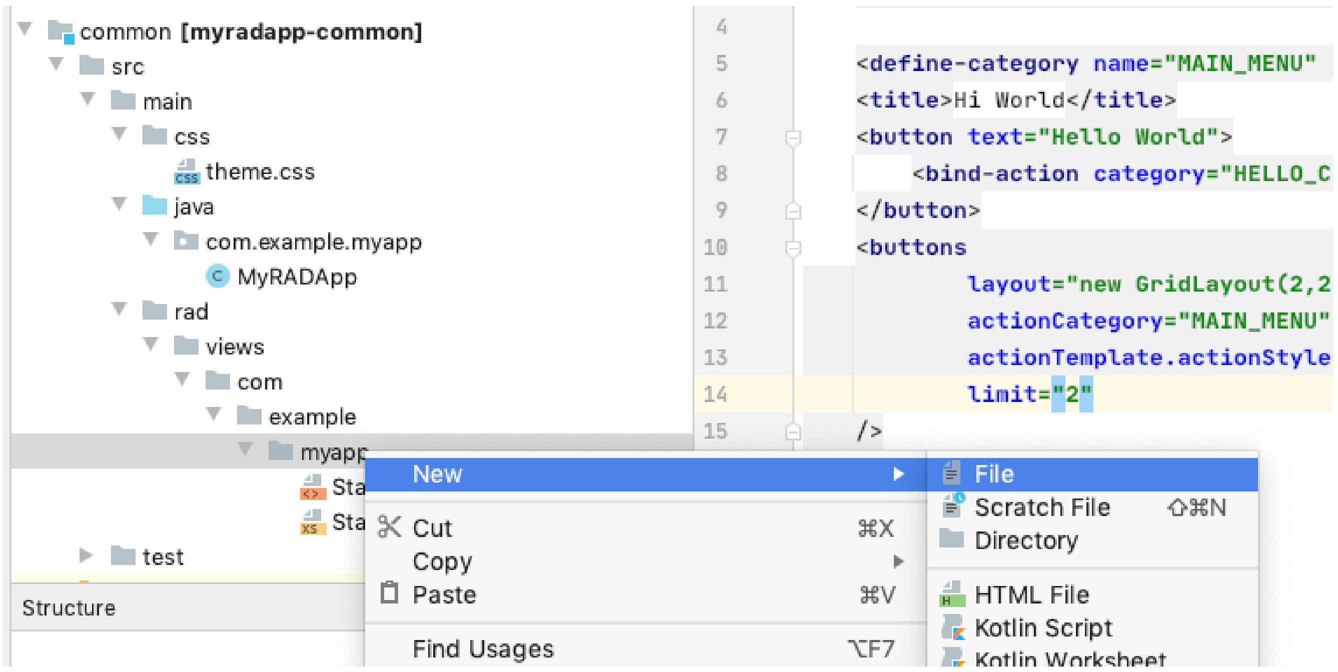


If you change the limit to "2", it will show the first action, *Notifications*, in the buttons, and then it will show the remaining two actions when the user presses the "More" button.



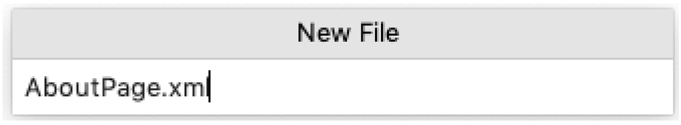
Form Navigation

It’s time to grow beyond our single-form playpen, and step into the world of multi-form apps. Let’s create another view in the same folder as *StartPage.xml*. We’ll name this *AboutPage.xml*. If you’re using IntelliJ, like me, you can create this file by right clicking the "myapp" directory in the project inspector, and select *New* > *File* as shown here:



```
<define-category name="MAIN_MENU"
<title>Hi World</title>
<button text="Hello World">
    <bind-action category="HELLO_C
</button>
<buttons
        layout="new GridLayout(2,2
        actionCategory="MAIN_MENU"
        actionTemplate.actionStyle
        limit="2"
/>
```

Then enter "AboutPage.xml" in the dialog:



New File

AboutPage.xml

And press *Enter*

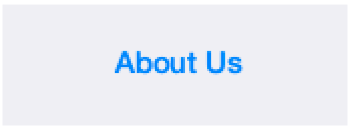
Add the following placeholder contents to the newly created *AboutPage.xml* file:

```
<?xml version="1.0"?>
<y>
    <title>About Us</title>
    <label>Under construction</label>
</y>
```

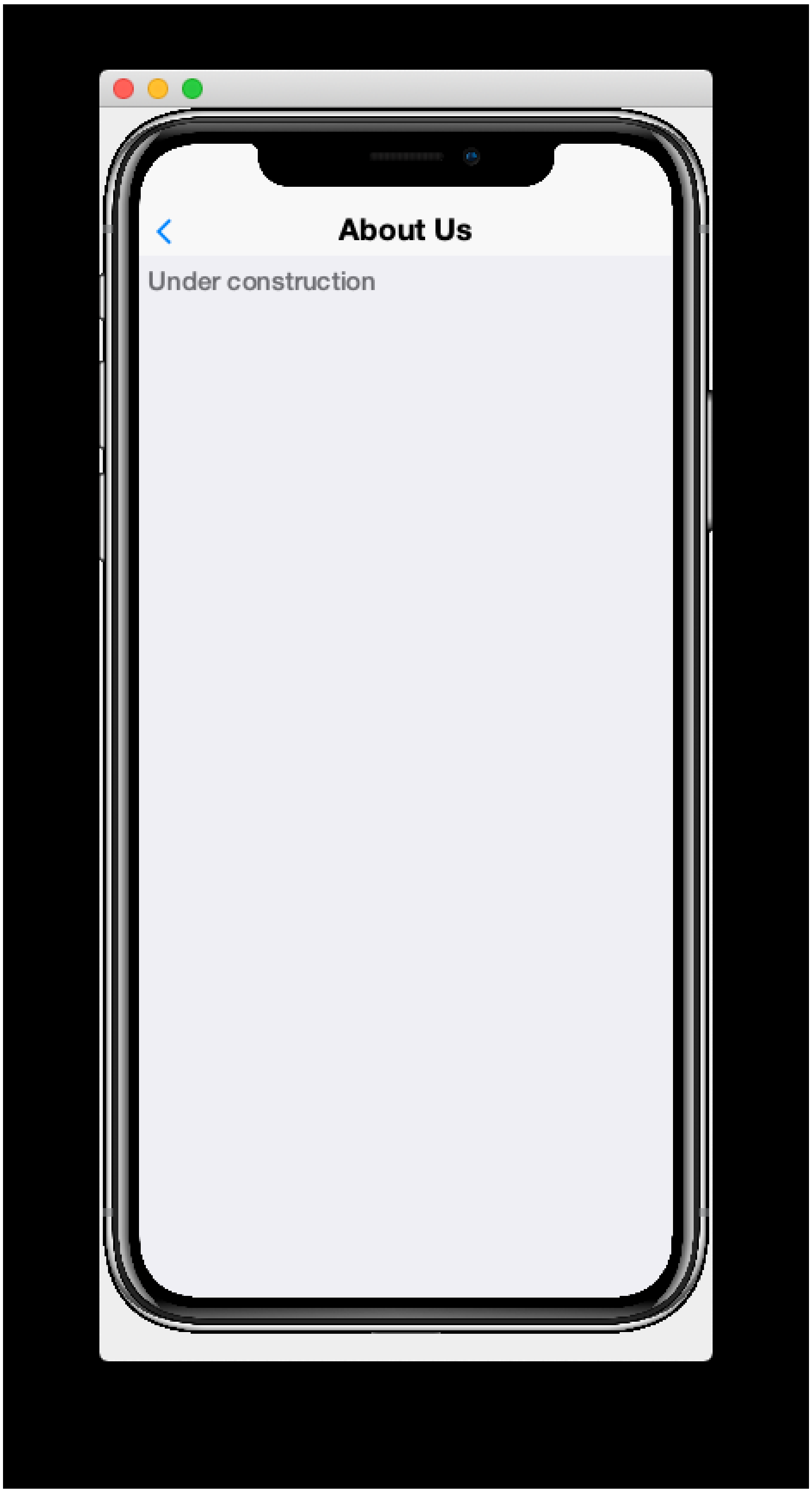
Finally, let's add a button to our original view, *StartPage.xml* as follows:

```
<button rad-href="#AboutPage">About Us</button>
```

When the simulator reloads, you should now see this button:



Click on this button, and it should take you to the "About Us" view we just created.



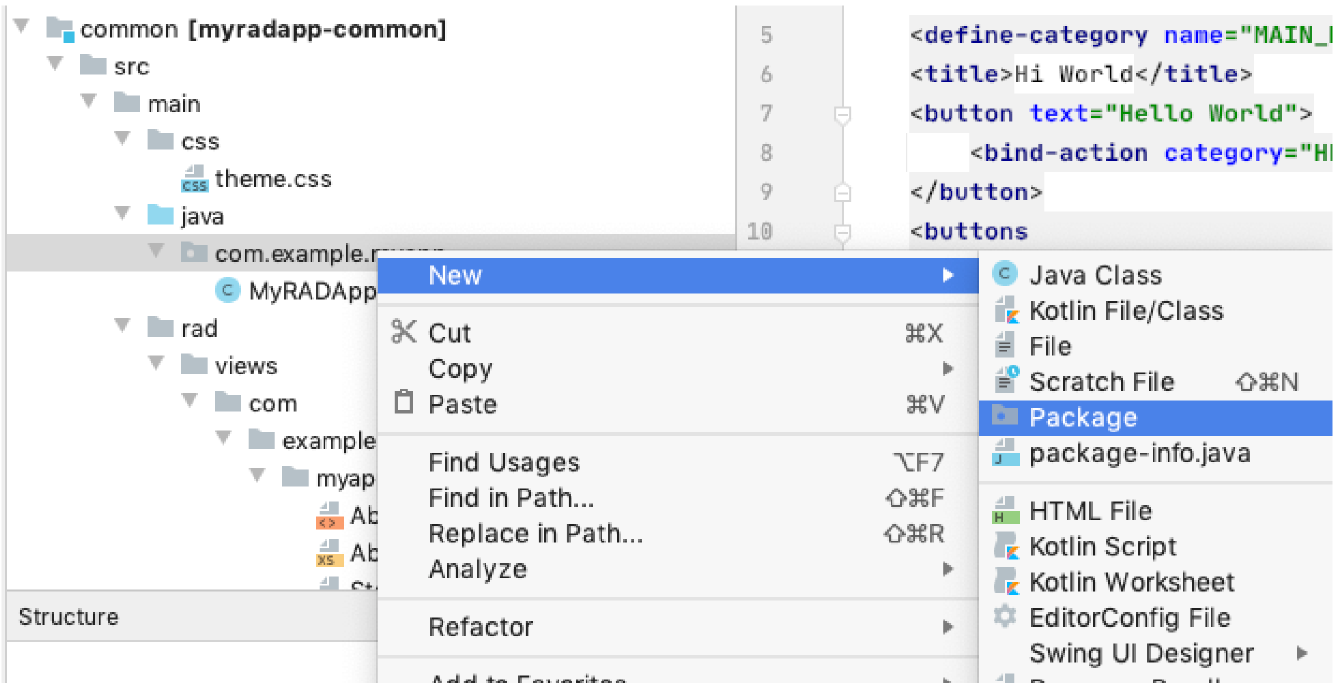
Notice that the *About Us* form includes a *Back* button that returns you to the *Start Page*. This is just one of the nice features that you get for free by using CodeRAD. There is a lot of power packed into the `rad-href` attribute. In this case we specified that we wanted to link to the *AboutPage* view using the "#AboutPage" URL, it enables other URL types as well. To learn more about the *rad-href* attribute, see (TODO section of manual on rad-href).

Models

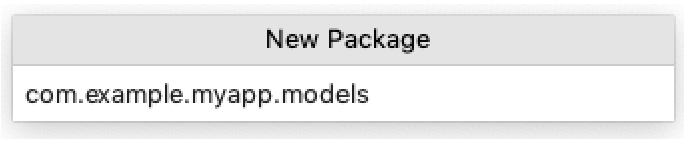
So far we've been working only with the *V* and *C* portions of *MVC*. In this section, I introduce the final pillar in the trinity: *the Model*. Model objects store the data of the application. In CodeRAD, *model* objects implement the *com.codename1.rad.models.Entity* interface. We're going to skip the conceptual discussion of *Models* in this tutorial, and dive directly into an example so you can see how they work. After we've played with some models, we'll circle back and discuss the theories and concepts in greater depth.

Most apps need a model to encapsulate the currently logged-in user. Let's create model named *UserProfile* for this purpose.

Create a new package named "com.example.myapp.models". In IntelliJ, you can achieve this by right clicking on the *com.example.myapp* node in the project inspector (inside the *src/main/java* directory of the *common* module), and select *New* > *Package*, as shown here:


```
<define-category name="MAIN_I
<title>Hi World</title>
<button text="Hello World">
    <bind-action category="HI
</button>
<buttons
```

Then enter "models" for the package name in the dialog:


New Package

com.example.myapp.models

Now create a new Java interface inside this package named "UserProfile".

```
package com.example.myapp.models;

import com.codename1.rad.annotations.RAD;
import com.codename1.rad.models.Entity;
import com.codename1.rad.models.Tag;
import com.codename1.rad.schemas.Person;

@RAD ①
public interface UserProfile extends Entity {

    /*
     * Declare the tags that we will use in our model. ②
     */
    public static final Tag name = Person.name;
    public static final Tag photoUrl = Person.thumbnailUrl;
    public static final Tag email = Person.email;

    @RAD(tag="name") ③
    String getName();
    void setName(String name);

    @RAD(tag="photoUrl")
    String getPhotoUrl();
    void setPhotoUrl(String url);

    @RAD(tag="email")
    String getEmail();
    void setEmail(String email);
}
```

① The `@RAD` annotation before the interface definition activates the CodeRAD annotation processor, which will generate a concrete implementation of this interface (named *UserProfileImpl) and a _wrapper* class this interface (named *UserProfileWrapper*). More *wrapper classes* shortly.

② We declare and import the tags that we intend to use in our model. Tags enable us to create views that are loosely coupled to a model. Since our *UserProfile* represents a person, we will tag many of the properties with tags from the *Person* schema.

③ The `@RAD` annotation before the `getName()` method directs the annotation processor to generate a *property* named "name". The `tag="name"` attribute means that this property will accessible via the *name* tag. This references the `public static final Tag name` field that we defined at the beginning of the interface definition. More on tags shortly.

Next, let's create a view that allows us to view and edit a *UserProfile*.

In the same directory as the *StartPage.xml* file, create a new file named *UserProfilePage.xml* with the following contents:

```
<?xml version="1.0" ?>

<y rad-model="UserProfile" xsi:noNamespaceSchemaLocation="UserProfilePage.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <import>
        import com.example.myapp.models.UserProfile;
    </import>
    <title>My Profile</title>
    <label>Name:</label>
    <radLabel tag="Person.name"/>
    <label>Email:</label>
    <radLabel tag="Person.email" />
</y>
```

This view looks very similar to the *StartPage* and *AboutPage* views we created before, but it introduces a couple of new elements:

`rad-model="UserProfile"`
: This attribute, added to the root element of the XML document specifies that this view's *model* will a *UserProfile*.

IMPORTANT	Remember to import `UserProfile` class in the `<import>` tag, or the view will fail to compile because it won't know where to find the *UserProfile* class.

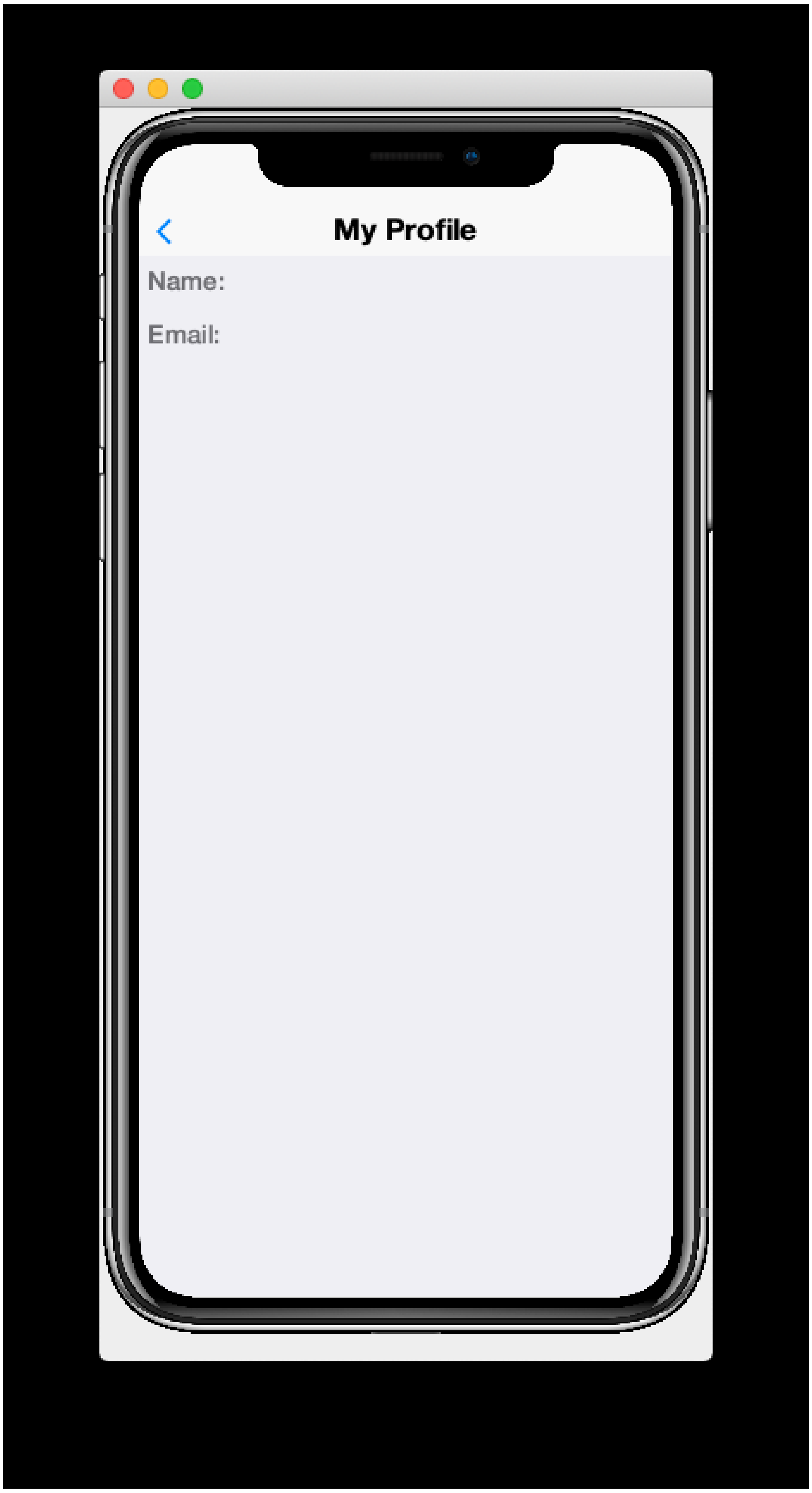
`<radLabel tag="Person.name"/>`
: The `<radLabel>` tag is a wrapper around a `Label` that supports binding to a model property. In this case the `tag=Person.name` attribute indicates that this label should be bound to the property of the model with the `Person.name` tag. Recall that the *name* property of the *UserProfile* included the `@RAD(tag="name")` annotation, which effectively "tagged" the property with the "name" tag.

TIP	In this example I chose to reference the `Person.name` tag from the *Person* schema, but since our *UserProfile* class referenced this tag in its `name` static field, we could have equivalently referenced `tag="UserProfile.name"` here.

Before we fire up the simulator, we also need to add a *link* to our new form so we can test it out. Add a button to the *StartPage* view that links to our *UserProfilePage*:

```
<button rad-href="#UserProfilePage">User Profile</button>
```

Now fire up the simulator and click on the *User Profile* button we added. YOu should see something like the this:



This is a little boring right now because we haven't specified a *UserProfile* object to use as the model for this view, so it just creates a new (empty) instance of *UserProfile* and uses that. Let's remedy that by instantiating a *UserProfile* in our controller, and then use *that* profile as the view for our profile.

Open the RADApp class and implement the following method:

```
@Override
protected void onStartController() {
    super.onStartController();

    UserProfile profile = new UserProfileImpl();
    profile.setName("Jerry");
    profile.setEmail("jerry@example.com");
    addLookup(UserProfile.class, profile);
}
```

TIP The `onStartController()` method is the preferred place to add initialization code for your controller. Placing initialization here rather than in the constructor ensures the controller is "ready" to be initailized.

Most of this snippet should be straight forward. I'll comment on two aspects here:

1. We use the `UserProfileImpl` class, which is the default concrete implementation of our *UserProfile* entity that was generated for us by the annotation processor.
2. The `addLookup()` method adds a *lookup* to our controller so that the profile we just created can be accessed throughout the app by calling the `Controller.lookup()` method, passing it `UserProfile.class` as a parameter. Lookups are used throughout CodeRAD as they are a powerful way to "share" objects between different parts of your app while still being loosely coupled.

Now, we will make a couple of changes to the *StartPage* view to inject this profile into the *UserProfile* view.

First, we need to add *UserProfile* to the *imports* of *StartPage*.

```
<import>
import com.example.myapp.models.UserProfile;
</import>
```

Next, add the following tag somewhere in the root of the *StartPage.xml* file:

```
<var name="profile" lookup="UserProfile"/>
```

This declares a "variable" named *profile* in our view with the value of the *UserProfile* lookup. This is roughly equivalent to the java:

```
UserProfile profile = controller.lookup(UserProfile.class);
```

Finally, change the `<button>` tag in the *StartPage* that we used to link to the *UserProfile* page to indicate that it should use the *profile* as the model for the *UserProfilePage*:

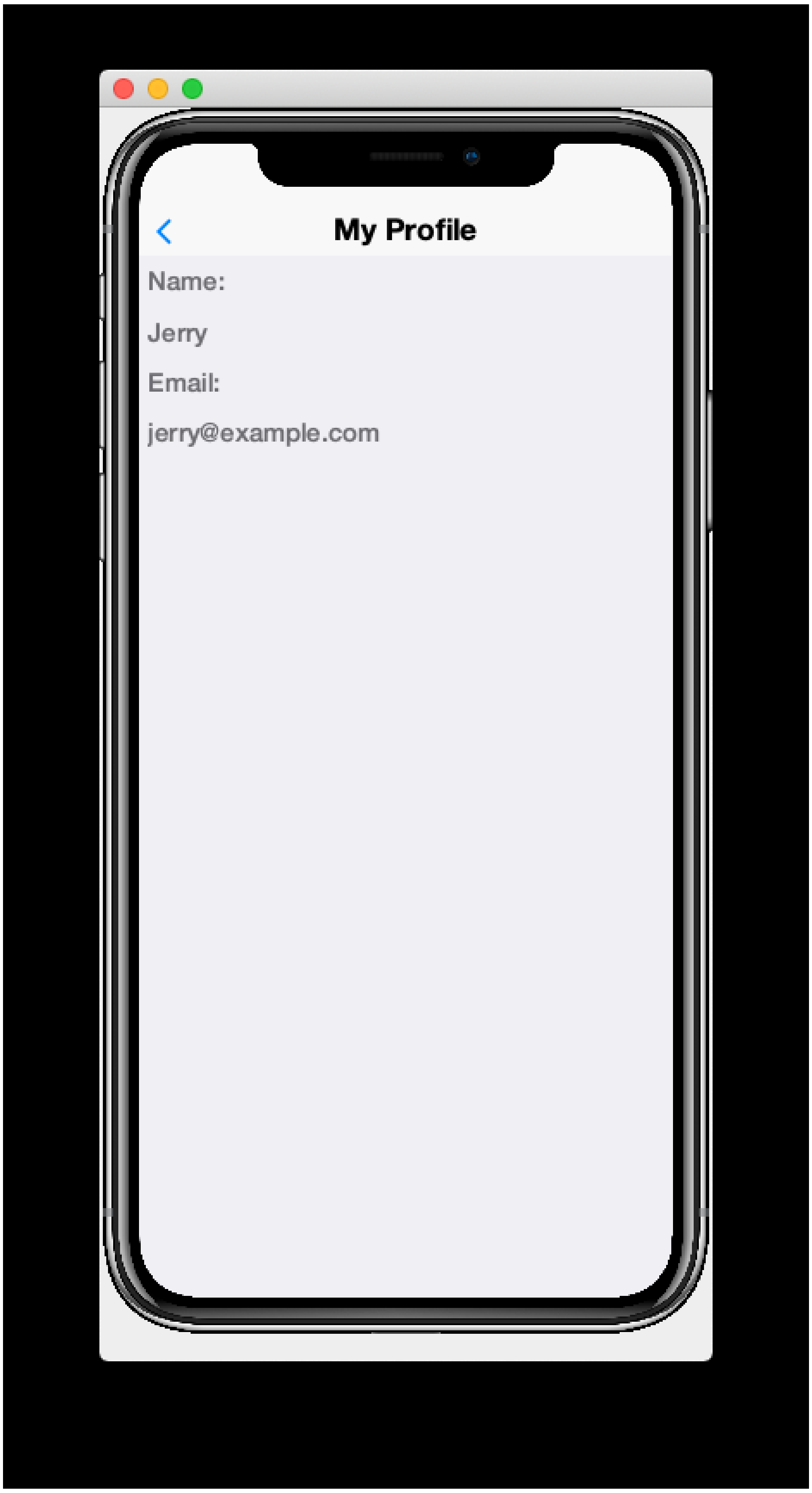
```
<button rad-href="#UserProfilePage{profile}">User Profile</button>
```

The active ingredient we added here was the "{profile}" suffix to the URL. This references the `<var name="profile"…>` tag we added earlier.

When we're done, the `StartPage.xml` contents will look like:

```
<?xml version="1.0"?>
<y scrollableY="true" xsi:noNamespaceSchemaLocation="StartPage.xsd" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
    <!-- We need to import the UserProfile class since we use it
        in various places of this view. -->
    <import>
        import com.example.myapp.models.UserProfile;
    </import>

    <!-- Reference to the UserProfile looked up
        from the Controller.  This lookup is registered
        in the onStartController() method of the MyRADApp class. -->
    <var name="profile" lookup="UserProfile"/>
    <define-category name="HELLO_CLICKED"/>

    <define-category name="MAIN_MENU" />
    <title>Hi World</title>
    <button text="Hello World">
        <bind-action category="HELLO_CLICKED"/>
    </button>
    <buttons
            layout="new GridLayout(2,2)"
            actionCategory="MAIN_MENU"
            actionTemplate.actionStyle="IconOnly"
            limit="2"
    />
    <button rad-href="#AboutPage">About Us</button>

    <!-- This button links to the UserProfilePage
         The {profile} suffix means that the UserProfilePage
         should use the "profile" reference created by
         the <var name="profile"...> tag above.
     -->
    <button rad-href="#UserProfilePage{profile}">User Profile</button>

</y>
```

Now, we we click on the *User Profile* button, it should display the details of the profile we created:



Name:

Jerry

Email:

jerry@example.com

The "Poor Man's" Hot Reload

CodeRAD has experimental support for full, proper, *Hot reload* but it currently requires some additional setup that is beyond the scope of this *getting started* tutorial.

Throughout this tutorial, we have been using a sort of *pseudo* hot-reload mechanism whereby the simulator reloads your app entirely when it detects changes to your app's source code. This is actually quite efficient if you are working on the *first* form of your app. But if you are working on a form that you need to navigate to, it can become tedious, since with each change to your source, your app will restart, and you'll need to navigate to the form of interest *again*.

In order to simplify our lives, it can be helpful to *temporarily* modify the application controller to show the form that you are currently "working" on. In our case, we will be working on the *UserProfilePage* for the next few sections, so I will add the following to the `actionPerformed()` method of the `MyRADApp` class:

```
if (true) {
    new UserProfilePageController(this, lookup(UserProfile.class)).show();
    return;
}
```

The full `actionPerformed()` method would, then, look something like :

```
public void actionPerformed(ControllerEvent evt) {

    with(evt, StartEvent.class, startEvent -> {
        if (true) {
            new UserProfilePageController(this, lookup(UserProfile.class)).show(
);
            return;
        }
        startEvent.setShowingForm(true);
        new StartPageController(this).show();
    });
    super.actionPerformed(evt);
}
```

There are a couple of things worth commenting on here:

1. The `UserProfilePageController` class is a *FormController* subclass that is generated by the annotation processor. It is a default implementation of a controller for displaying the `UserProfilePage` view.
2. The second parameter of the `UserProfilePageController` constructor takes the *view model* for the `UserProfilePage` view as a parameter. In this case we are using the `UserProfile` object that we added as a lookup previously in the `onStartController()` method.

This little snippet will cause the app to *always* show the *UserProfilePage* when it starts. We'll remove this once we have finished "working on" the *UserProfilePage* view.

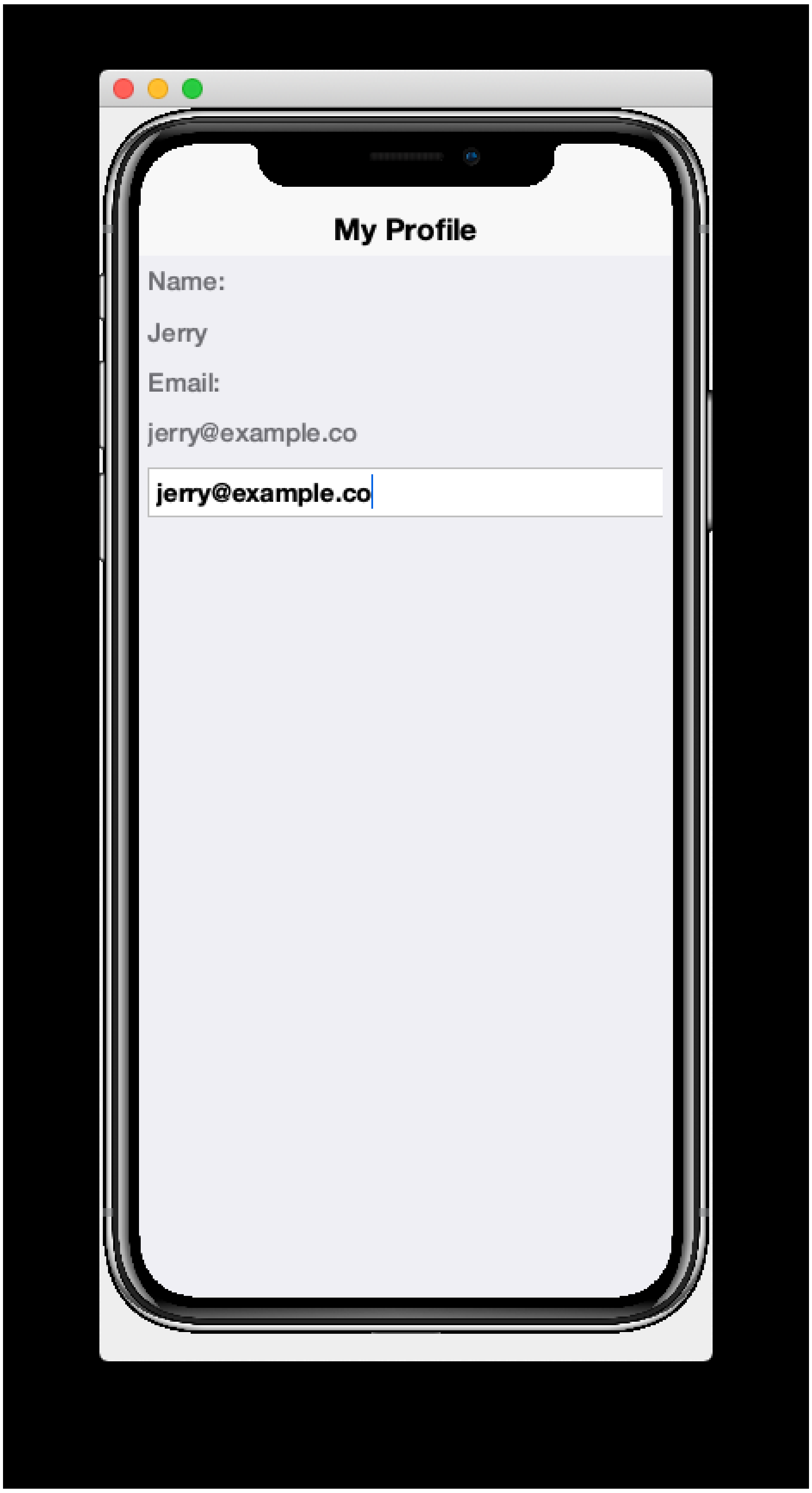
Fun with Bindings

TIP Throughout this guide I use the terms *model* and *entity* interchangeably because CodeRAD names it's *model* class `Entity`.

CodeRAD models are designed to allow for easy binding to other models and to user interface components. We've already seen how the `<radLabel>` tag can be bound to a model property using the `tag` attribute, but you aren't limited to static labels. There are `radXXX` components for many of the fundamental Codename One components. E.g. `<radTextField>`, `<radTextArea>`, `<radSpanLabel>`, and many more. Later on, you'll also learn how to build your own *binding* components, but for now, let's have a little bit of fun with the standard ones.

To demonstrate that you can bind more than one component to the same property, let's add a `<radTextField>` that binds to the *email* property just below the existing `<radLabel>`.

```
<radTextField tag="Person.email"/>
```



You’ll notice that as you type in the *email* text field, the value of the *email* label also changes. This is because they are bound to the same property of the same model.

We can even go a step further. It is possible to bind *any* any property to the result of an arbitrary Java expression so that the property will be updated whenever the model is changed.

As an example, let's add a button that is enabled *only* when the model's *email* property is non-empty:

```
<button bind-enabled="java:!getEntity().isEmpty(UserProfile.email)">Save</button>
```

TIP The *bind-** attributes, by default expect their values to be references to a tag (e.g. `UserProfile.email`), but you can alternatively provide a Java expression prefixed with `java:`.

You will notice, now, that if you delete the content of the *email* text field on the form, the *Save* button becomes disabled. If you start typing again, the button will become enabled again.

In this example we bound the *enabled* property of *Button* so that it would be updated whenever the model is changed. You aren't limited to the *enabled* property though. You can bind on any property you like. You can even bind on sub-properties, e.g.:

```
<button bind-style.fgColor="java:getEntity().isEmpty(UserProfile.email) ? 0xff0000 :
0x0">Save</button>
```

In the above example, the button text will be red when the email field is empty, and black otherwise.

Transitions

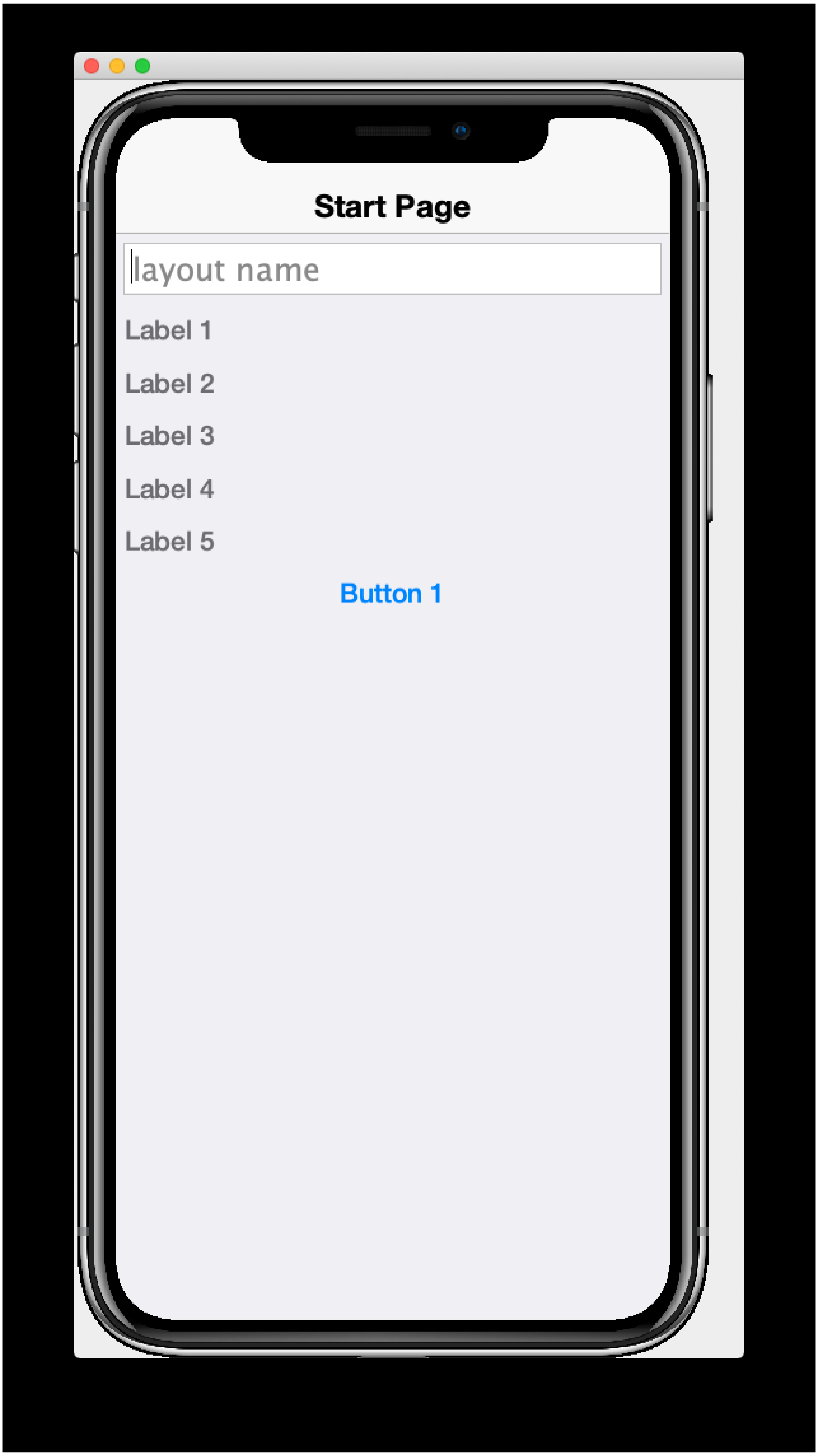
By default, changes to bound properties take effect immediately upon property change. For example, if you bind the *visible* property of a label, then it will instantly appear when the value changes to true, and instantly disappear when the value changes to false. Interfaces feel *better* when changes are animated.

The *rad-transition* attribute allows you to specify how transitions are handled on property bindings. Attributes that work particularly well with transitions are ones that change the size or layout of a component.

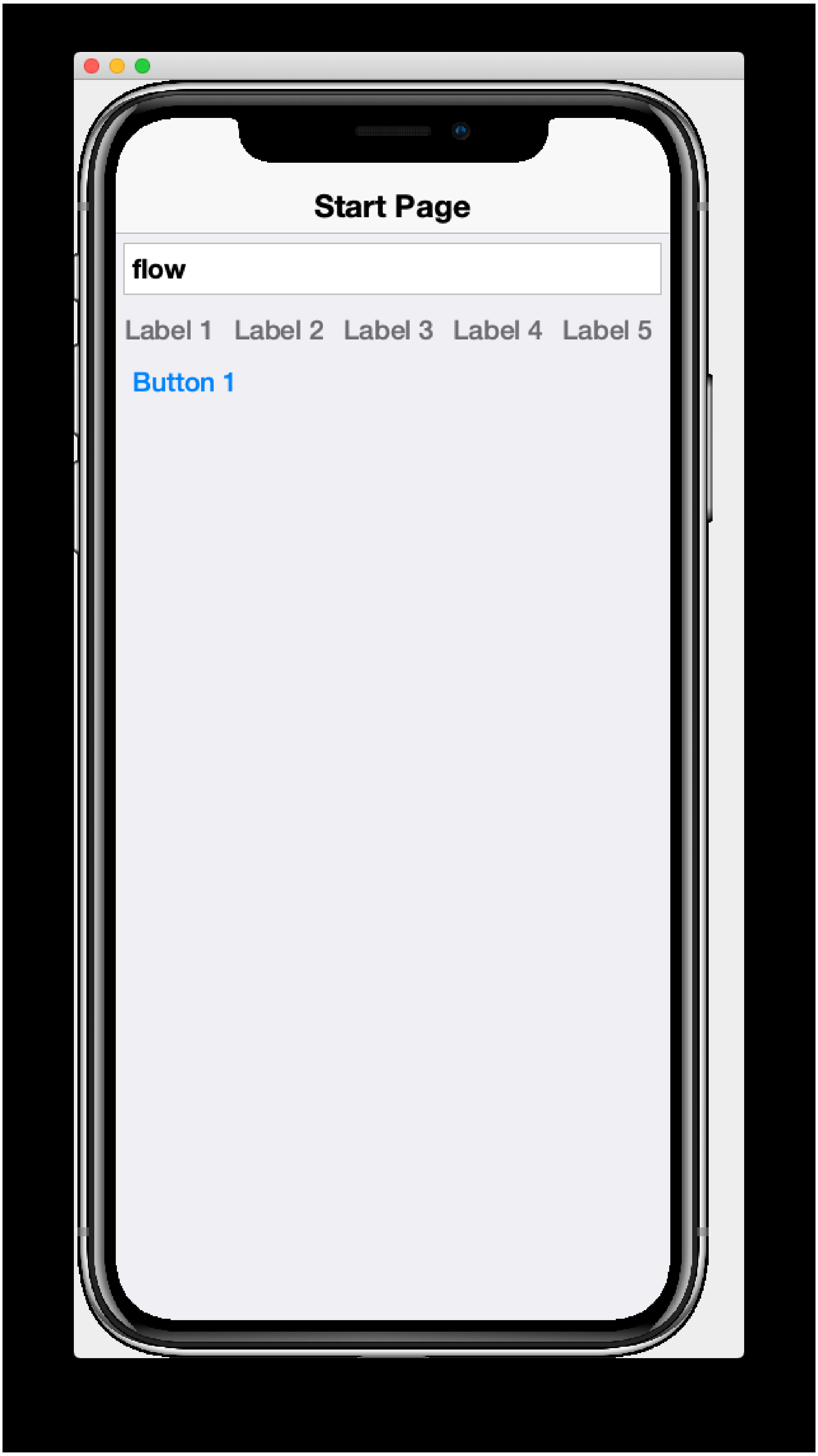
The following example binds the "layout" attribute on a container so that if the user enters "flow" into the text field, the layout will change to a *FlowLayout*, and for any other value, the layout will be *BoxLayout.Y*:

```xml
<?xml version="1.0"?>
<border xsi:noNamespaceSchemaLocation="StartPage.xsd" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
    <title>Start Page</title>

    <!-- Define a tag for the layout property.
            This will add a property to the auto-generated view model class.
    -->
    <define-tag name="layout"/>

    <!-- A text field that is bound to the "layout" property
            As user types, it updates the "layout" property of the view model. -->
    <radTextField tag="layout"  layout-constraint="north"/>

    <!-- A Container with initial layout BoxLayout.Y.
        We bind the "layout" property to a java expression that will set layout
        to FlowLayout if the model's "layout" property is the string "flow", and
        BoxLayout.Y otherwise.

        The rad-transition="layout 1s" attribute will cause changes to the "layout"
property
        to be animated with a duration of 1s for each transition.
    -->
    <y bind-layout='java:"flow".equals(getEntity().getText(layout)) ? new FlowLayout()
: BoxLayout.y()'
       rad-transition="layout 1s"
       layout-constraint="center"
    >
        <label>Label 1</label>
        <label>Label 2</label>
        <label>Label 3</label>
        <label>Label 4</label>
        <label>Label 5</label>
        <button>Button 1</button>

    </y>

</border>
```

If you run the above example, it will begin with rendering the labels vertically in a *BoxLayout.Y* layout as shown below:


Start Page
layout name
Label 1
Label 2
Label 3
Label 4
Label 5
Button 1

If you type the word "flow" into the textfield, it will instantly (upon the "w" keystroke) start animating a change to a flow layout, the final result shown below:


Start Page
flow
Label 1
Label 2
Label 3
Label 4
Label 5
Button 1

► https://www.youtube.com/watch?v=vY60zLo6f5E *(YouTube video)*

A video clip of this transition

Implicit View Models

If you don’t specify the model class to use for your view using the `rad-model` attribute (see the *UserProfilePage* example), it will use an *implicit* view model - meaning that the annotation processor generates a view model for this view automatically. In such cases, it will generate properties on the view model to correspond *tag definitions* in the view.

In the above *transition* example, we defined a tag named "layout" using the the *define-tag* tag:

```
<define-tag name="layout"/>
```

This resulted in our view model having a property named "layout", which is assigned this "layout" tag. We then bound the text field to this property using:

```
<radTextField tag="layout"/>
```

And we referenced it in the binding expression for the *layout* parameter of the `<y>` container:

```
<y bind-layout='java:"flow".equals(getEntity().getText(layout)) ? new
FlowLayout() : BoxLayout.y()'>...</y>
```

Let’s unpack that expression a little bit:

The part that refers to our "layout" tag is:

```
getEntity().getText(layout))
```

`getEntity()` gets the view model of this view, which is an instance of our *implicit* view model. The `getText(layout)` method gets the value of the `layout` tag (which we defined above in the `<define-tag>` tag) as a string.

Supported Properties

Currently transitions don’t work with every property. Transitions are primarily useful only for properties that change the size or layout of the view. For example, currently if you add a transition to a binding on the "text" property of a label, the text itself will change *instantly*, but if the bounds of the new text is different than the old text, you will see the text bounds grow or shrink according to the transition.

Style animations are also supported on the "uiid" property, so that changes to colors, font sizes, padding etc, will transition smoothly when the *uiid* is changed. Currently style attributes (e.g.

style.fgColor) won't use transitions, but this will be added soon.

Entity Lists

So far our examples have involved only views of *single* models. Most apps involve *list* views where multiple models are rendered on a single view. E.g. In mail apps that include a list of messages, each row of the list corresponds to a distinct *message* model. CodeRAD's `<entityList>` tag provides rich support for these kinds of views.

To demonstrate this, let's create a view with an entityList. The contents of this view are as follows:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<border xsi:noNamespaceSchemaLocation="StartPage.xsd" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
    <title>Entity List Sample</title>
    <entityList layout-constraint="center"
        provider="com.example.myapp.providers.SampleListProvider.class"
    />
</border>
```

This defines a view with single entityList. The *provider* attribute specifies the class will provide data to this view. We need to implement this class *and* add a lookup to an instance of it in the controller.

The following is a sample provider implementation:

```java
package com.example.myapp.providers;

import com.codename1.rad.models.AbstractEntityListProvider;
import com.codename1.rad.models.EntityList;
import com.example.myapp.models.UserProfile;
import com.example.myapp.models.UserProfileImpl;

public class SampleListProvider extends AbstractEntityListProvider {

    @Override
    public Request getEntities(Request request) {
        EntityList out = new EntityList();
        {
            UserProfile profile = new UserProfileImpl();
            profile.setName("Steve Hannah");
            profile.setEmail("steve@example.com");
            out.add(profile);
        }
        {
            UserProfile profile = new UserProfileImpl();
            profile.setName("Shai Almog");
            profile.setEmail("shai@example.com");
            out.add(profile);
        }
        {
            UserProfile profile = new UserProfileImpl();
            profile.setName("Chen Fishbein");
            profile.setEmail("chen@example.com");
            out.add(profile);
        }
        request.complete(out);
        return request;
    }

}
```

Our provider extends `AbstractEntityListProvider` and needs to implement at least the *getEntities()* method. For most real-world use-cases you'll need to override the `createRequest()` method, but we'll reserve discussion of that for later.

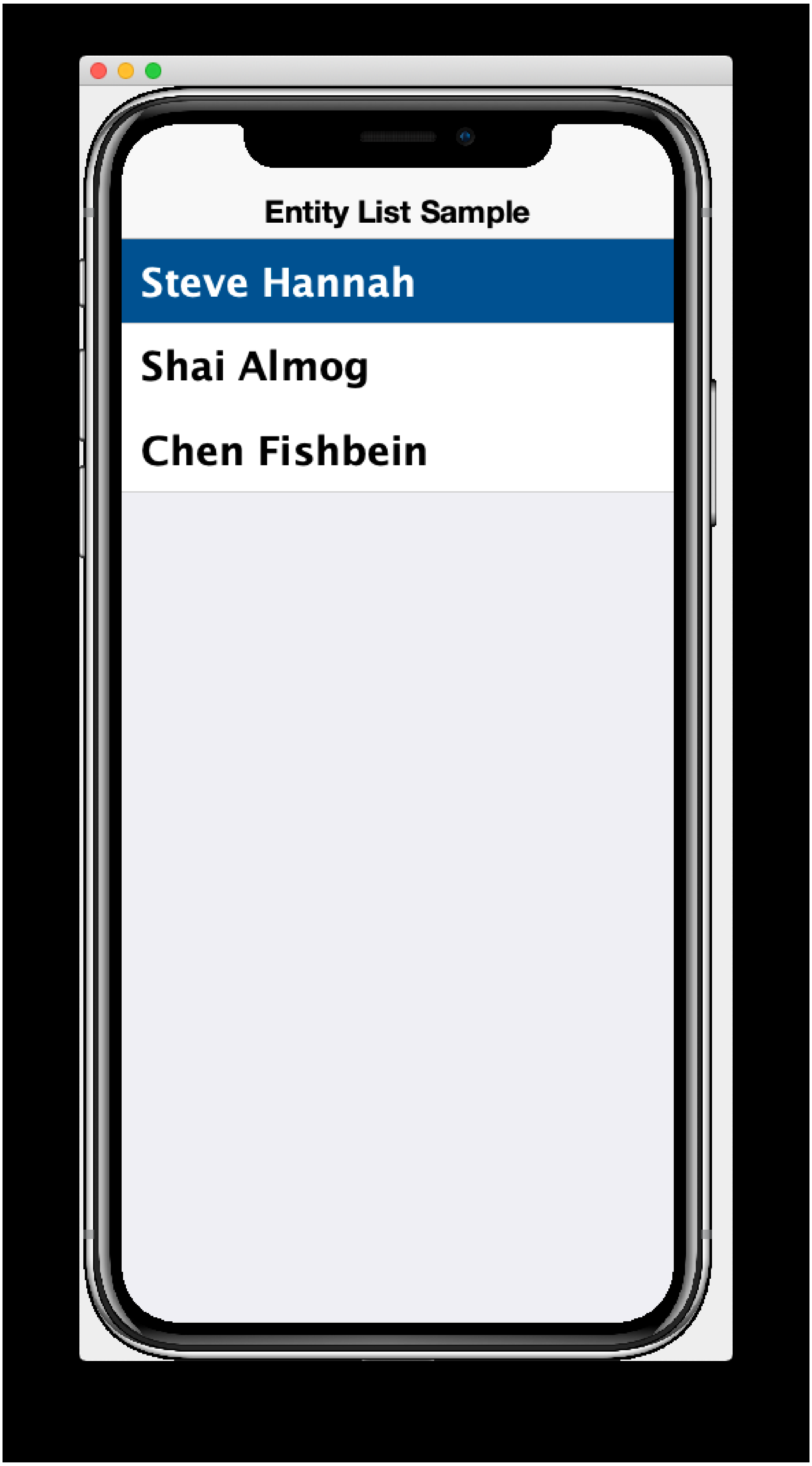
`getEntities()` is triggered whenever the entity list is requesting data. The *request* parameter may include details about which entities the list would like to receive. Out of the box, there two basic request types: *REFRESH* and *LOAD_MORE*. A *REFRESH* request is triggered when the list if first displayed, and whenever the user does a "Pull to refresh" action on the list view. A *LOAD_MORE* request is triggered when the user scrolls to the bottom of the list.

You can use the `Request.setNextRequest()` method to provide details about the current cursor position, so that the next *LOAD_MORE* request will know where to "start".

One last thing, before we fire up the simulator: We need to add a lookup to an instance of our provider. The best place to register lookups is in the `onStartController()` method of the controller. In your *MyRadApp*'s `onStartController()` method, add the following:

```
addLookup(new SampleListProvider());
```

Now, when you launch the simulator, you will see something like the following:


Entity List Sample
Steve Hannah
Shai Almog
Chen Fishbein

List Row Renderers

I'll be the first to admit that our list looks a little plain. Let's spice it up a bit by customizing its row renderer. We will tell the list view how to render the rows of the list by providing a `<row-template>` as shown below:

```
<?xml version="1.0" encoding="UTF-8" ?>
<border xsi:noNamespaceSchemaLocation="StartPage.xsd" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
    <title>Entity List Sample</title>
    <entityList layout-constraint="center"
                provider="com.example.myapp.providers.SampleListProvider.class"
    >
        <row-template>
            <border uiid="SampleListRow">
                <profileAvatar size="1.5rem" layout-constraint="west"/>
                <radLabel tag="Person.name" layout-constraint="center"
                    component.style.font="native:MainRegular 1rem"
                          component.style.marginLeft="1rem"
                />
            </border>
        </row-template>
    </entityList>
</border>
```

Let's unpack this snippet so we can see what is going on. The `<row-template>` tag directs its parent `<entityList>` tag to use its *child* container as a row template. The `<border>` tag inside the `<row-template>`, then will be duplicated for each row of the list.

Inside this `<row-template>` tag, the *context* is changed so that the *model* is the row model, rather than the model of the the parent view class. Therefore property and entity views like `<radLabel>` and `<profileAvatar>` will use the row's entity object as its model. Notice that the `<radLabel>` component is bound to the *Person.name* tag, so it will bind to the corresponding property of the row.

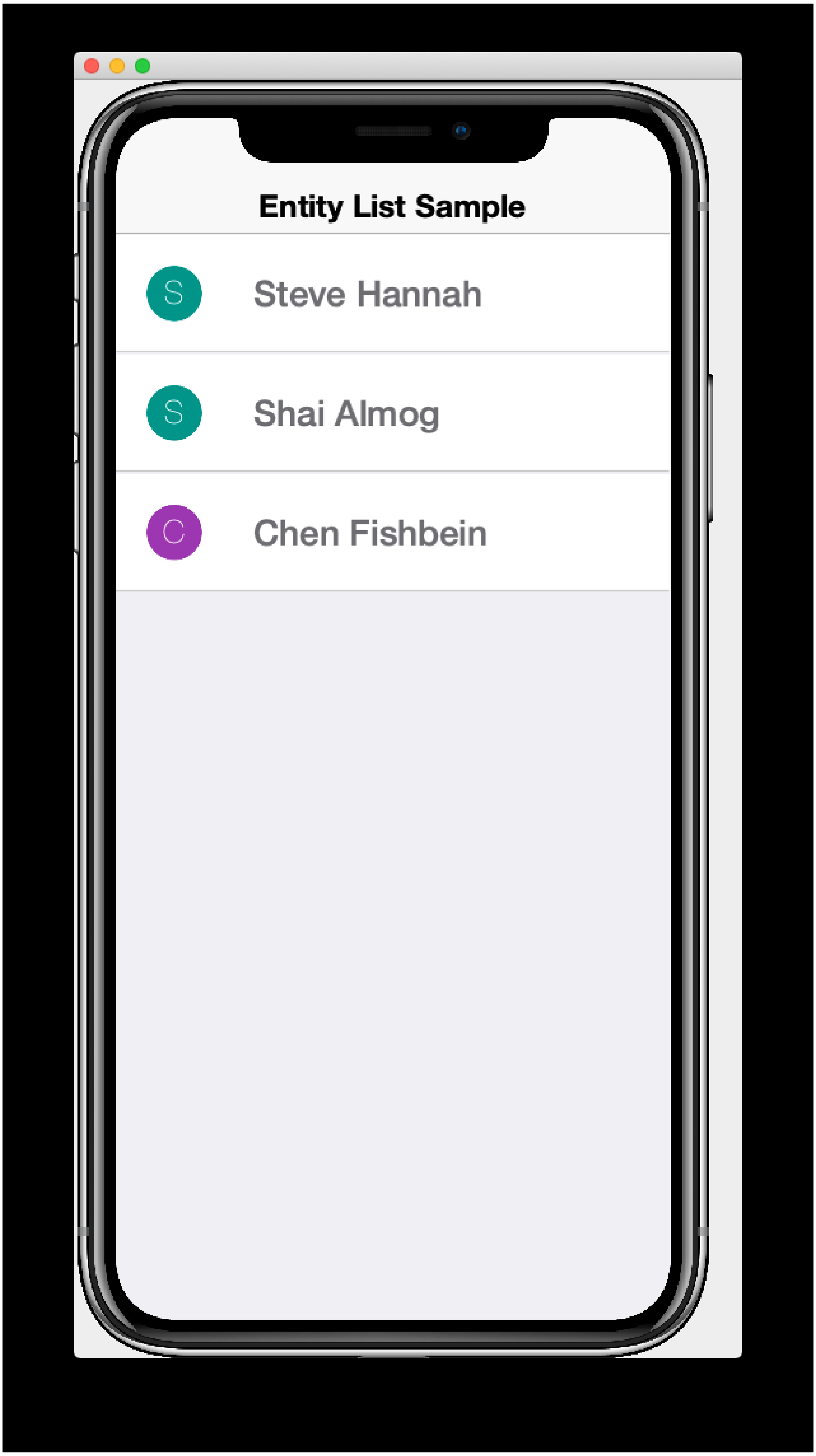
TIP This example used the `Person.name` tag whereas we could have used the `UserProfile.name` tag here. Since we defined the `UserProfile.name` tag as being equal to `Person.name` inside the `UserProfile` interface, these are equivalent. I generally prefer to reference the more generic schema tags (e.g. From the `Thing` and `Person` schemas) in my views to make them more easily portable between projects.

The `<profileAvatar>` tag is a handy component that will display an avatar for the entity. This will check to see if the entity has any properties with the `Thing.thumbnailUrl` tag, and display that image if found. Otherwise it will render an image of the first letter of the name (I.e. the value of a property with the `Thing.name` tag). For the `size` parameter we specify "1.5rem", which means that we want the avatar to be 1.5 times the height of the default font.

One last thing, before we fire up the simulator. The `<border>` tag in the row template has `uiid="SampleListRow"`, which refers to a style that needs to be defined in the CSS stylesheet. Add the following snippet to the common/src/main/css/theme.css file:

```css
SampleListRow {
    background-color: white;
    border:none;
    border-bottom: 0.5pt solid #ccc;
    padding: 0.7rem;
}
```

Now, if you start the simulator, it should show you something like the following:



Entity List Sample

S Steve Hannah

S Shai Almog

C Chen Fishbein

Responding to List Row Events

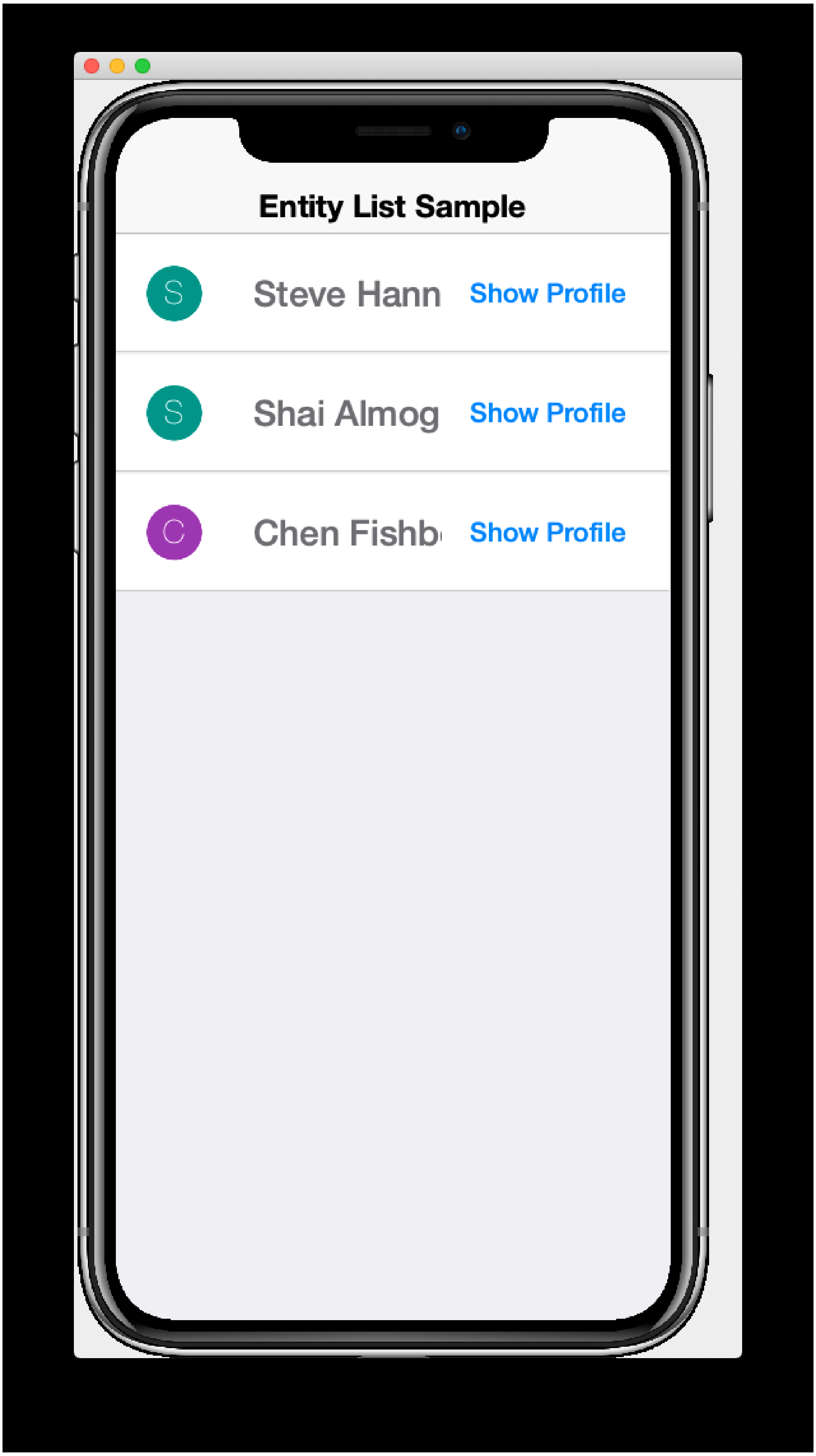
Suppose we want the app to navigate to a UserProfile form for the selected user, when the user clicks on one of the rows of the list.

The simplest way to achieve this is to add a button to our row-template as follows:

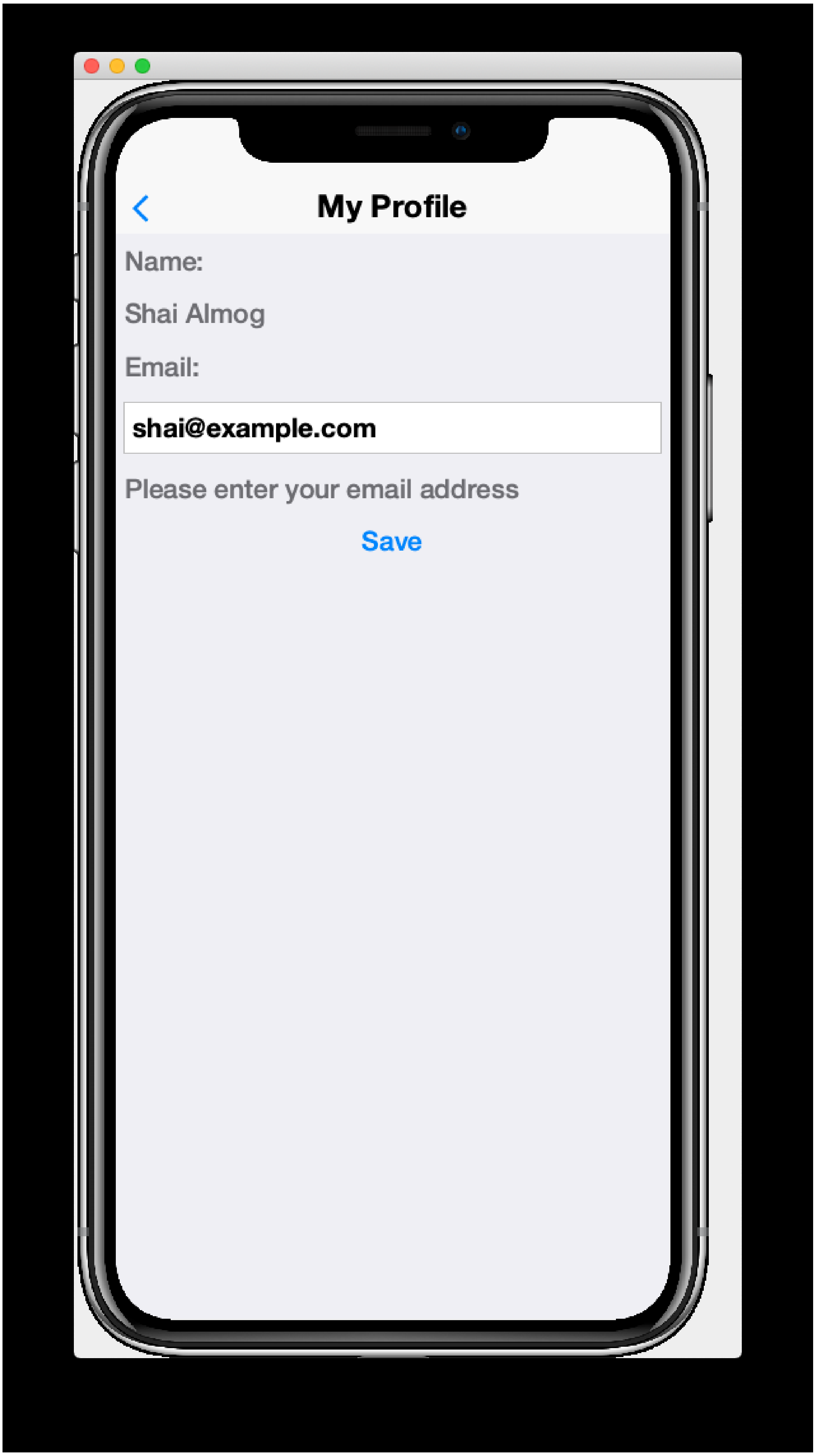
```
<button layout-constraint="east"
    rad-href="#UserProfilePage{}">Show Profile</button>
```

The `{}` at the end of the *rad-href* URL is a short-hand for the "current entity", and in this context the current entity is the one corresponding to the list row. This would be the same as `#UserProfilePage{context.getEntity()}`.

Upon saving the *StartPage.xml* file, the simulator should reload with the "Show Profile" button added to each row as shown here:


Entity List Sample
S
Steve Hann
Show Profile
S
Shai Almog
Show Profile
C
Chen Fishb
Show Profile

And clicking the *ShowProfile* button on any row, will show the *UserProfilePage* for the corresponding UserProfile. E.g. If I click on the "Shai Almog" row’s *ShowProfile* button, it will display:


My Profile
Name:
Shai Almog
Email:
shai@example.com
Please enter your email address
Save

Using a Lead Component

It seems a bit redundant to have a "Show Profile" button on each row. Why not just show the profile when the user presses anywhere on the row. This can be achieved by setting the button as the *lead component* for the row's container. Then the container will pipe all of its events to the button for handling. We would generally, then, hide the button from view.

We use the `rad-leadComponent` attribute on the container to set its lead component. This attribute takes a query selector (similar to a CSS selector) to specify one of its child components as the lead component.

Change the `<row-template>` and its contents to the following:

```
<row-template>
    <border uiid="SampleListRow" rad-leadComponent="ShowProfileButton">
        <profileAvatar size="1.5rem" layout-constraint="west"/>
        <radLabel tag="Person.name" layout-constraint="center"
            component.allStyles.font="native:MainRegular 1rem"
                  component.allStyles.marginLeft="1rem"
        />
        <button layout-constraint="east"
                hidden="true"
                uiid="ShowProfileButton"
                rad-href="#UserProfilePage{}">Show Profile</button>
    </border>
</row-template>
```

The key ingredients here are:

`rad-leadComponent="ShowProfileButton"`

This says to use the component with UIID "ShowProfileButton" as the lead component.

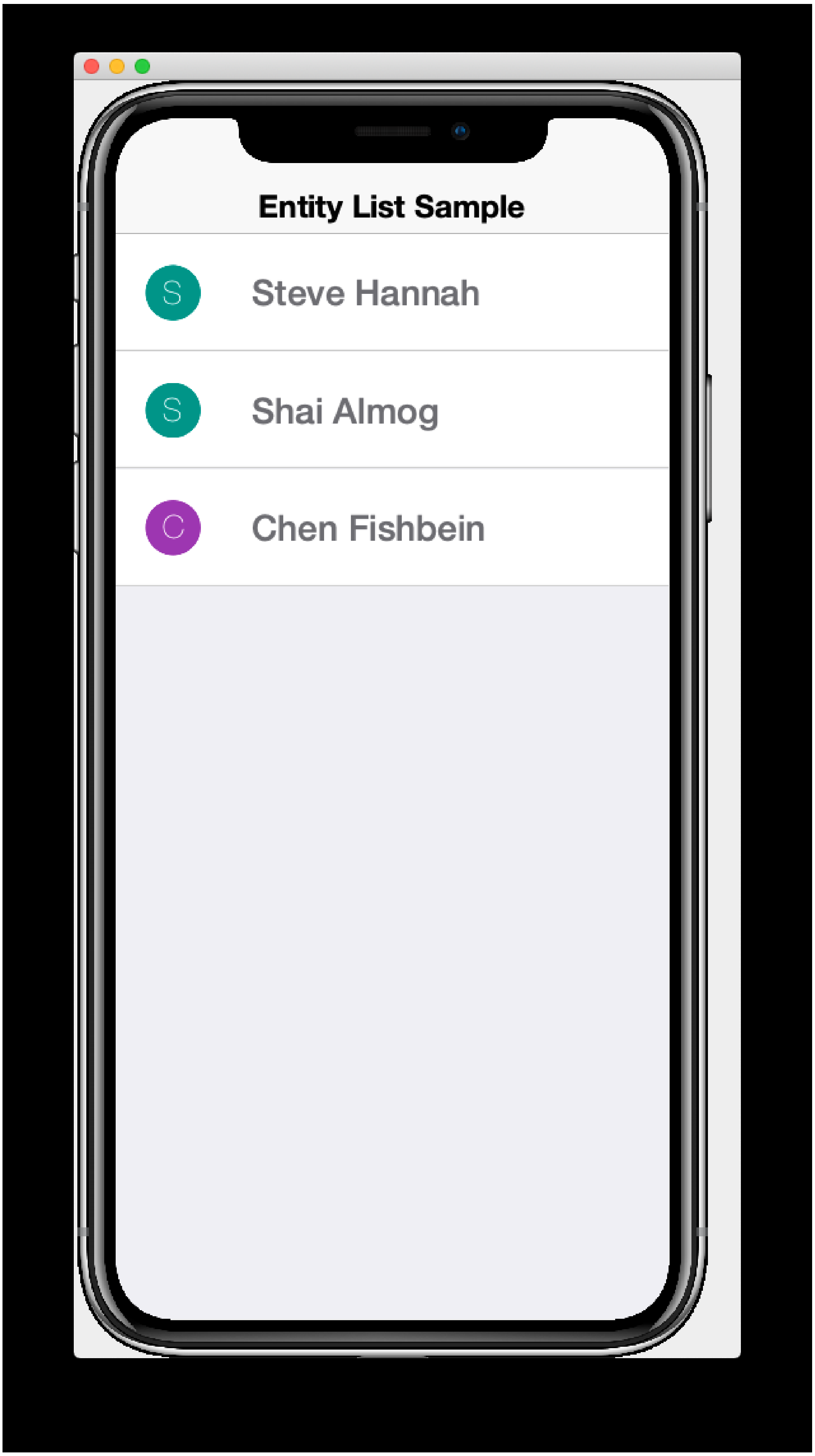
`<button … uiid="ShowProfileButton"…>`

Assign the "ShowProfileButton" uiid to the button so that the `rad-leadComponent` selector will find it correctly.

`<button … hidden="true" …>`

Set the button to be hidden so that it doesn't appear on in the view. It isn't sufficient to set `visible="false"` here, as this would still retain its space in the layout. The `hidden` attribute hides the button completely without having space reserved for it in the UI.

After making these changes, the view should look like:


Entity List Sample

Steve Hannah

Shai Almog

Chen Fishbein

And clicking on any row will trigger the `rad-href` attribute on the button, which will display the user profile for that particular row.