Developing Native Mobile Apps with CodeRAD

Steve Hannah

Table of Contents

Preface

Chapter 1. Introduction

1.1. What is Codename One?

Codename One is a toolkit for Java and Kotlin developers who want to build native apps for both iOS and Android, but don't want to have to maintain two separate codebases. It provides a cross-platform SDK for developing native mobile apps using a single codebase with 100% code reuse between iOS and Android.

Unlike Flutter, which uses Dart as programming language, Codename One apps are written in Java or Kotlin, giving you access to its well established and mature eco-system.

1.2. What is CodeRAD?

CodeRAD is a library for Codename One that facilitates rapid development of Codename One apps using established design patterns such as dependency injection and MVC (model-view-controller) . It includes foundation classes for developing models, views, and controllers, which dramatically increases code re-use and reduces application complexity. It also provides an XML-based declarative language for building views.

> *From Wikipedia:*
>
> Model–view–controller (usually known as MVC) is a software design pattern commonly used for developing user interfaces which divides the related program logic into three interconnected elements. This is done to separate internal representations of information from the ways information is presented to and accepted from the user. This kind of pattern is used for designing the layout of the page.
>
> [200px MVC Process.svg] |

NOTE

https://upload.wikimedia.org/wikipedia/commons/thumb/a/a0/MVC-

Process.svg/200px-MVC-Process.svg.png

1.3. Goals of CodeRAD

CodeRAD aims to increase developer productivity in building mobile applications by:

1. **Employing MVC (Model-View-Controller) principles** for structuring applications, leading to cleaner code that is easier to maintain.
2. **Providing reusable, high-level UI components** that can be easily integrated into applications without having to do any UI design.
3. **Providing Extensible Application Templates** that can be used as a starting point for building complete applications.

Another way of stating this is to say that CodeRAD aims provide second, and third order components to mobile app developers, as described in the following sections.

1.3.1. First-Order UI Components vs Second-Order UI Components

A first-order UI component is a basic UI component, like a label, button, or text field, that is completely application-agnostic. These form the fundamental building blocks of a GUI application, they need to be stitched together by the developer to form a coherent user interface for the user.

A second-order UI component is a complex UI component, usually composed of multiple basic components, which is designed for a specific type of application. Some examples of second-order UI components are login forms, contacts lists, chat room components, news lists, etc..

First-order components are easier to develop and reuse because they don't rely on any particular "model" or application type. But having to reinvent the wheel every time you need a common type of second-order component (e.g. a login form) can be tedious and time-consuming.

Second-order components, being more complex, are harder to develop, and even harder to re-use. Most second-order components are so application-specific that they are of no use to any app other than the one it was originally built for.

The Codename One core library includes a comprehensive set of first-order components, but very few second-order components, because of the reusability issues. There are lots of tutorials on how to build your own second-order components using the core components as building blocks, but even this can be time-consuming.

CodeRAD aims to improve the situation by providing a foundation for second-order UI components such as chat rooms, login forms, settings forms, and contact lists. The key innovation that makes this possible is its use of "loose-coupling" to allow for greater degree of reusability and customization. The goal is to develop UI kits for as many genres of applications as possible. The first proof-of-concept component was the ChatRoomView, which provides a fully-functional UI chat room component. The second project is the Tweet App UI Kit which provides high-quality UI components for building apps like Twitter.

1.3.2. Third-Order UI Components: App-in-a-Box

A third-order component is a reusable UI component that provides full-application functionality out of the box. The same principles used by CodeRAD to build second-order components could theoretically be used to produce fully-functional, yet reusable and customizable applications as turn-key components. This is beyond the initial scope of CodeRAD's aims, but after we've tamed enough of the second-order frontier, we may broaden our horizons and begin targetting entirely reusable apps. Stay tuned…

1.4. Fundamental Concepts

There are just a few fundamental concepts required to start using CodeRAD to accelerate your development.

1. MVC (Model-View-Controller) - A design pattern employed by CodeRAD which is used for developing user interfaces which divides the related program logic into three interconnected elements

   [200px MVC Process.svg] | *https://upload.wikimedia.org/wikipedia/commons/thumb/a/a0/MVC-*

Process.svg/200px-MVC-Process.svg.png

2. **Entities** - CodeRAD introduces `RADEntity`, a base class for "model" classes. This includes all the required plumbing for developing reusable components, such as property binding, property change events, data conversion, property lookup, etc...
3. **Tags** - Tags enable loose-coupling of components. Properties may contain one or more "tags" which can be used as a more generic way to reference properties on an entity.
4. **Views** - A View is a user interface component that renders a model in a specific way.
5. **Controllers** - Controllers define the structure and flow of an application. All user interaction is handled by the controller. Your application's main class will be an instance of `ApplicationController`. Each form can have an associated `FormController`. In some cases you may associate a `ViewController` with other UI components also.
6. **Actions** - Actions provide a means of extending the functionality of a view. Each view will publish a list of action categories that it supports. The controller may then register actions in these categories to embed buttons, menus, and functionality into the view.

1.5. Entities, Properties, Schemas and Tags

The **Entity** sits at the core of CodeRAD. The Entity class is the base class of all model classes in CodeRAD. Each Entity has an EntityType which defines the properties that are available in an entity. Properties, in turn, may be "tagged" with zero or more Tags. These tags can be used to lookup properties on an entity in a more generic way than referring to the property itself.

We provide a set of existing tags in the schemas package that can be used as a common foundation by both models and views. These tags were adapted from the schema definitions at https://schema.org.

> https://schema.org provides a large set of schemas for common data types that one might need in an application. It provides a base schema, Thing that includes properties that may be common to any type of "thing", such as `name`, `description`, `identifier`, `image` etc.. This schema has been ported into Java as the Thing interface.
>
> Each property has a corresponding Tag defined.

The concept of tags is a simple one, but they have powerful effect. If a view needs to render its model's "phone number" (e.g. a contact list view), then it doesn't need to know anything about the properties in the model. It is able to look up the phone number of the model by the `Person.telephone` tag:

```
String telephone = model.getText(Person.telephone);
```

As long as the model includes a property that is tagged with the `Person.telephone` tag, this will work. If the model doesn't include this property, then this will simply return null.

The following diagram depicts how Tags can be used as a sort of "glue" layer between the View and

the Model, and Action categories (discussed later under "Controllers") as a glue lasyer between the View and the Controller.

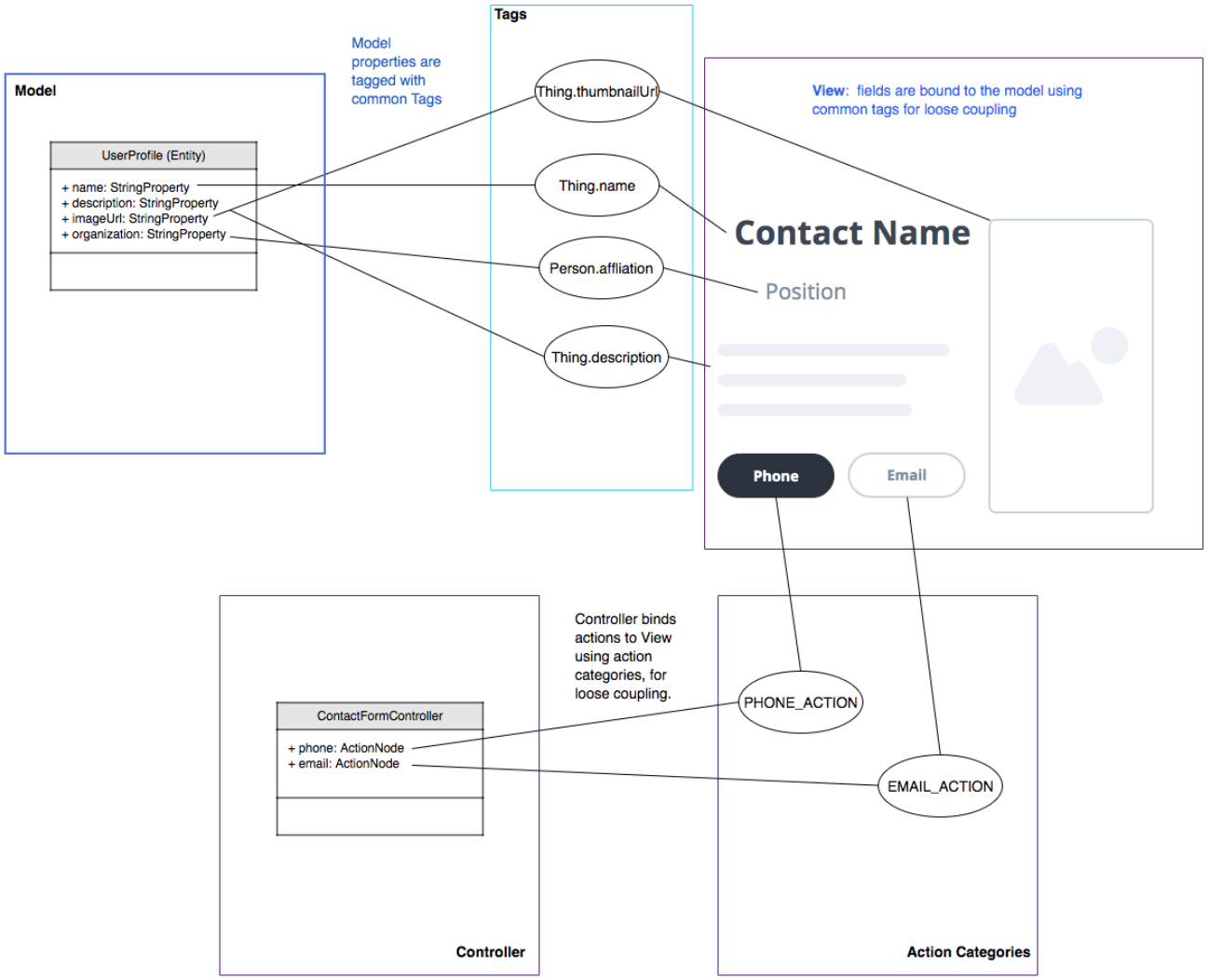


Figure 1. Tags are used to bind views to the appropriate properties of their view model using loose coupling. Action categories are used to bind views to their controllers using loose coupling.

1.5.1. Example Entity Class

The following figure shows the definition of a very simple entity class:

```
import com.codename1.rad.schemas.Person;
import com.codename1.rad.models.Entity;

@RAD ①
public interface UserProfile extends Entity <2> {
    @RAD(tag="Person.name") ③
    public String getName();
    public void setName(String name);

    @RAD(tag="Person.description")
    public String getDescription();
    public void setDescription(String description);

}
```

① The `@RAD` annotation tells the CodeRAD annotation processor to generate a concrete implementation for this class, as well as a Wrapper class.

② We extend the `Entity` interface which all model classes must implement.

③ The `@RAD(tag="Person.name")` annotation tells the CodeRAD annotation processor to bind the `getName()` method to the `Person.name` tag. This means that `myProfile.getName()` would be the same as `myProfile.getText(Person.name)`, and `myProfile.setName(name)` is the same as `myProfile.setText(Person.name, name)`.

The CodeRAD annotation processor will generate two concrete implementations of our `UserProfile` interface during compilation:

1. *UserProfileImpl* : A default implementation of the UserProfile interface that you can use for instantiating the User profile. E.g. When you want to create a new instance of `UserProfile`, you could call:

   ```
   UserProfile myProfile = new UserProfileImpl();
   ```

2. *UserProfileWrapper* : A class that can "wrap" an entity of any type to allow the `UserProfile` interface to be used to interact with it. This is handy for converting between different entity types that support the same tags. It forms part of the basis for the loose-coupling feature that makes CodeRAD such a powerful toolkit. This wrapper class includes a static `wrap()` method that takes an entity as a parameter, and returns either the same entity (if it already implements *UserProfile*), or wraps it with *UserProfileWrapper* (if it doesn't implement *UserProfile*.). You can think of this sort of like casting one Entity type to another Entity type. E.g.

   ```
   Entity someEntity = ...;
   UserProfile profile = UserProfileWrapper.wrap(someEntity);
   String name = profile.getName();
       // This calls someEntity.getText(Person.name) internally
   ```

1.5.2. Accessing Property Values

You can access property values using the "getter" methods of your entity type. E.g.

```
myProfile.getName(); // returns the profile name
```

Alternatively, you can access them using the tag that was assigned to the property. e.g.

```
myProfile.getText(Person.name); // Returns the profile name
```

The two examples above are equivalent for our *UserProfile* entity because we assigned the `Person.name` tag to the *name* property of our entity.

TIP

Which method you use will depend on the context. Use the first form if you are working with a *UserProfile* object directly. Use the 2nd form if you are working with an *Entity* of unknown type and you want to retrieve or set its `Person.name` property. This is especially handy for developing reusable UI components that expect models to include certain "tags". For example, a "Contact Card" component probably expects a `Person.name`, `Person.email`, `Person.phone`, etc.. property, but it doesn't care exactly what type of model it is. That way it will work with any Entity class that it is given. All the Entity class needs to do is include properties corresponding to those tags. It goes without saying that such a view would need to be able to handle the case where the entity doesn't inclode one or more of those tags.

1.6. Views

The "View" is the piece of the MVC pie that we are most interested in sharing and reusing. A View is a Component that includes support to "bind" to a view model (an Entity), such that when properties on the view model are changed, the "View" updates to reflect the change.

The recommended way to develop views is using RADL, Code RAD's declarative markup language for creating visual components. RADL files are converted to Java view classes at compile time by CodeRAD's annotation processor.

The following is a basic RADL file that simply displays "Hello World"

```
<?xml version="1.0"?>
<y>
    <label>Hello World</label>
</y>
```

NOTE

You can also create views directly using Java or Kotlin by extending either `AbstractEntityView` or `EntityListView`. However, using RADL is much easier, and provides many other benefits, such as dependency injection to make like more enjoyable.

Views should be placed inside the *common* module in the *src/main/rad/views* directory, using Java package hierarchy conventions. E.g. If want your view to compile to the class `com.example.MyView`, then you would create your view file at *src/main/rad/views/com/example/MyView.xml*.

A single RADL file will generate multiple Java classes. For a given view file *MyView.xml*, it will generate the following classes:

1. *MyView* - The view class.
2. *MyViewSchema* - An interface with *Tag*, and *Category* definitions that are declared in the view.
3. *MyViewModel* - An an *Entity* interface representing the view model of for the view.
4. *MyViewModelImpl* - A default implementation of *MyViewModel*.
5. *MyViewModelWrapper* - Wrapper class for *MyViewModel*.
6. *MyViewController* - A default *FormController* implementation for for using *MyView*.
7. *IMyViewController* - A marker interface that you can use to mark controllers as compatible with *MyViewController*.

1.7. Controllers and Actions

Controllers serve two functions in CodeRAD:

1. **Separation of Concerns** - Controllers handle all of the "application logic" as it pertains to the user's interaction with the app. Keeping application logic separate from the view and the model has many advantages, including, but not limited to, easier code reuse.
2. **Application Structure & Control Flow** - Controllers provide hierarchical structure for applications similar to the way that Components provide hierarchical structure for user interfaces. While it possible to use CodeRAD components in isolation, (without a controller hierarchy), you would be missing out on some of CodeRAD's best features.

1.7.1. The "Navigation Hierarchy"

It is useful to think of your app's controllers through the lens of a "navigation hierarchy". The "root" node of this navigation hierarchy is the ApplicationController. To show the first form in our app, we create a `FormController`, which can be views as a "Child controller" of the application controller. If the user clicks a button that takes them to a new form, we create a new FormController, which is a child of the previous form controller.

CodeRAD's FormController class includes built-in logic for "back" navigation. If the FormController's parent controller is, itself, a FormController, then it will provide a "Back" button (and link up the Android "back" action) to return to the parent controller's form.

Typical code for creating a FormController is:

Typical code to create and show a FormController. This code is assumed to be in another FormController, so `this` refers to the current controller, passing it as the first parameter sets it as the `detailsController's parent.

```
DetailsFormController detailsController = new DetailsFormController(this, model);
detailsController.show();
```

1.7.2. Event Propagation

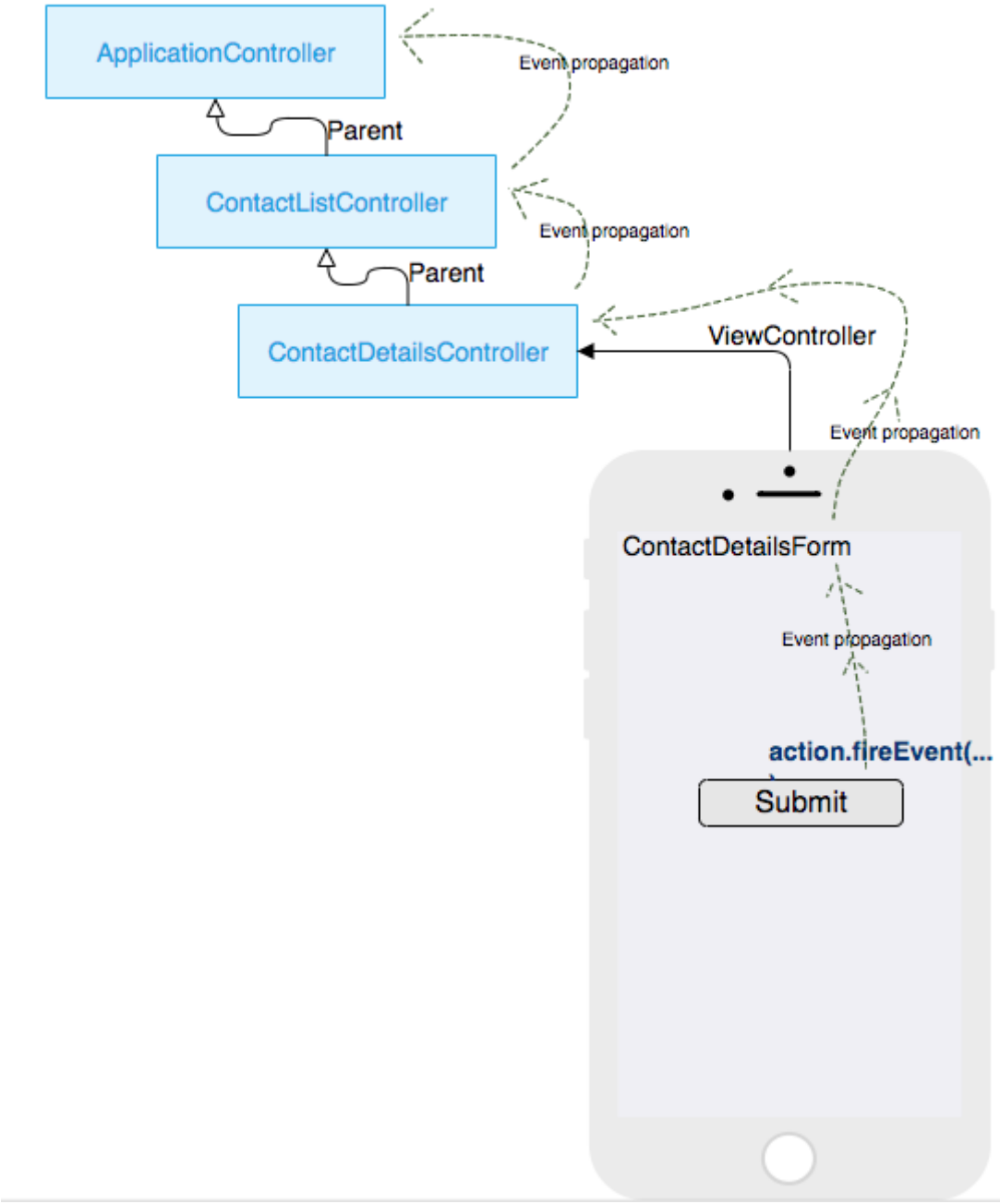
The hierarchical view of controllers is also useful for understanding event dispatch. When a ControllerEvent is fired on a UI component, it will propagate up the UI hierarchy (i.e. Component → parent ..parent...) until it finds a component with a ViewController. The event will then be dispatched up the controller hierarchy until it is consumed.

For example, suppose, in our application, we have the following controller hierarchy:

1. **Root Controller** - The ApplicationController
    a. **ContactListController** - The main form of the app: A contact list.
        i. **ContactDetailsController** - The user clicked on a contact in the list, so they navigated to the "Details" form for that contact. Thus the *ContactDetailsController* is a "child" of the *ContactListController*.

The following diagram depicts this hierarchy. Suppose that there is a button on the contact details form, that the user clicks to initiate an action event. Then the event will propagate up the UI hierarchy until it finds a component with a ViewController. In this case, the "Detail" form is the first component with a ViewController: The *ContactDetailsController*. If the *ContactDetailsController* contains a handler for the action that was fired, then it will process the event. If the event is still not consumed, it will propagate up to the parent (the *ContactListController*), and give it an opportunity to handle the event. If it is still not consumed, it will propagate up to the root controller (the *ApplicationController*).

This image depicts the propagation of an action event up the UI hierarchy and then the controller hierarchy.



The fact that action events propagate up through the controller hierarchy gives you flexibility on where you want to place your application logic for processing events. This is very handy in cases where you want to handle the same action in two different controllers.

For example, suppose you have a "phone" action that allows you to phone a contact. The *ContactListController* may support direct dialing of a contact in the list. Additionally, you probably have a "Phone" button on the contact details form. Since the *ContactDetailsController* is a "child" controller of the `ContactListController`, you can handle the action once inside the *ContactListController*, rather than duplicating code on both the list and details controllers.

Chapter 2. 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.

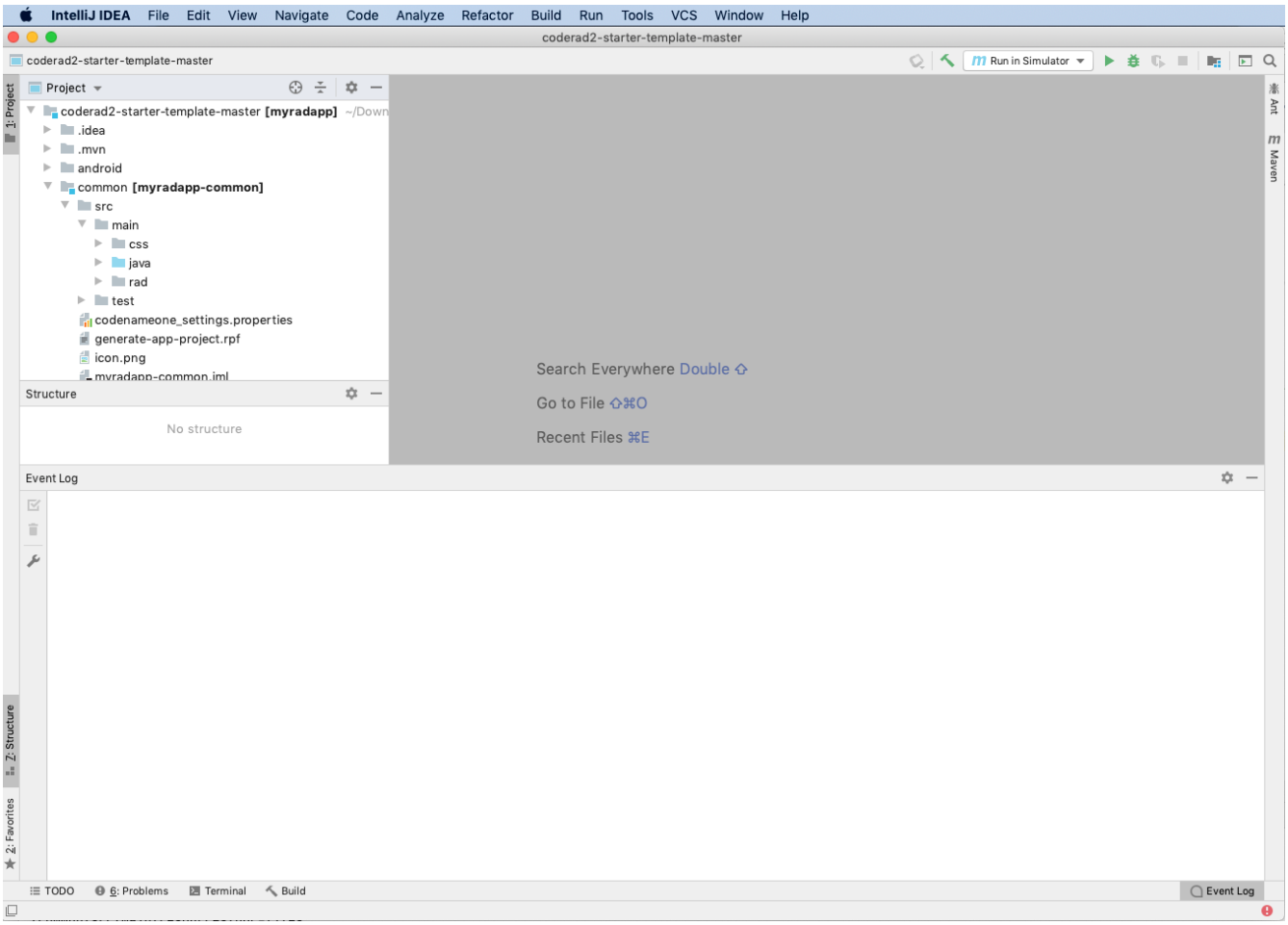


Figure 2. The Code RAD starter project opened in IntelliJ IDEA

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.

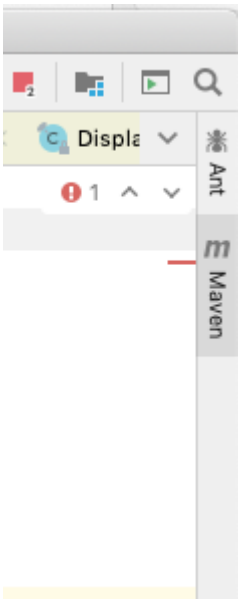
Figure 3. The starter project running inside the Codename One simulator

2.1. Customizing the Maven Artifact Coordinates

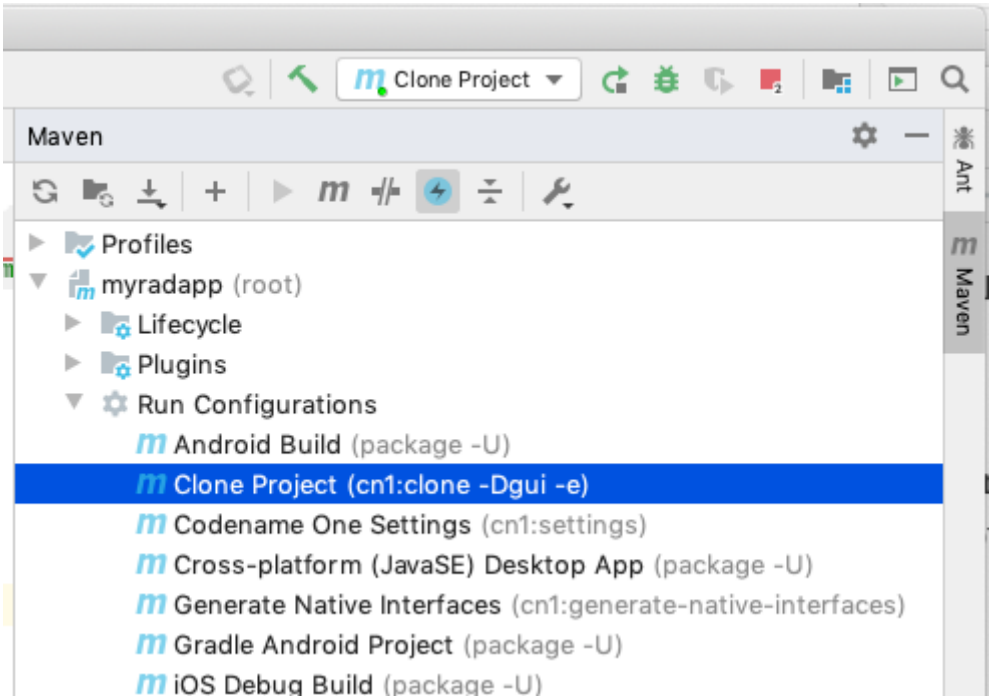
The starter project comes preconfigured with placeholder coordinates for `groupId` and `artifactId`. For the purposes of this tutorial, I will be using these default coordinates. However, for a *real* app, you will want to change these coordinates to your own `groupId` and `artifactId`. The easiest way to change these coordinates is using the `cn1:clone` maven goal.

WARNING	If you clone the project and change the `groupId` and `artifactId`, this will also change package structure in the project, so your package structure will be different than the one I use in the rest of this tutorial. E.g. Instead of your project files being in the *com.example.myapp* package, they will be in a package named after your *groupId*.

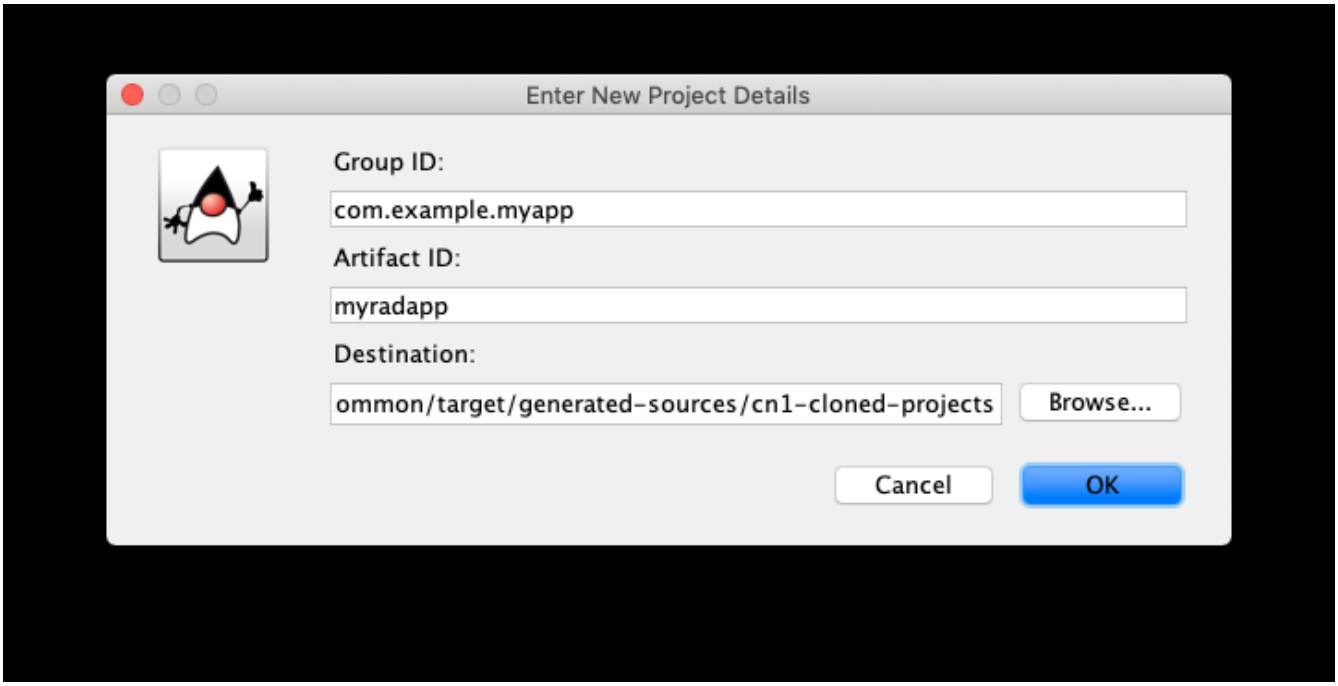
You can run this goal in IntelliJ by first expanding the "Maven" tab from the tabs in the right side of the window:



Once the *Maven* sidebar is opened, expand the *myapp* node, and the *Run Configurations* sub-node, and then double-click the *Clone Project* option shown here:



It should display a dialog prompt for you to enter the new coordinates and location for your Maven project:

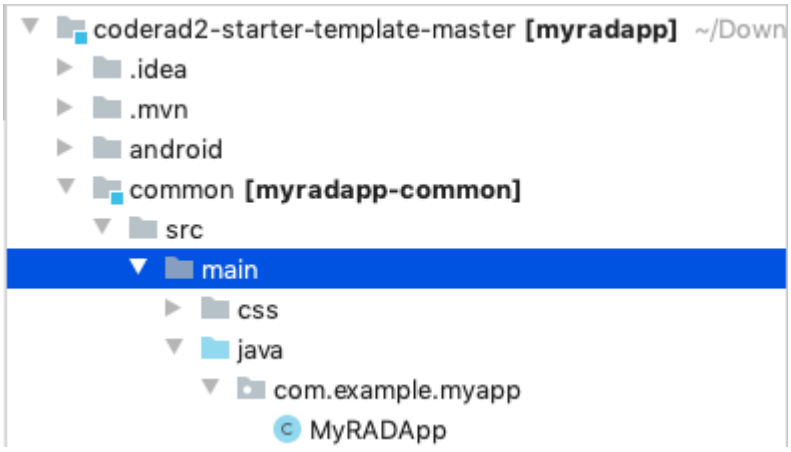


After entering the project details into the dialog, press *OK*. If all goes well, you should see a *BUILD SUCCESS* message in the output log of IntelliJ, and you should find your new project at the location that you specified. You can then close this project, and open your newly created project.

2.2. 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.



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.

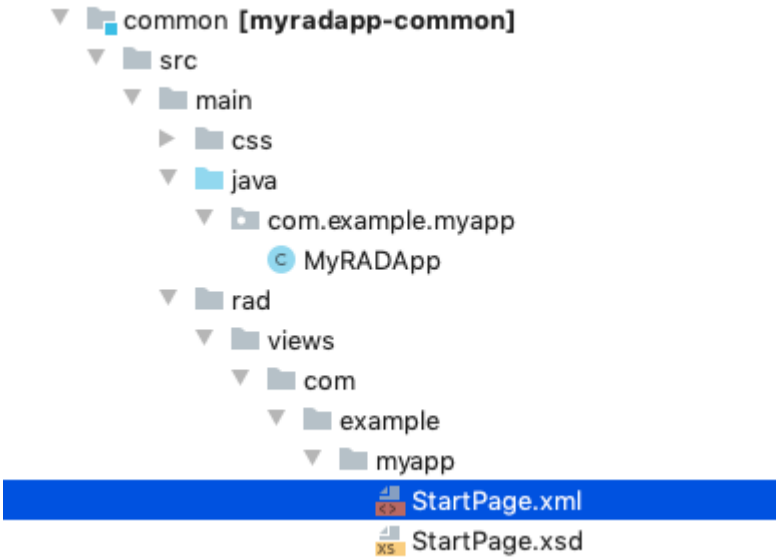
The effective action performed by the event handler is the line:

```
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.



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).

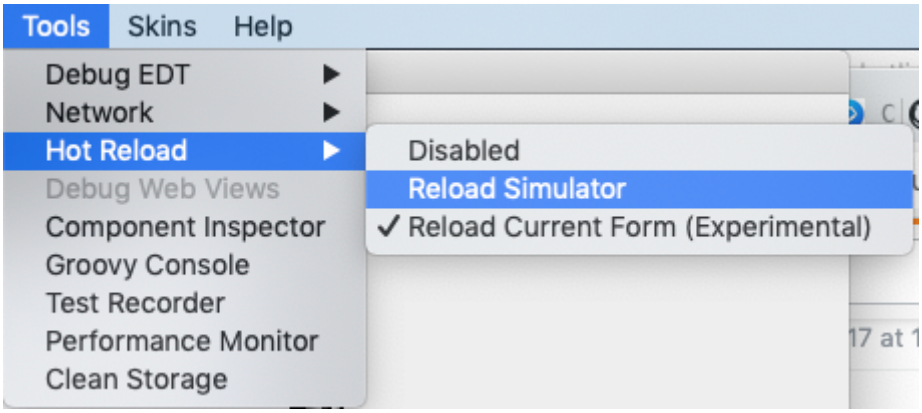
`<title>Hi World</title>`
: Not a child component, rather a "bean" that sets the title of the form to "Hi World"

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

2.3. 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* menu.



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.

The *Reload Current Form* option is the same as the *Reload Simulator* option except that it will automatically load the current form when the simulator reloads. When using this option, you will lose the navigation context (e.g. the *parent* controller will be the *ApplicationController*) and data, when the simulator reloads.

TIP	Technically these *hot reload* options aren'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

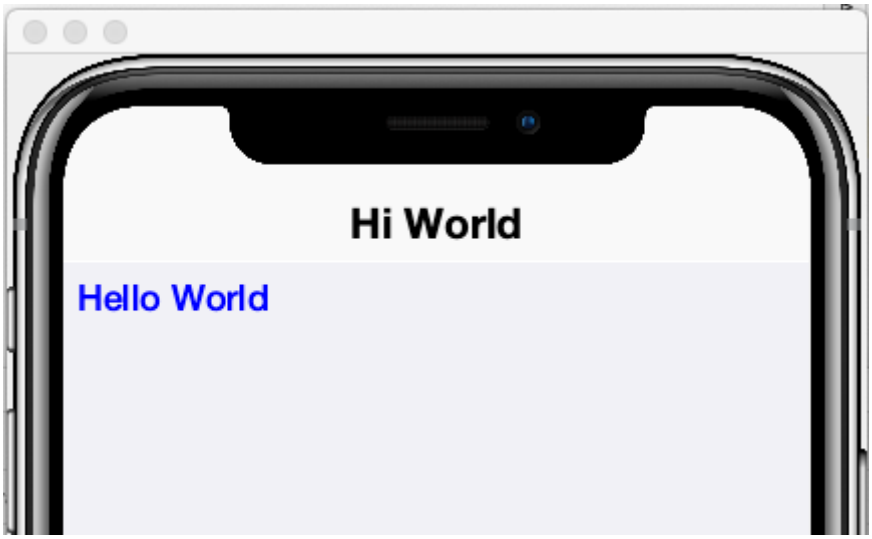
2.4. Changing the Styles

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;
}
```

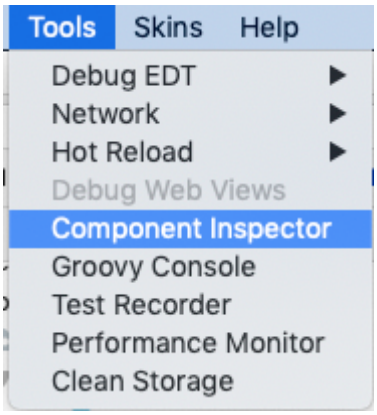
Within a second or two after you save the file, you should notice that the "Hello World" label in the simulator has turned blue.



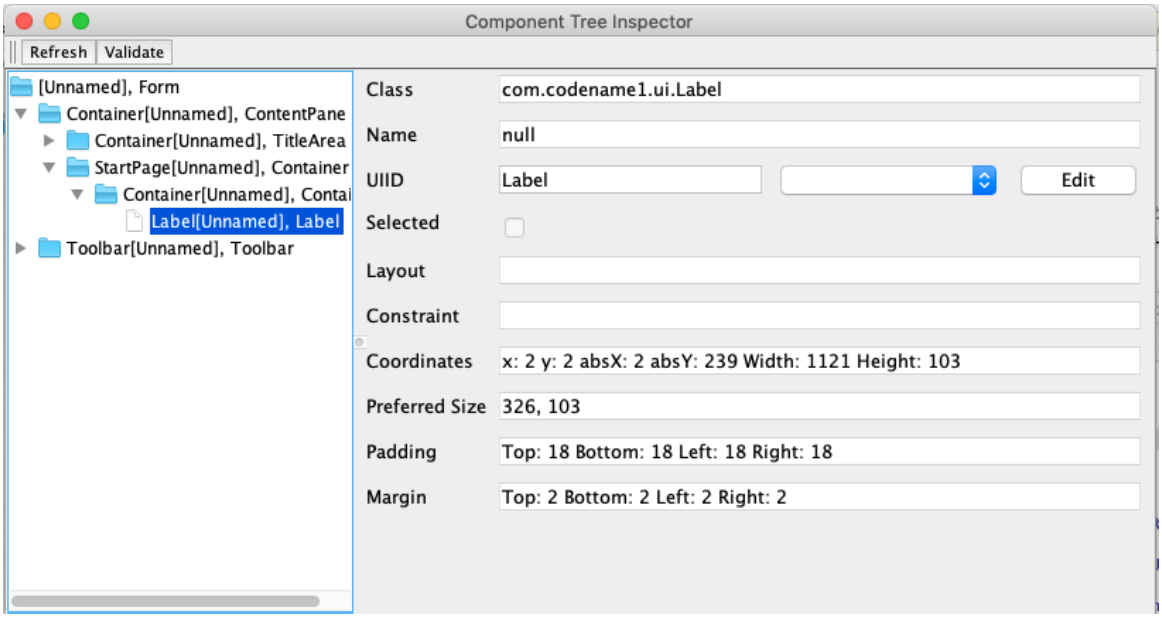
This is because the Label component's default UIID is "Label", so it adopts styles defined for the selector "Label" in the stylesheet.

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.



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.

2.4.1. Inline Styles

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

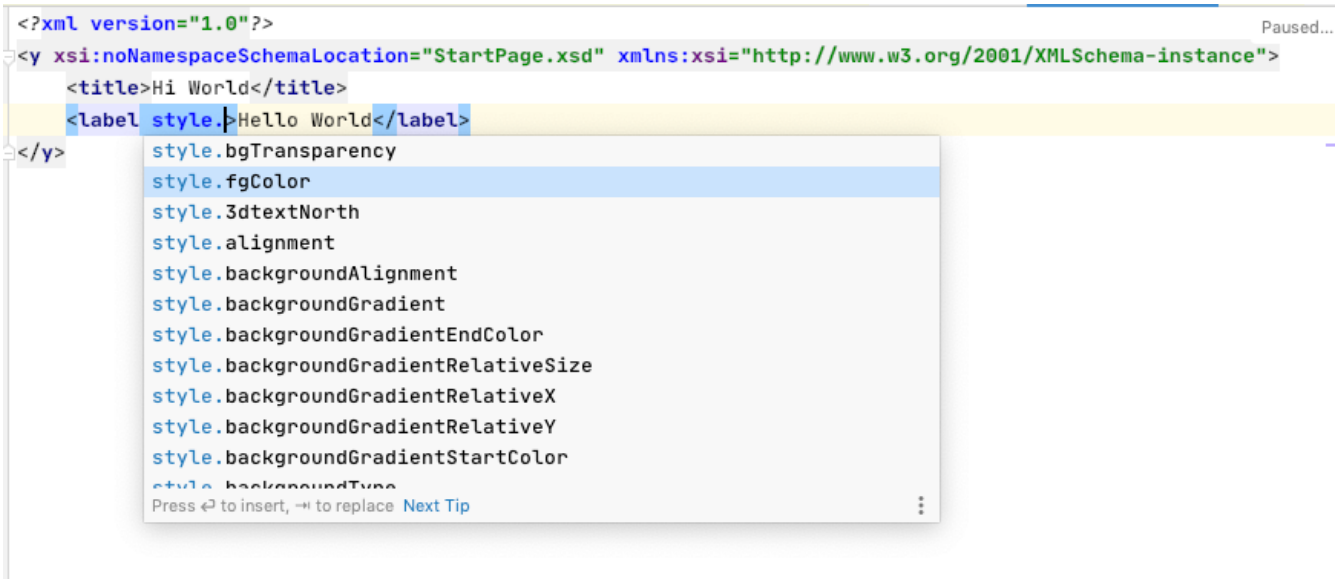


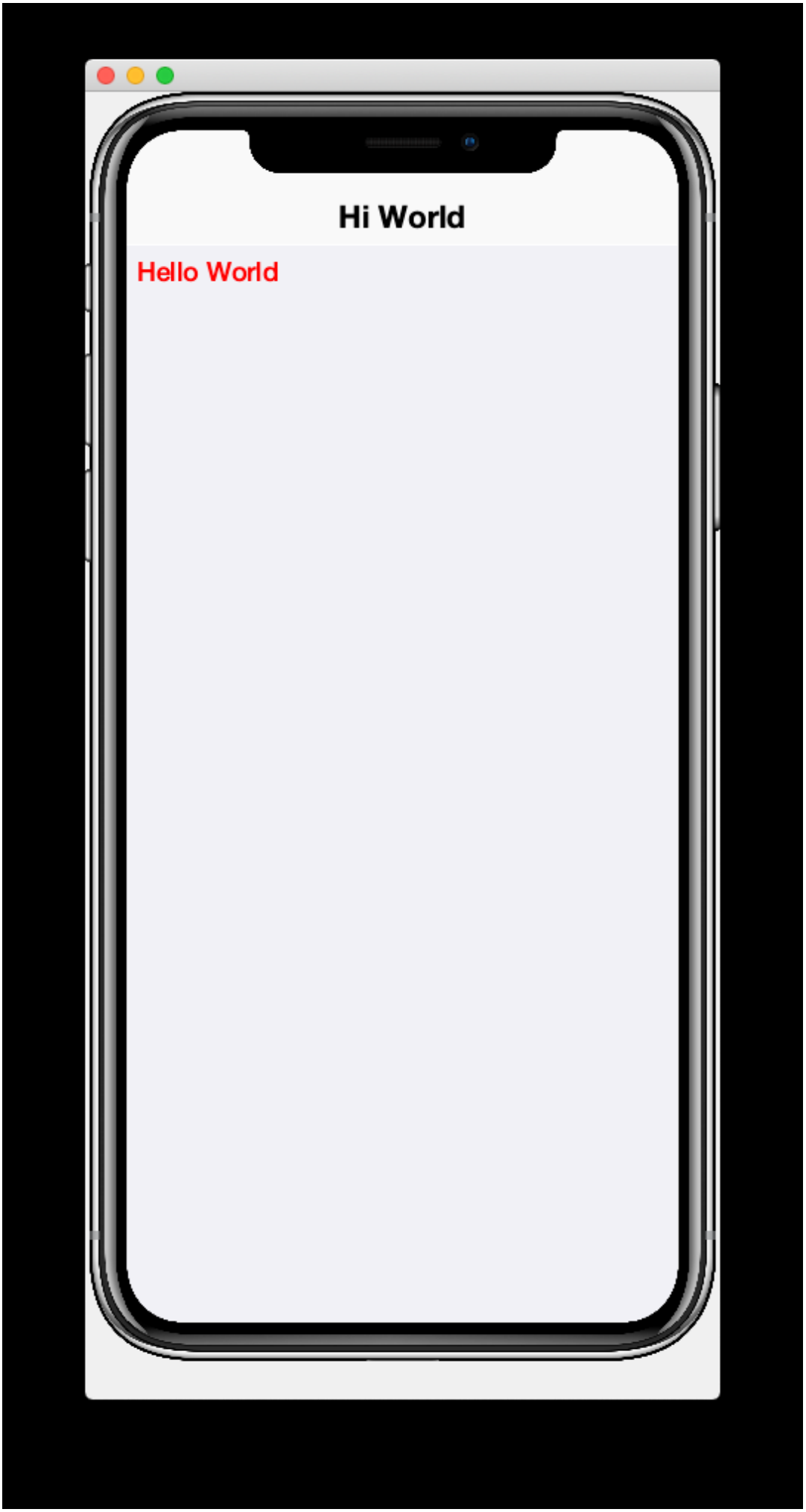
Figure 4. In IntelliJ's XML editor, you'll receive type hints for all tags and attributes as shown here.

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.

2.4.2. 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.

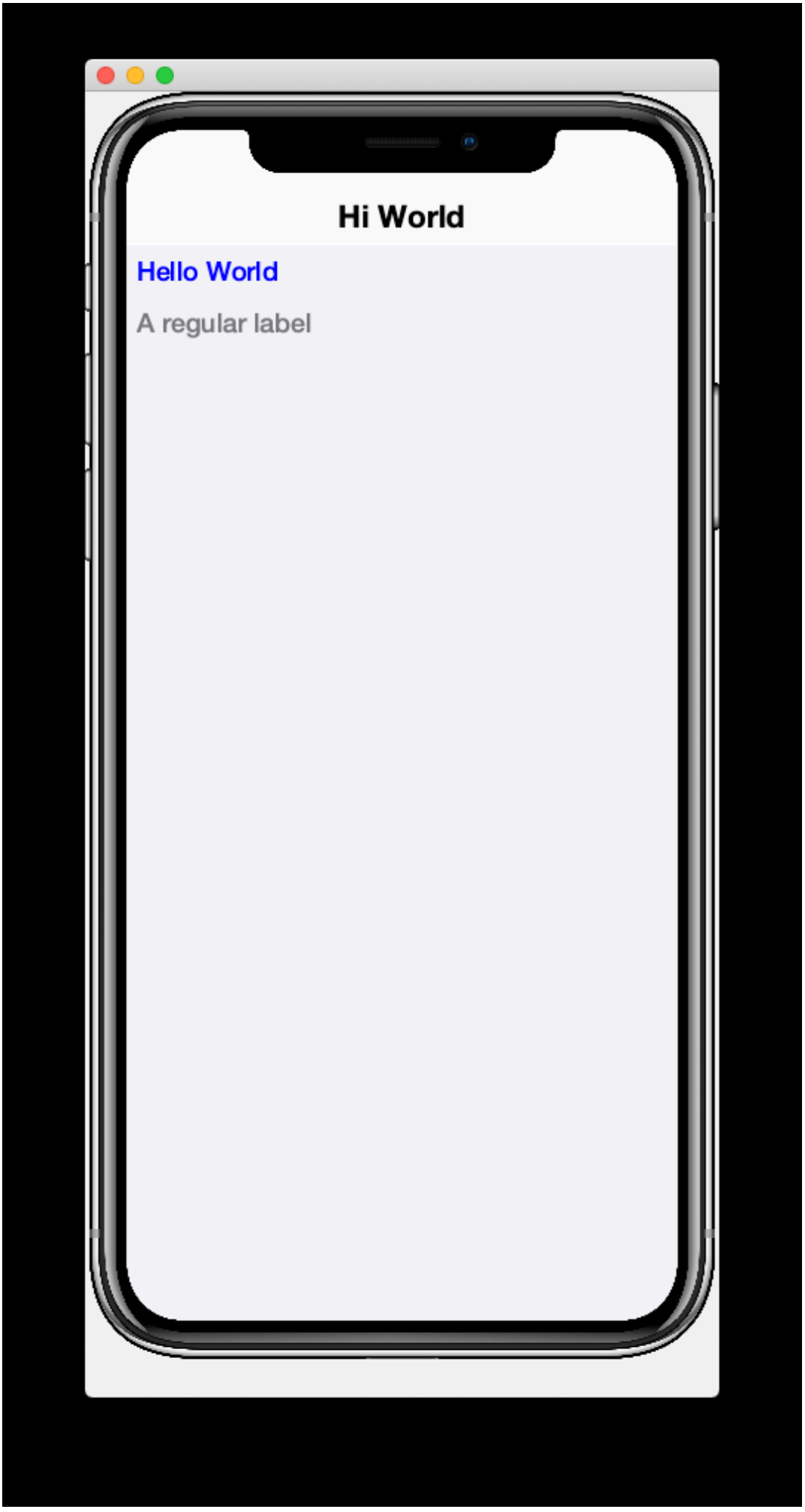
Now return to the *StartPage.xml* file and add `uiid="CustomLabel"` to the `<label>` tag. While we're at it, remove the inline `style.fgColor` attribute:

```
<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

2.5. 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 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;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.
Jun
2021
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
A checkbox
A Radio Button
Click Me

2.6. 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.

2.6.1. 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:

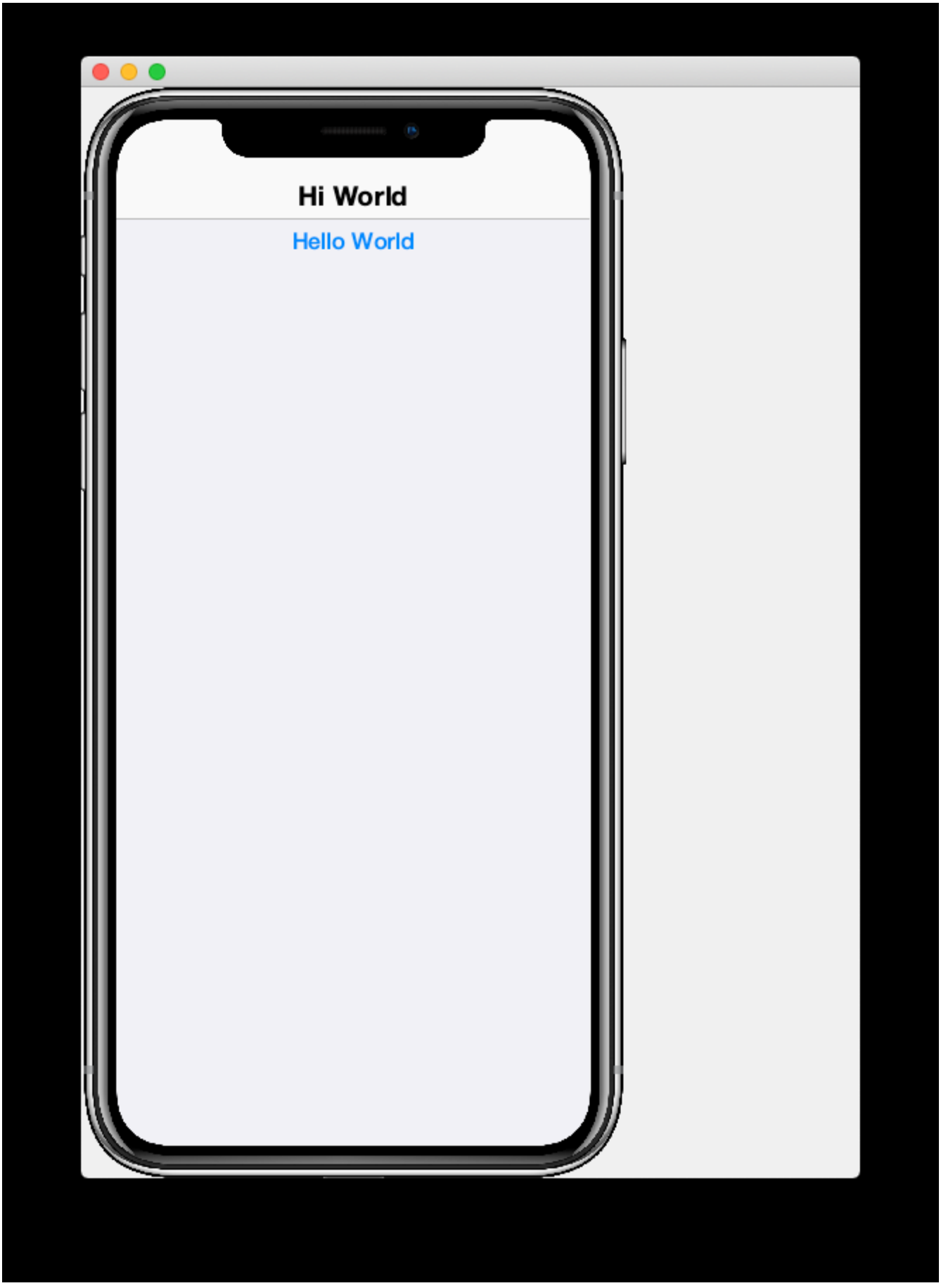
```
@Override
protected void initControllerActions() {
    super.initControllerActions();
    addAction(StartPage.HELLO_CLICKED, evt-> {
        evt.consume();
        Dialog.show("Hello", "You clicked me", "OK", null);
    });
}
```

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:

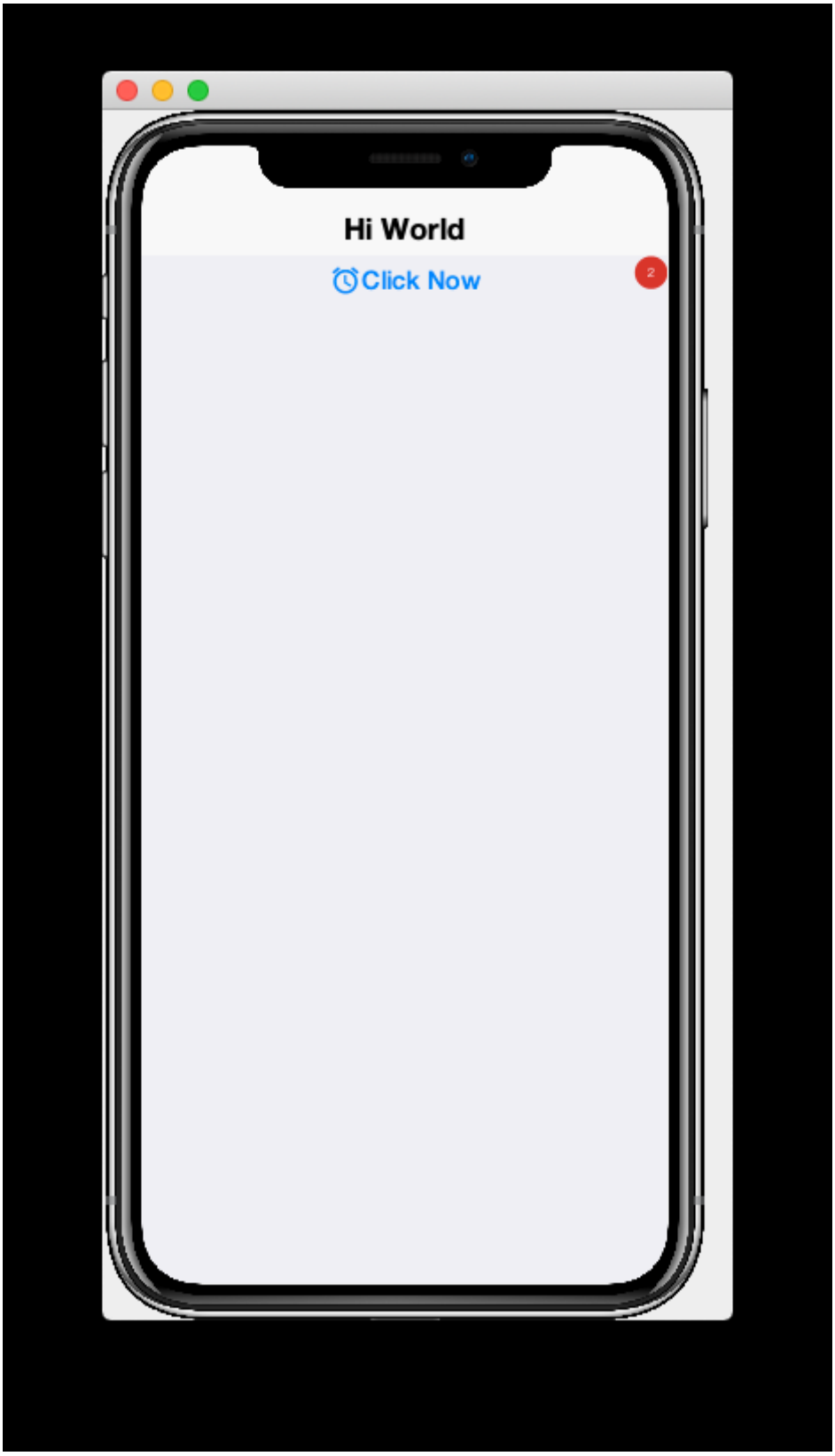
2.6.2. Customizing Action Rendering

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.

2.7. 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:

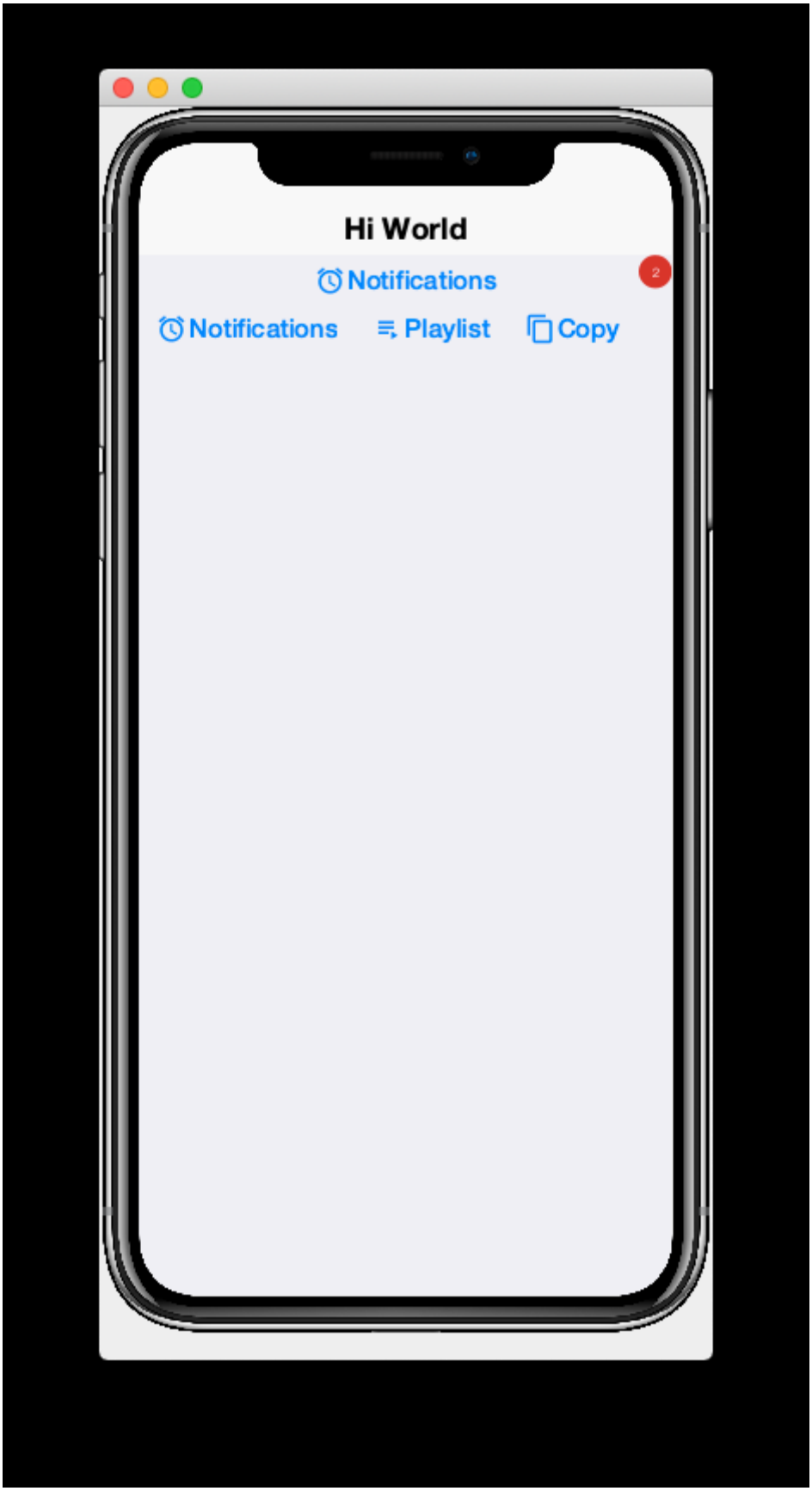
1. Define a new category called `MAIN_MENU`.
2. Add a `<buttons>` component to our view with `actionCategory="MAIN_MENU"`.
3. Define some actions in the controller, and register them with the new `MAIN_MENU` category.

```
<?xml 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:

```
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:



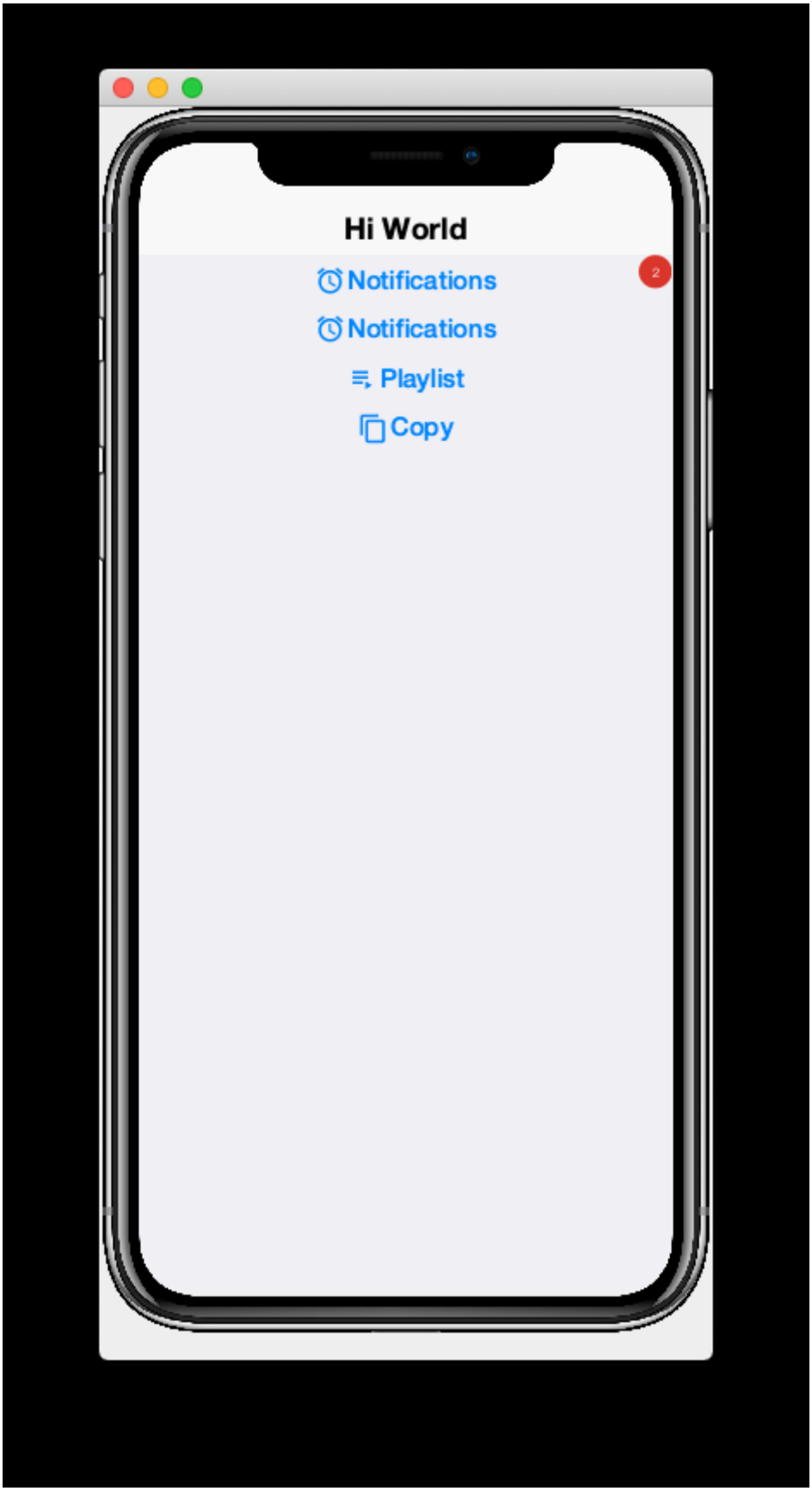
2.7.1. Buttons Layout

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"/>
```

2.7.2. 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:

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:

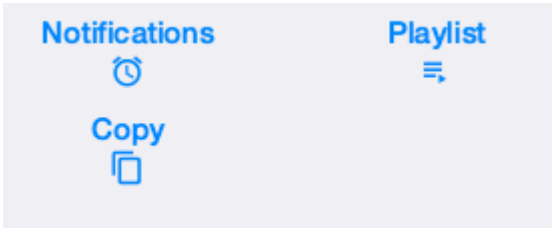


Figure 5. IconBottom

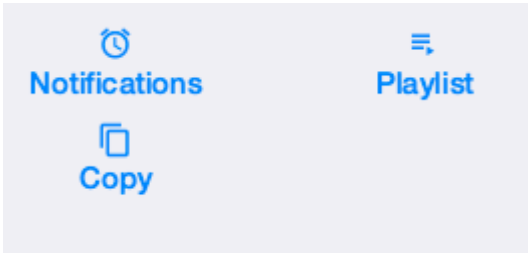


Figure 6. IconTop

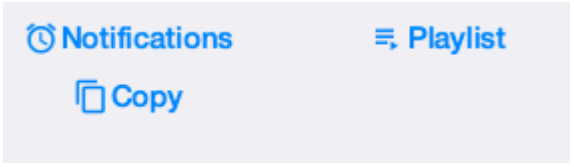


Figure 7. IconLeft

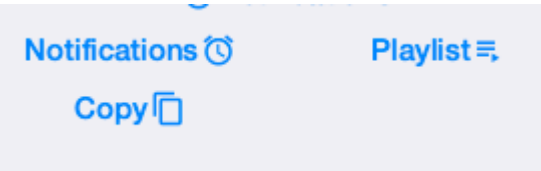


Figure 8. IconRight

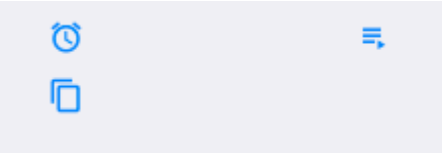


Figure 9. IconOnly

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).

2.7.3. Overflow Menus

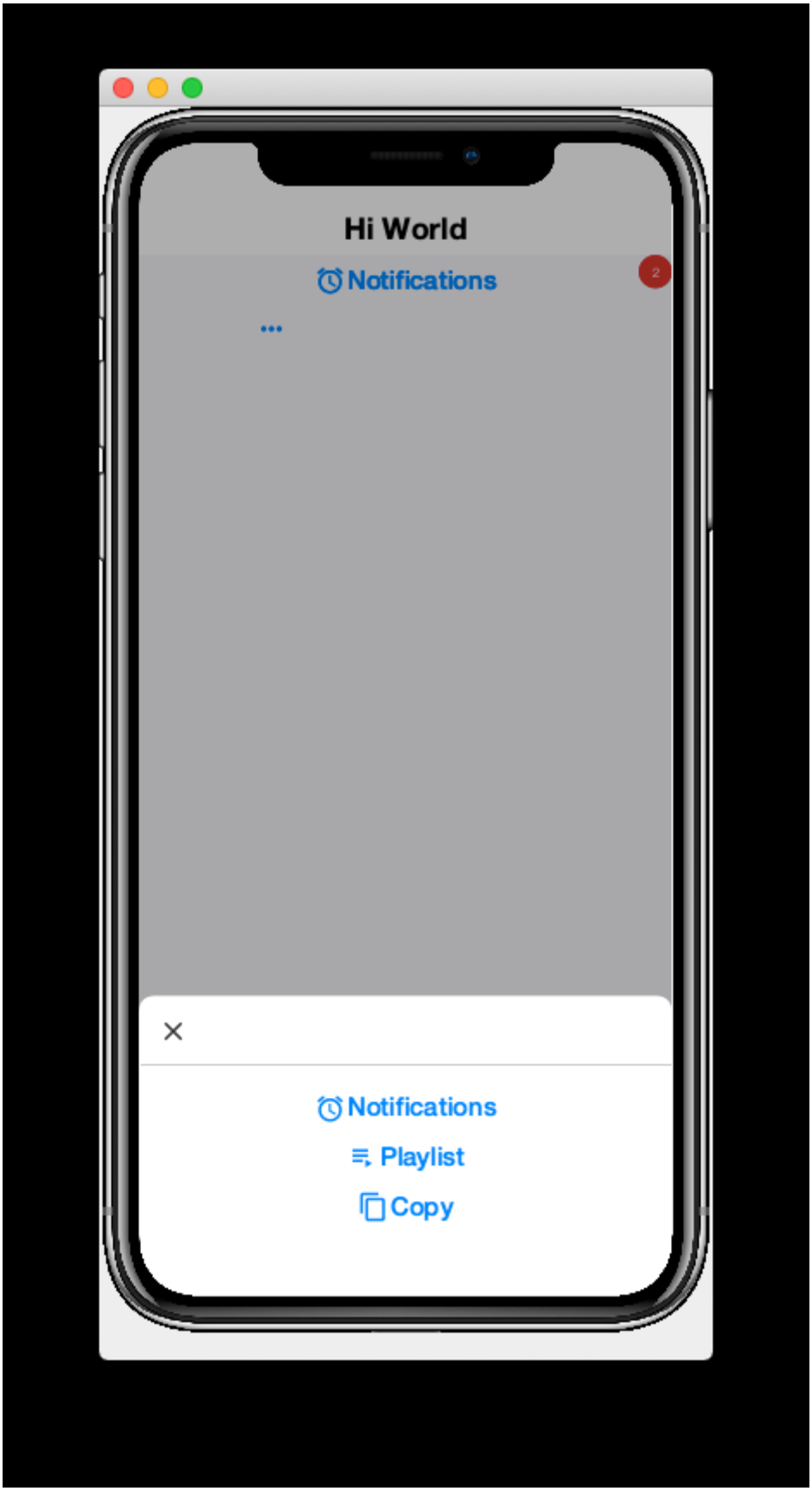
In some cases, your view may only have room for one or two buttons in the space provided, but you want to be able to support more actions than that. You can use the *limit* attribute to specify the maximum number of buttons to render. If the number of actions in the action category is greater than this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

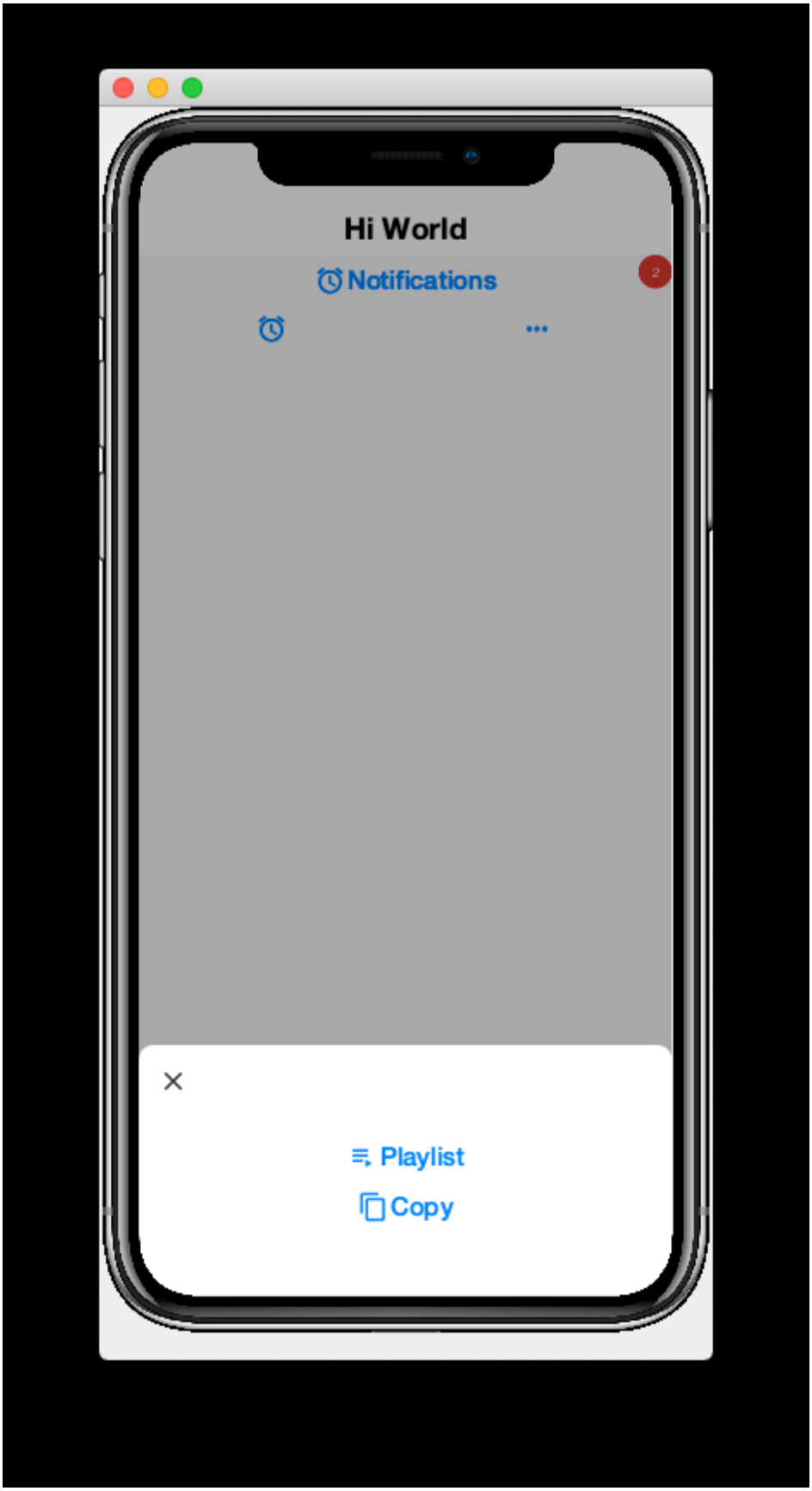
```
<buttons
        layout="new GridLayout(2,2)"
        actionCategory="MAIN_MENU"
        actionTemplate.actionStyle="IconOnly"
        limit="1"
/>
```

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.



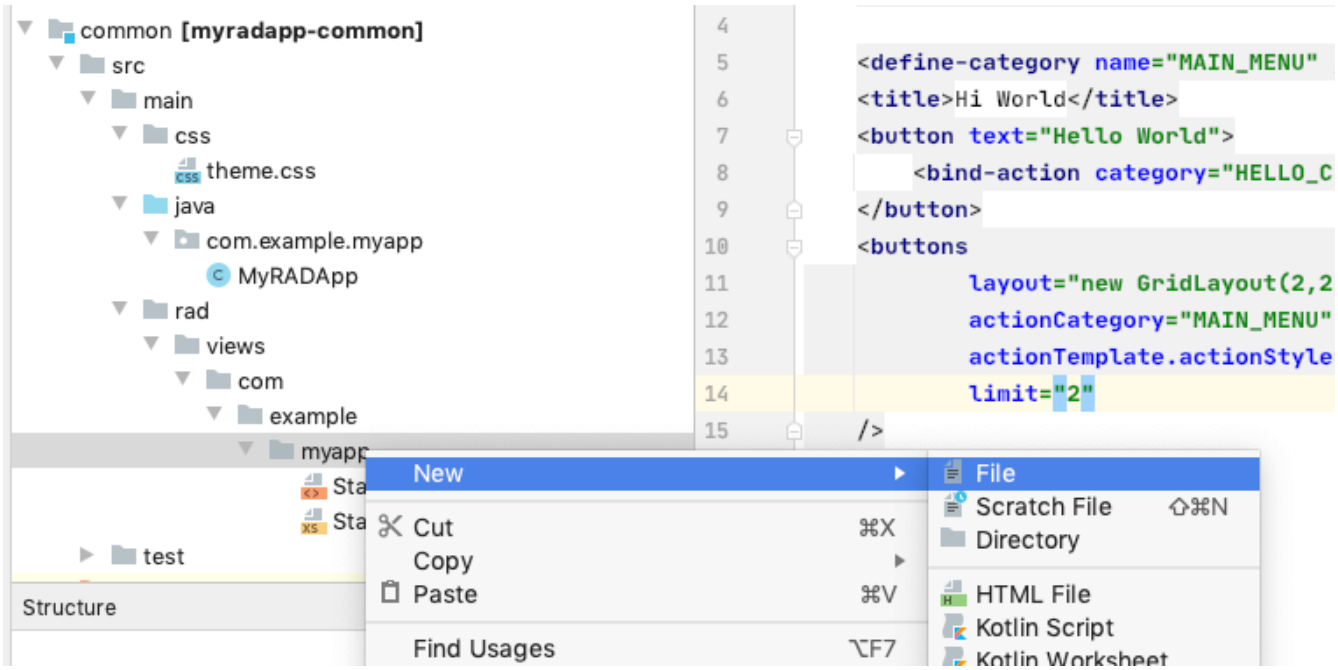
If you change the limit to "2", it will show the first action, *Notifications*, in the buttons, and then it will show the remaining two actions when the user presses the "More" button.



2.8. 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:



Then enter "AboutPage.xml" in the dialog:

And press *Enter*

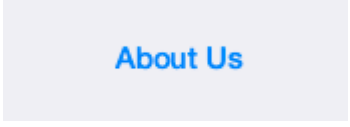
Add the following placeholder contents to the newly created *AboutPage.xml* file:

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

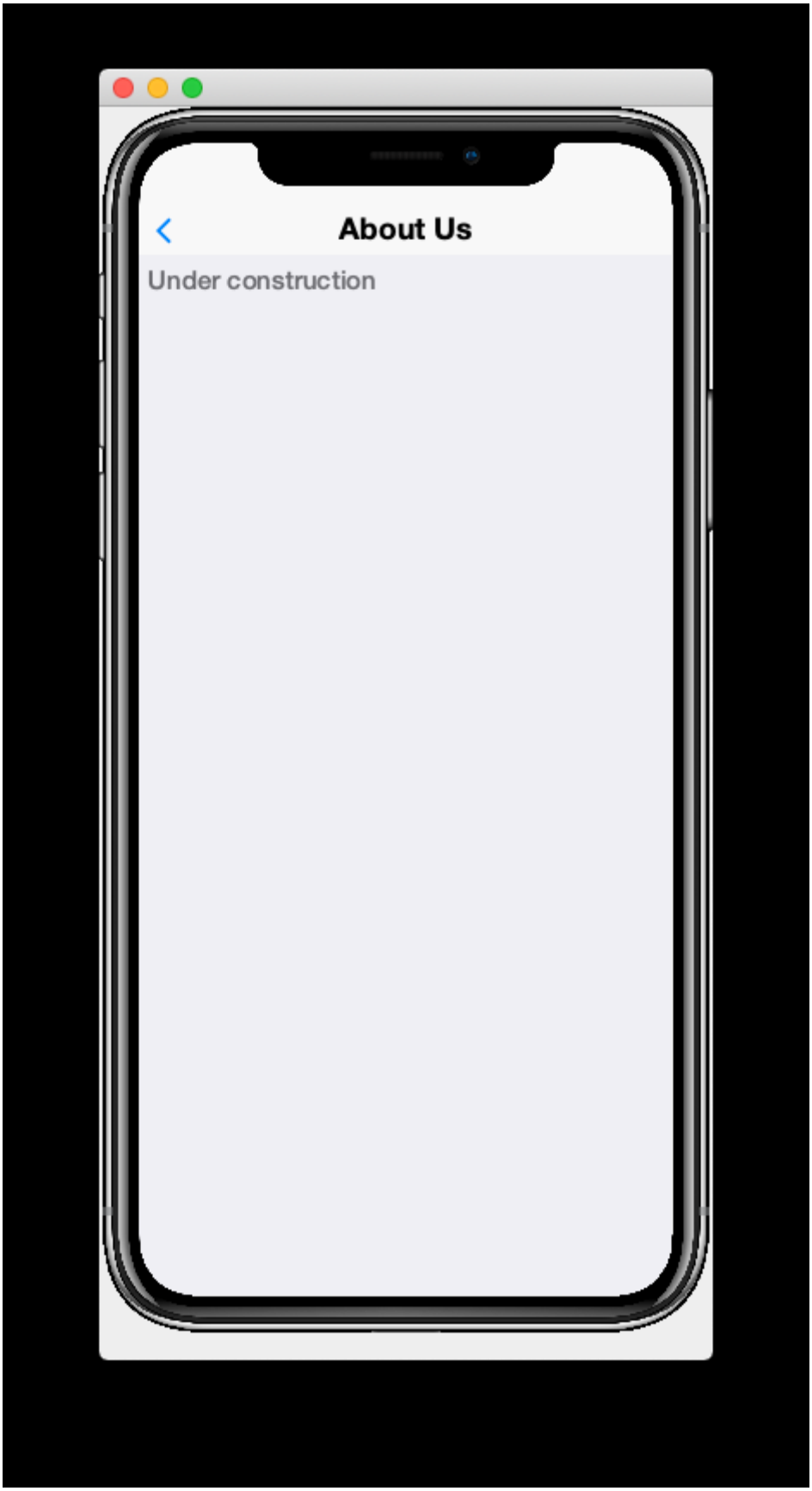
Finally, let's add a button to our original view, *StartPage.xml* as follows:

```
<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).

TIP

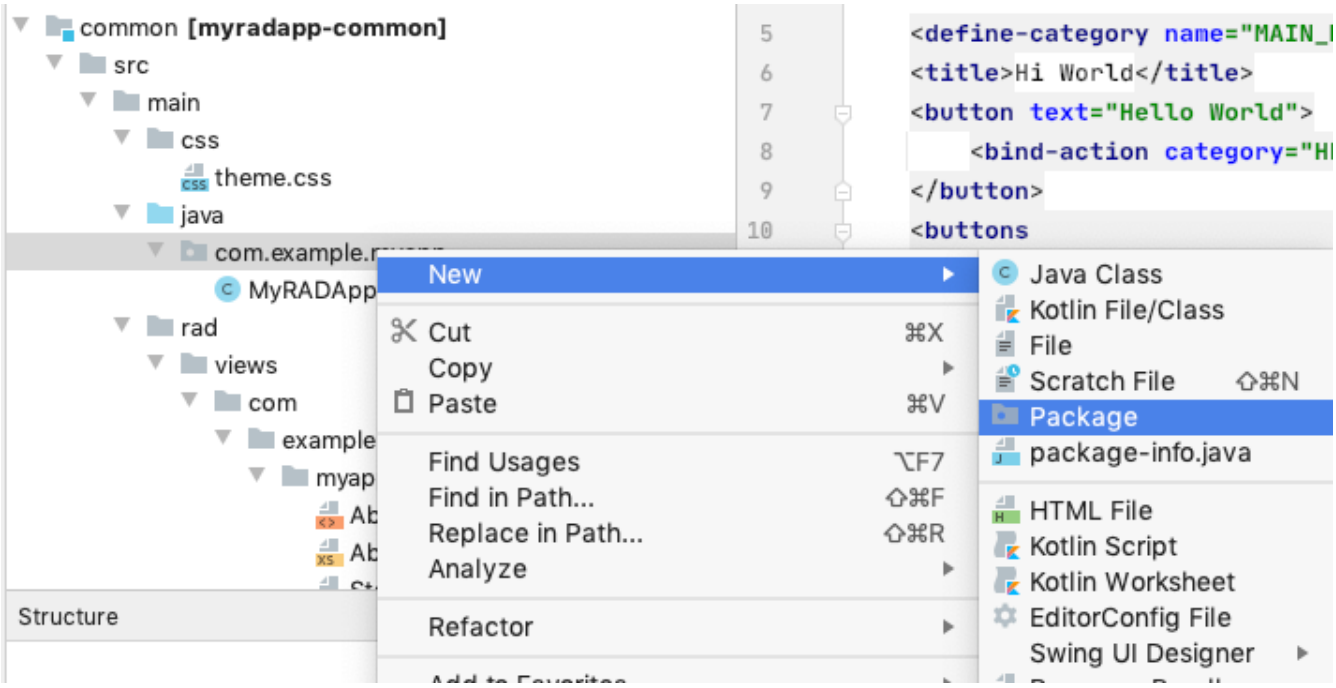
This section described only how to navigate to a different form. It is also possible to load views within the current form using the `rad-href` attribute. This is commonly used on tablet and desktop to create a *master-detail* view. See Intra-Form Navigation for some examples.

2.9. Models

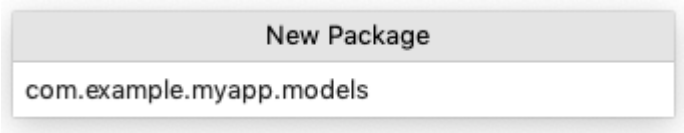
So far we've been working only with the *V* and *C* portions of *MVC*. In this section, I introduce the final pillar in the trinity: *the Model*. Model objects store the data of the application. In CodeRAD, *model* objects implement the *com.codename1.rad.models.Entity* interface. We're going to skip 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:



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



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"/>`

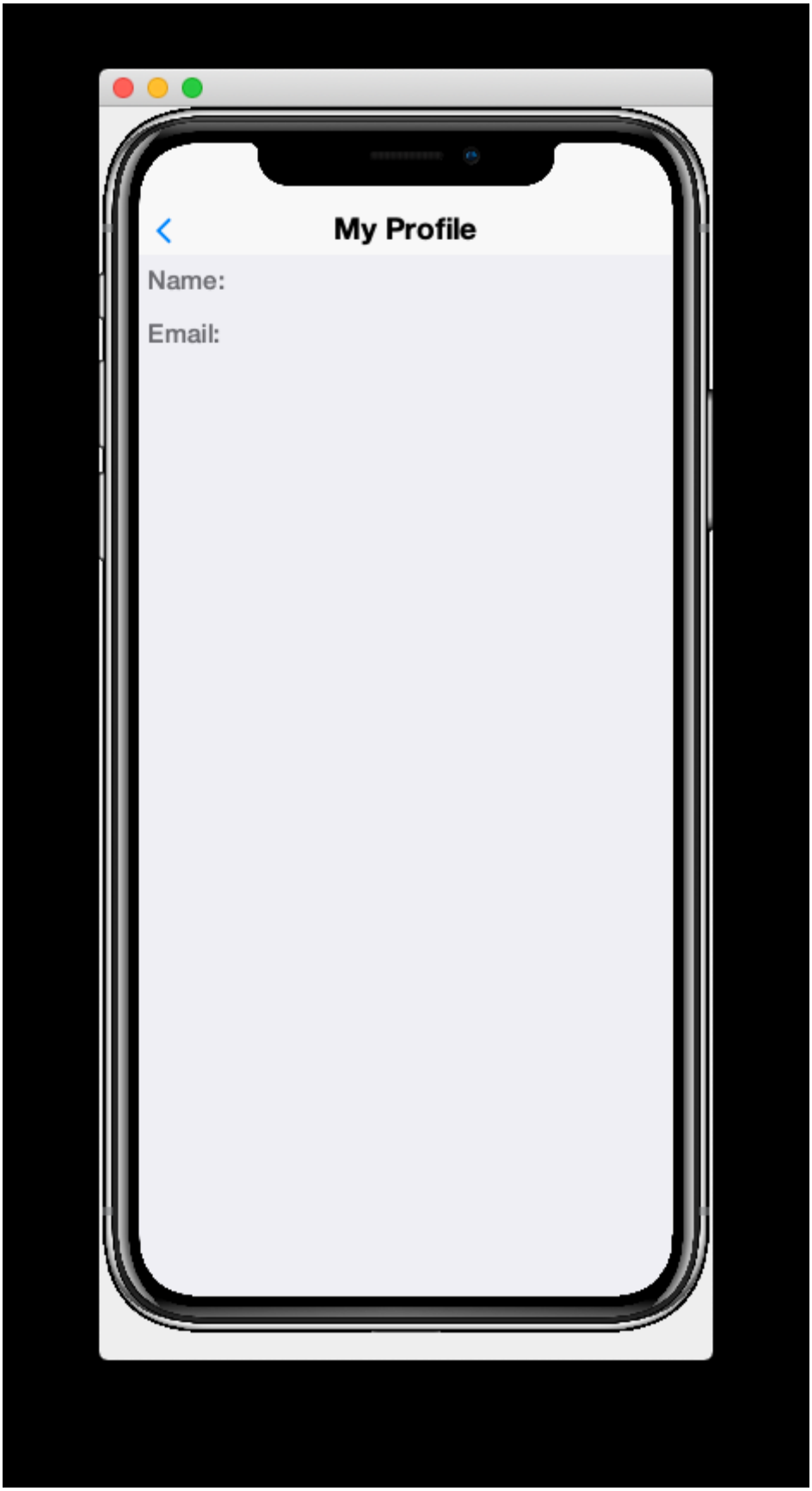
The `<radLabel>` tag is a wrapper around a `Label` that supports binding to a model property. In this case the `tag=Person.name` attribute indicates that this label should be bound to the property of the model with the `Person.name` tag. Recall that the *name* property of the *UserProfile* included the `@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*:

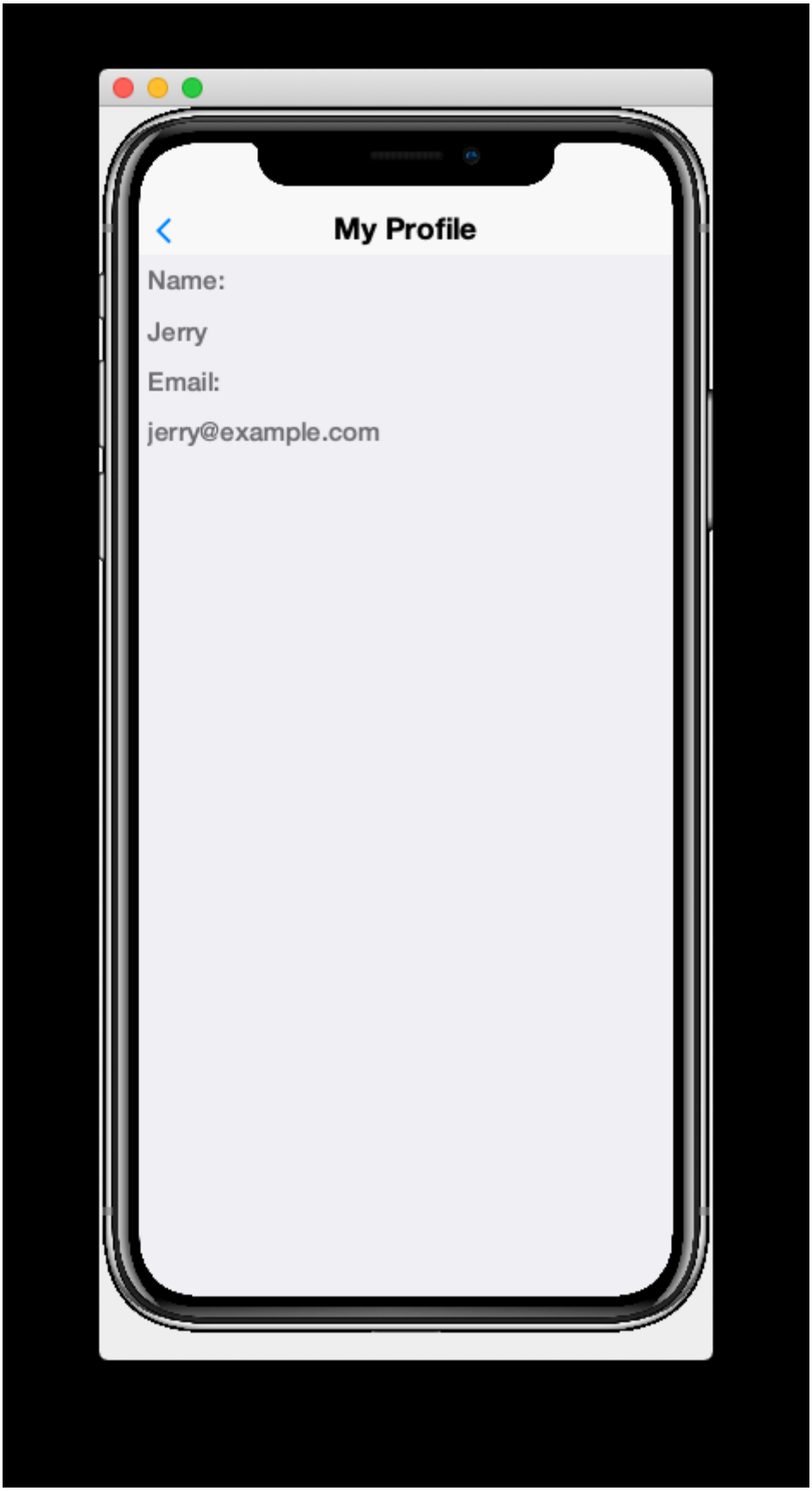
```
<button rad-href="#UserProfilePage{profile}">User Profile</button>
```

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:

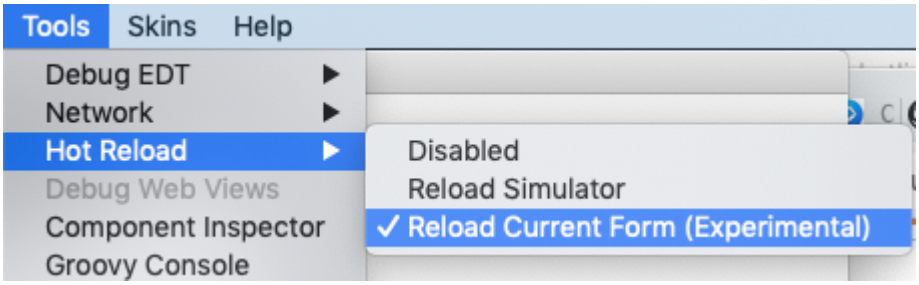
```
<?xml version="1.0"?>
<y scrollableY="true" xsi:noNamespaceSchemaLocation="StartPage.xsd" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
    <!-- We need to import the UserProfile class since we use it
        in various places of this view. -->
    <import>
        import com.example.myapp.models.UserProfile;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


My Profile
Name:
Jerry
Email:
jerry@example.com

TIP

Since the *My Profile* form is a "sub-form" of your app, the *Hot Reload* > *Reload Simulator* option would still require you to navigate back to the form when you make changes to the source. While working on "sub-forms" (i.e. forms that aren't displayed automatically on app start), I recommend enabling the *Hot Reload* > *Reload Current Form* option in the simulator.



This way, when you make changes to the source and the simulator reloads, it will automatically navigate back to the this form. Be aware, however, that upon reload, you will still lose your application state such as the controller hierarchy and model data. For example, you'll notice that the "back" button doesn't appear in your *My Profile* form after reload.

You can disable this feature when you are finished working on the *My Profile* form by changing *Hot Reload* back to *Reload Simulator*.

2.10. Fun with Bindings

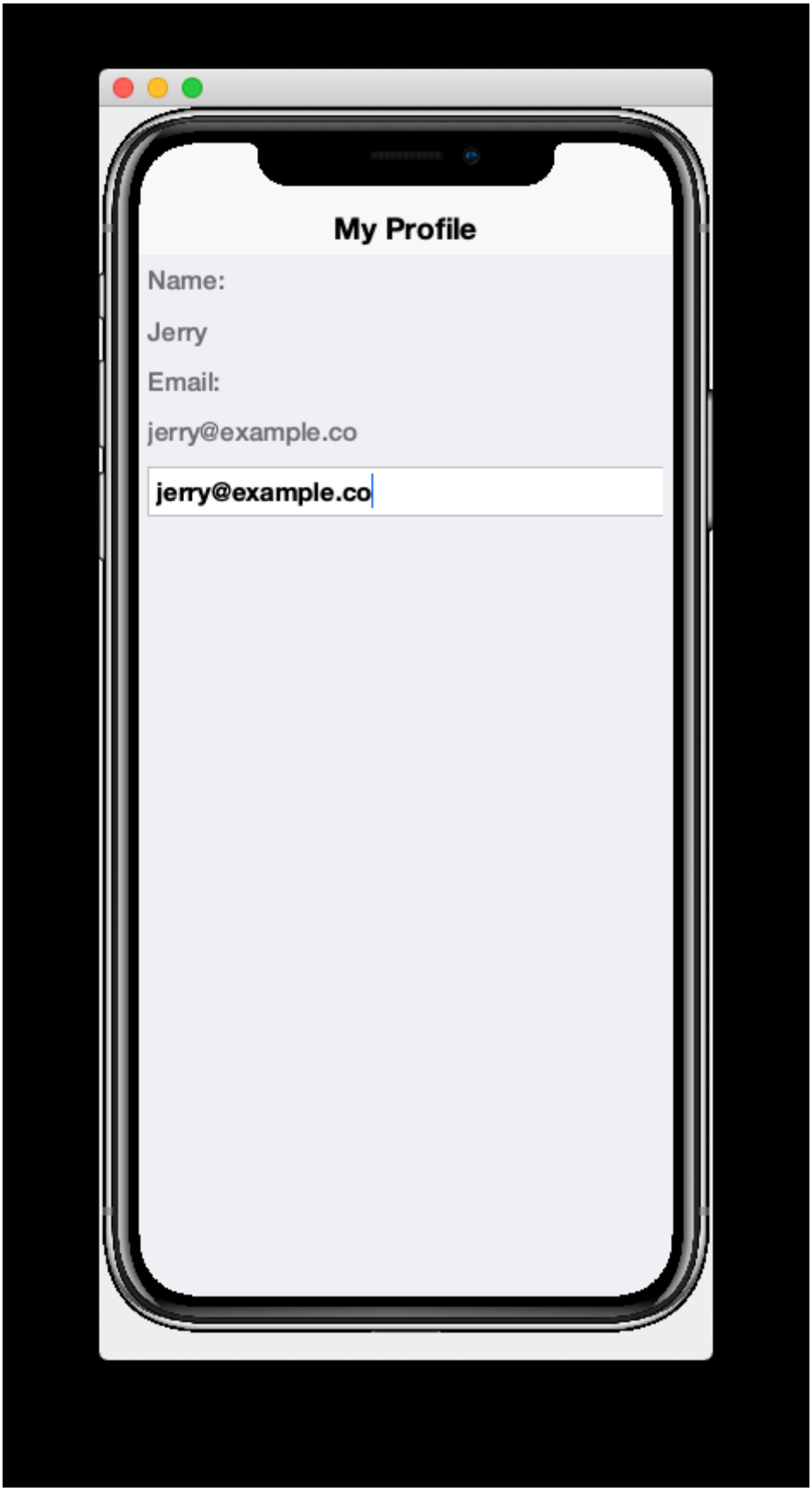
TIP

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.

2.11. 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 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.

2.11.1. 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.

2.12. 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 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:

```
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

2.12.1. 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.

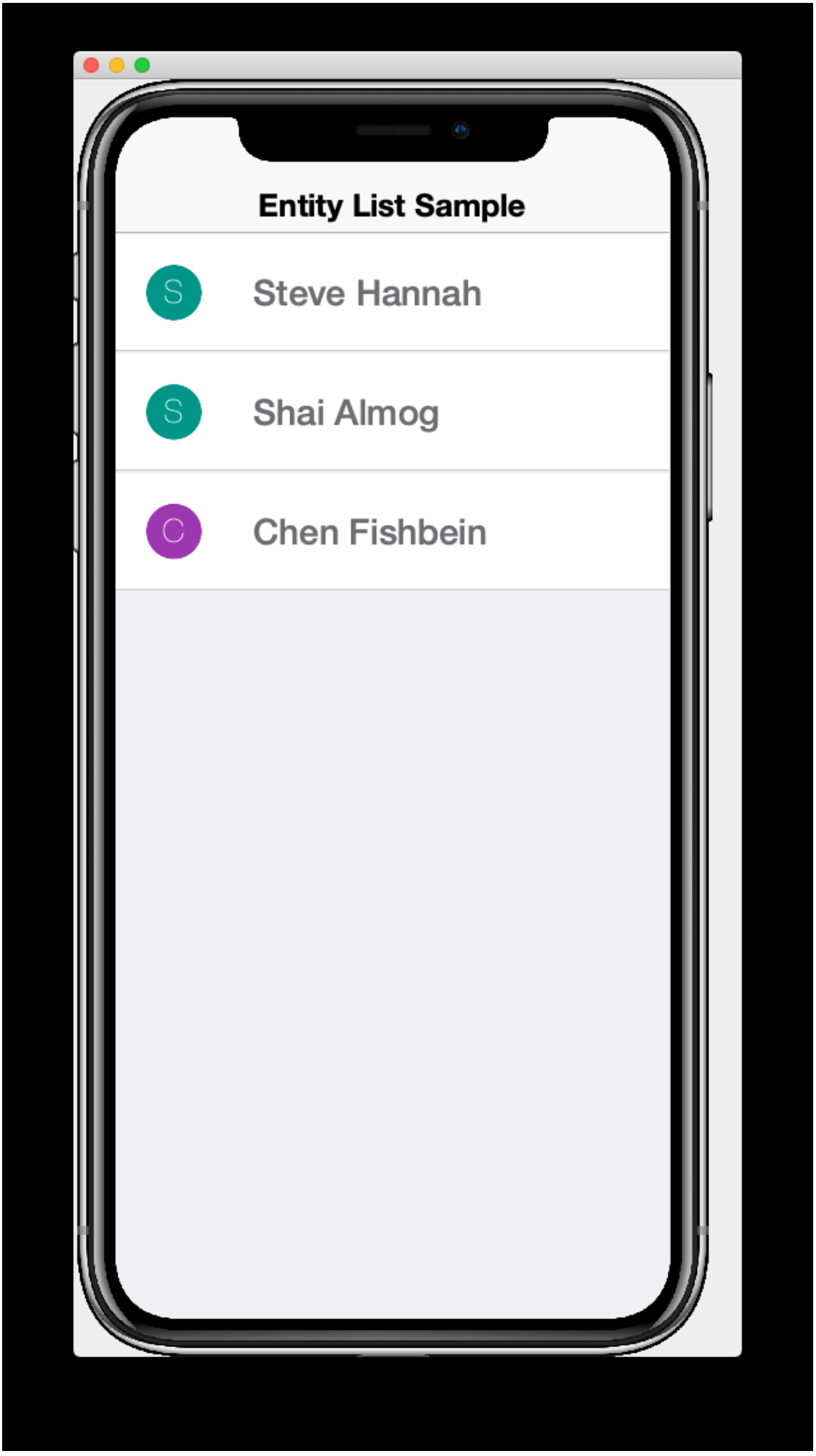
The `<profileAvatar>` tag is a handy component that will display an avatar for the entity. This will check to see if the entity has any properties with the `Thing.thumbnailUrl` tag, and display that image if found. Otherwise it will render an image of the first letter of the name (I.e. the value of a 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:

```
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

2.12.2. Responding to List Row Events

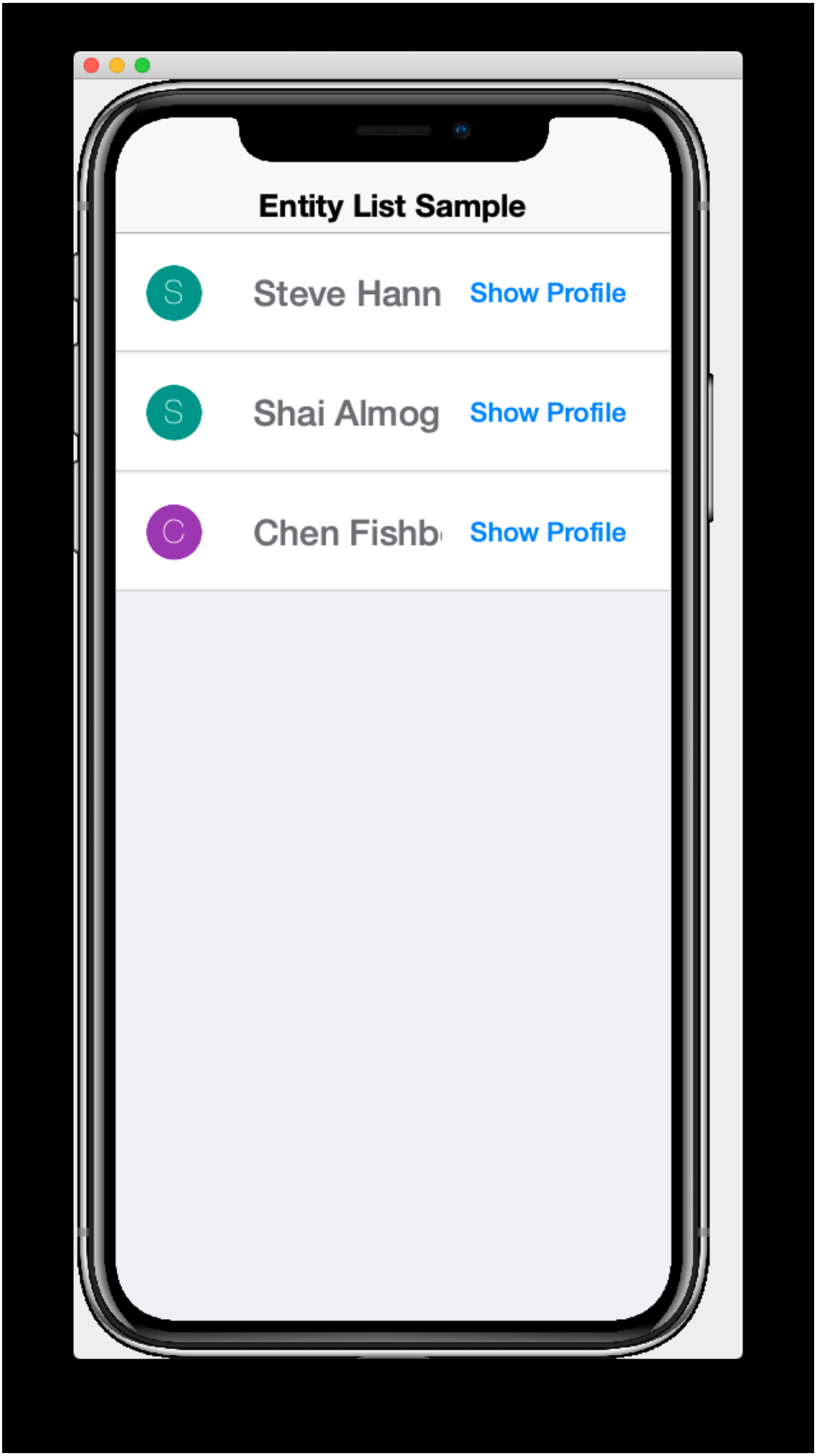
Suppose we want the app to navigate to a UserProfile form for the selected user, when the user clicks on one of the rows of the list.

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

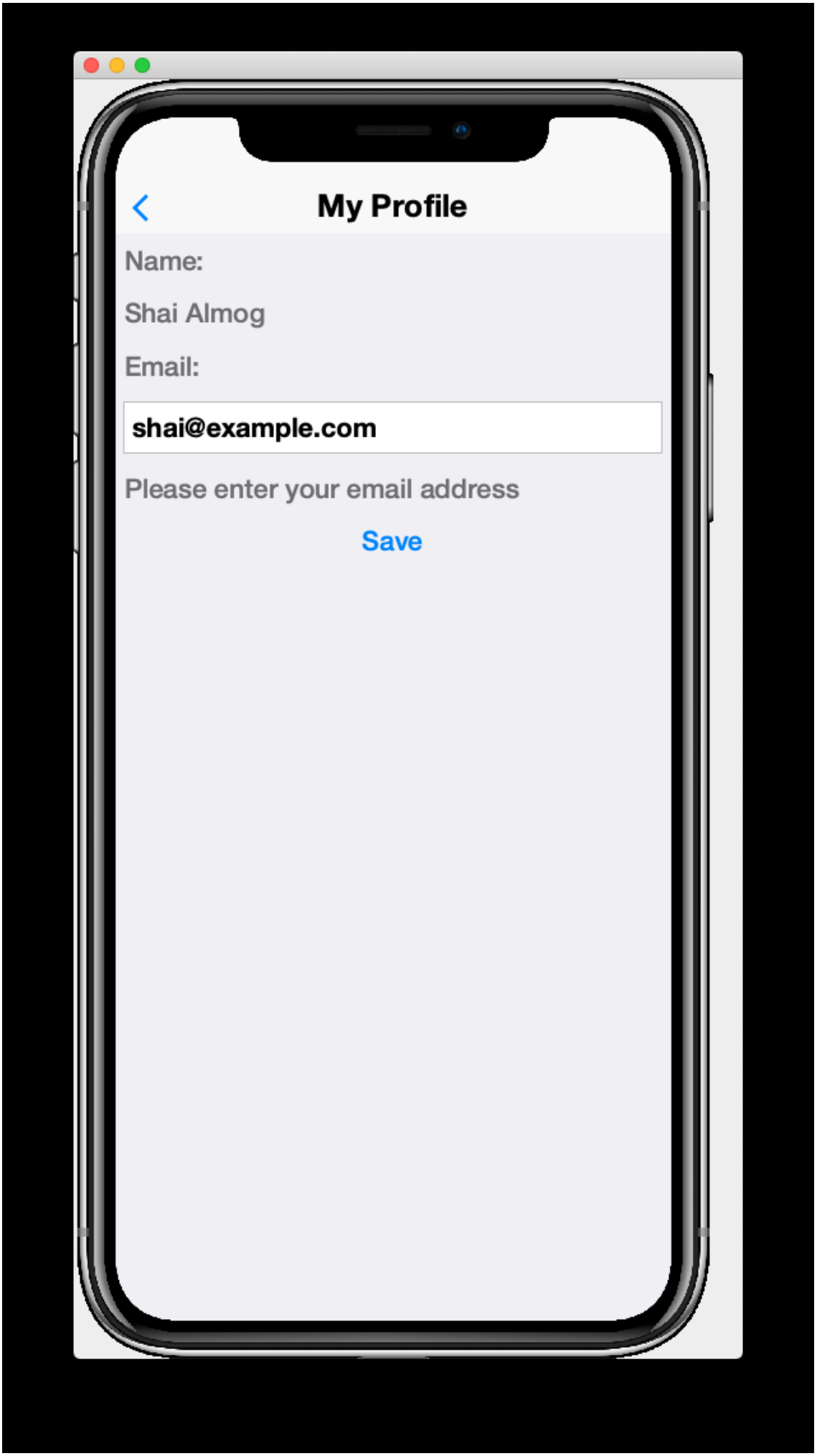
```
<button layout-constraint="east"
    rad-href="#UserProfilePage{}">Show Profile</button>
```

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

2.12.3. 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.

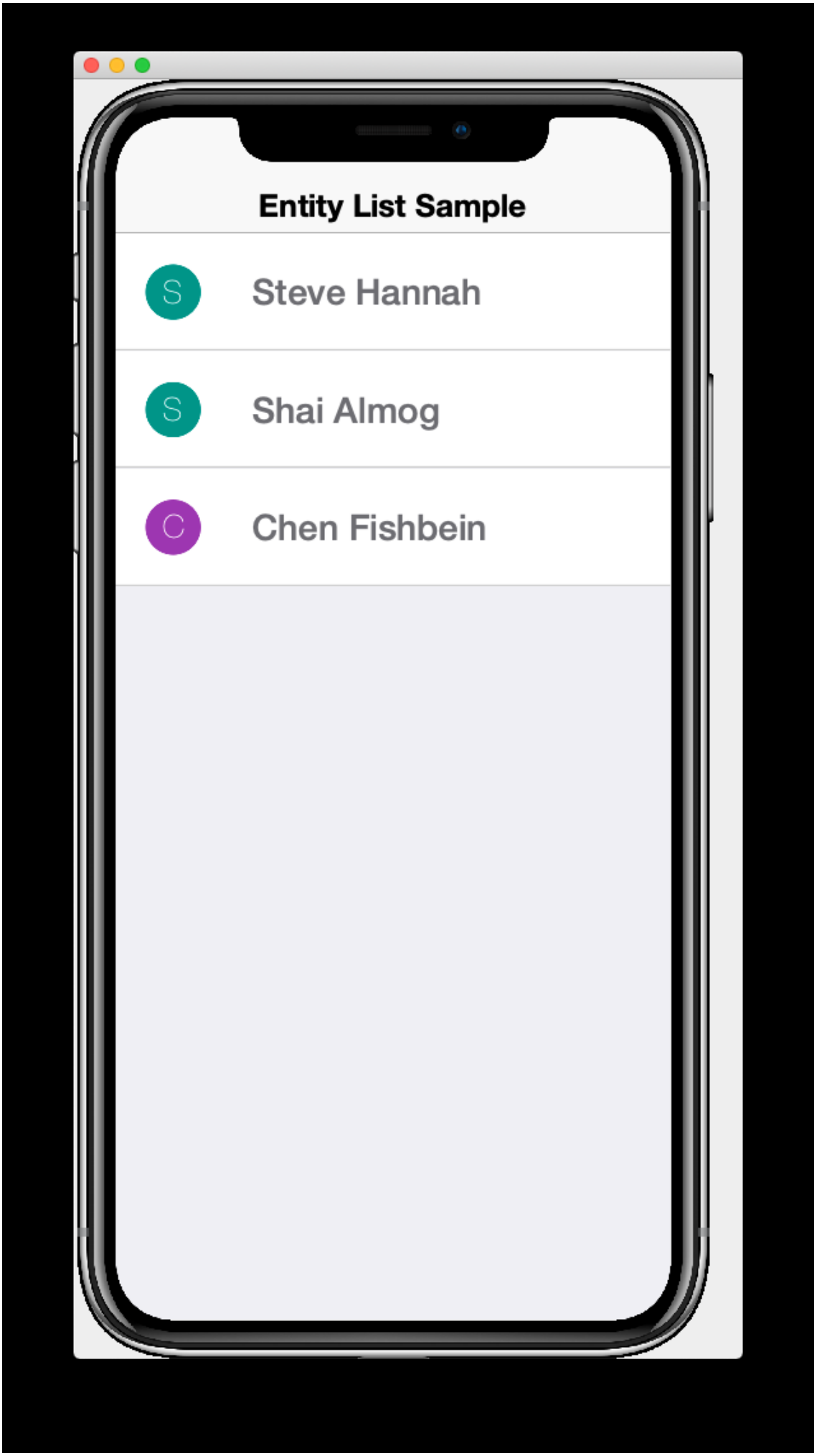
`<button ⋯ uiid="ShowProfileButton"⋯>`

: 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.

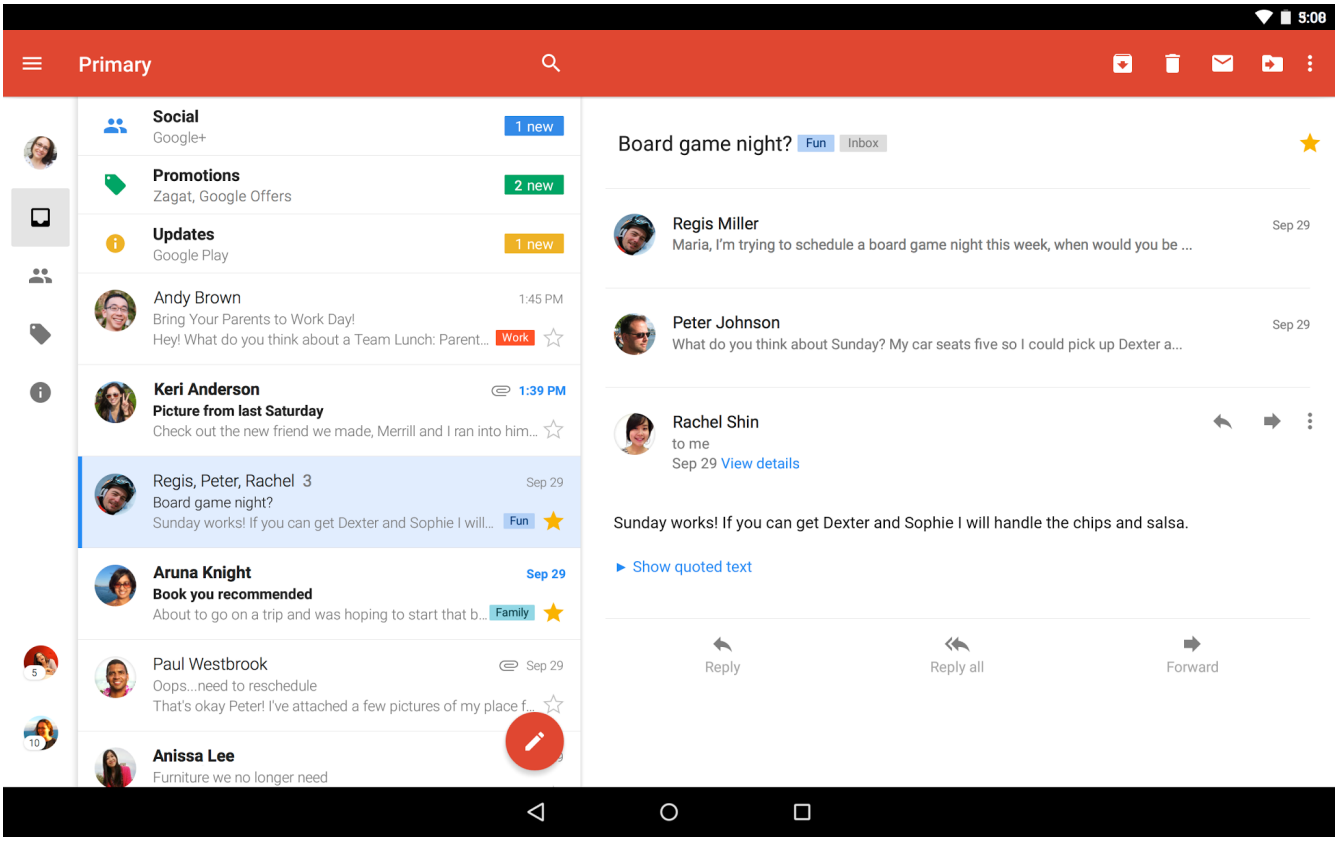
After making these changes, the view should look like:


Entity List Sample
S
Steve Hannah
S
Shai Almog
C
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.

2.13. Intra-Form Navigation

Earlier, in Form Navigation, we learned how to navigate between forms using a *button* tag with the `rad-href` attribute. When developing for tablet and desktop, you may want to navigate to a different view within the same form; sort of like using frames in HTML. A *mail* app will often have a list of messages on the left side of the screen, and details of the currently selected message on the right, as shown in the Gmail app screenshot below:



In our previous examples with `rad-href`, we specified *which* view we wanted to navigate to, but we didn't specify *where* we wanted the view to be displayed. By default, it navigates to a new form whose *FormController* is a child of the current *FormController*. The full syntax of `rad-href` supports targeting the view to a different location in both the view hierarchy and the controller hierarchy.

Suppose we wanted our view to be displayed inside a *Sheet* instead of a new form. Then we could do something like:

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

Alternatively, suppose we wanted to display the view inside a Container within the current form. Then we could do:

```
<border name="TargetFrame"></border> ①

...

<button rad-href="#AboutPage sel:#TargetFrame">About Page</button> ②
```

① A placeholder container where the *AboutPage* view will be loaded.

② The `sel:` prefix for the target means that the remainder will be treated as a `ComponentSelector` query, which is similar to a CSS selector of Javascript Query Selector. In our case we are targeting the component with name "TargetFrame".

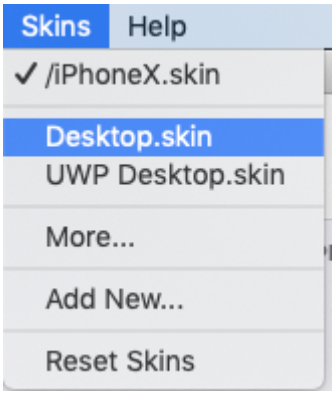
In the above example, when the user presses the button, it will load the *AboutPage* view into the *TargetFrame* container.

Change the contents of the *StartPage* view to:

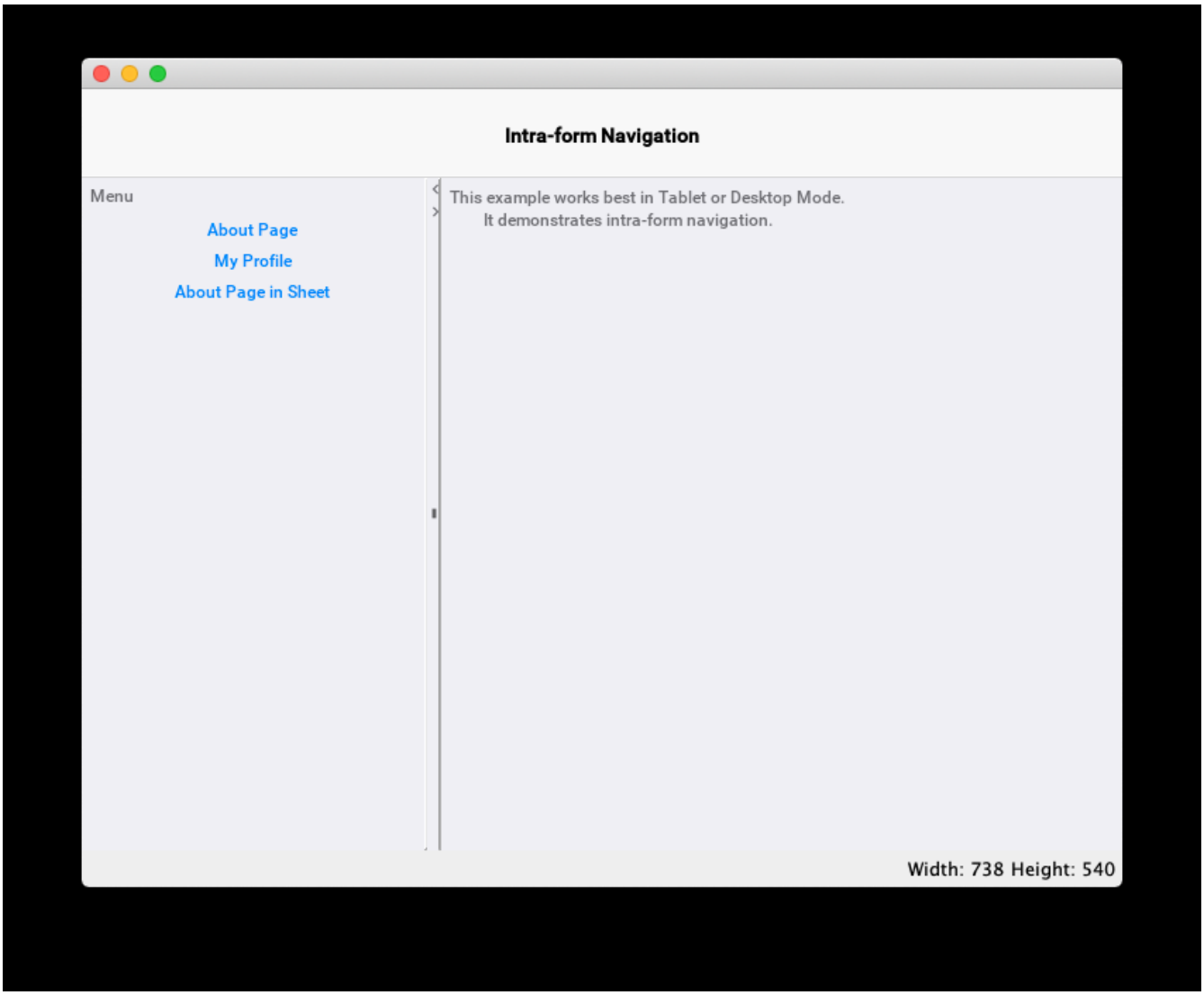
```
<?xml version="1.0" encoding="UTF-8" ?>
<splitPane xsi:noNamespaceSchemaLocation="StartPage.xsd" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
    <title>Intra-form Navigation</title>
    <var name="profile" lookup="com.example.myapp.models.UserProfile"/>
    <y>
        <label>Menu</label>
        <button rad-href="#AboutPage sel:#ContentPanel">About Page</button>
        <button rad-href="#UserProfilePage{profile} sel:#ContentPanel">My
Profile</button>
        <button rad-href="#AboutPage sheet">About Page in Sheet</button>
    </y>
    <border>
        <spanLabel layout-constraint="north">This example works best in Tablet or
Desktop Mode.
        It demonstrates intra-form navigation.
        </spanLabel>
        <border layout-constraint="center" name="ContentPanel"></border>

    </border>
</splitPane>
```

TIP The above example demonstrates the `<splitPane>` component that is useful for tablet and desktop UIs. See [using-split-panes] to learn more about the *SplitPane* component.

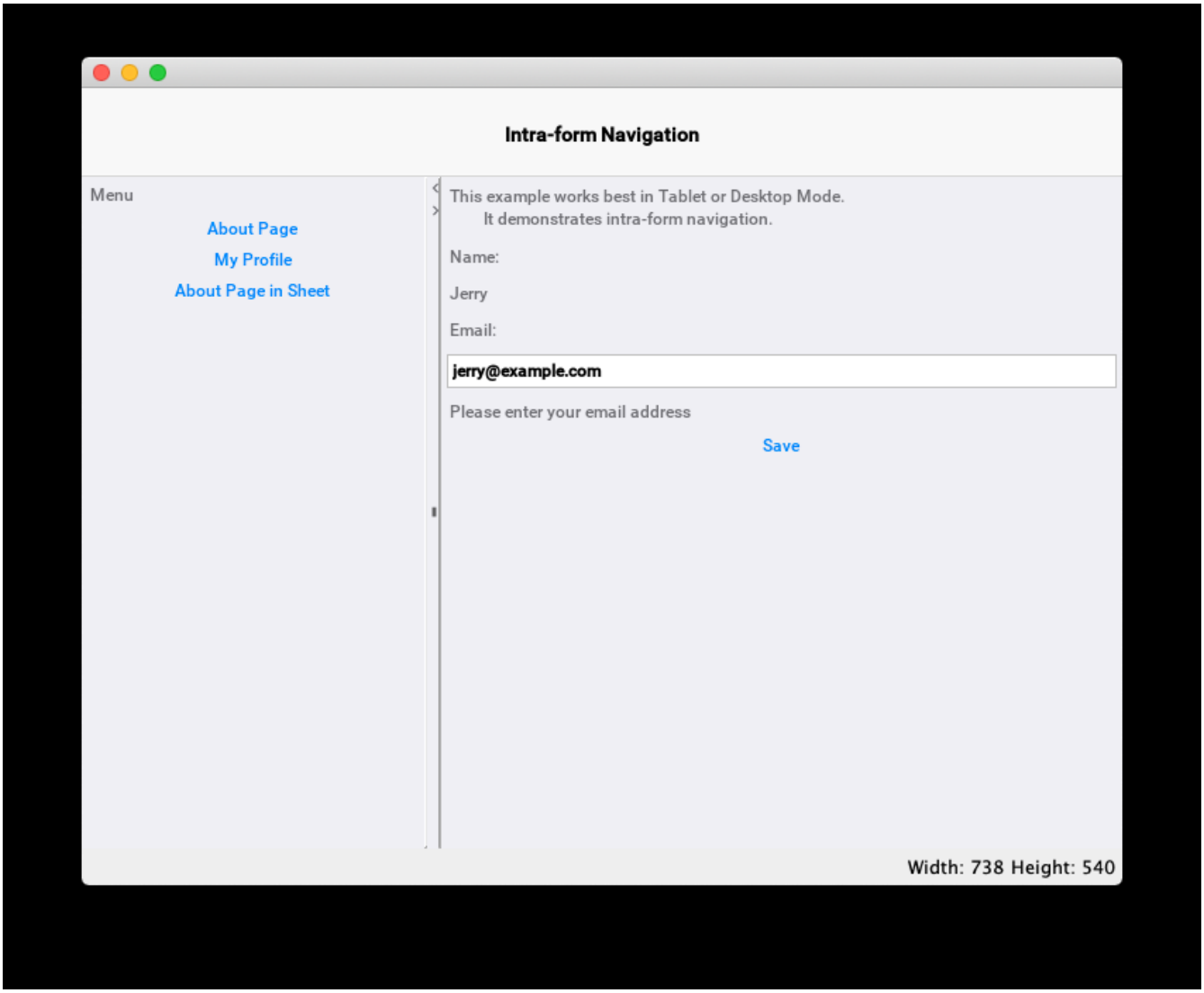
Launch the simulator, and enable the *Desktop* skin by selecting the *Skin* > *Desktop.skin* menu item as shown below.



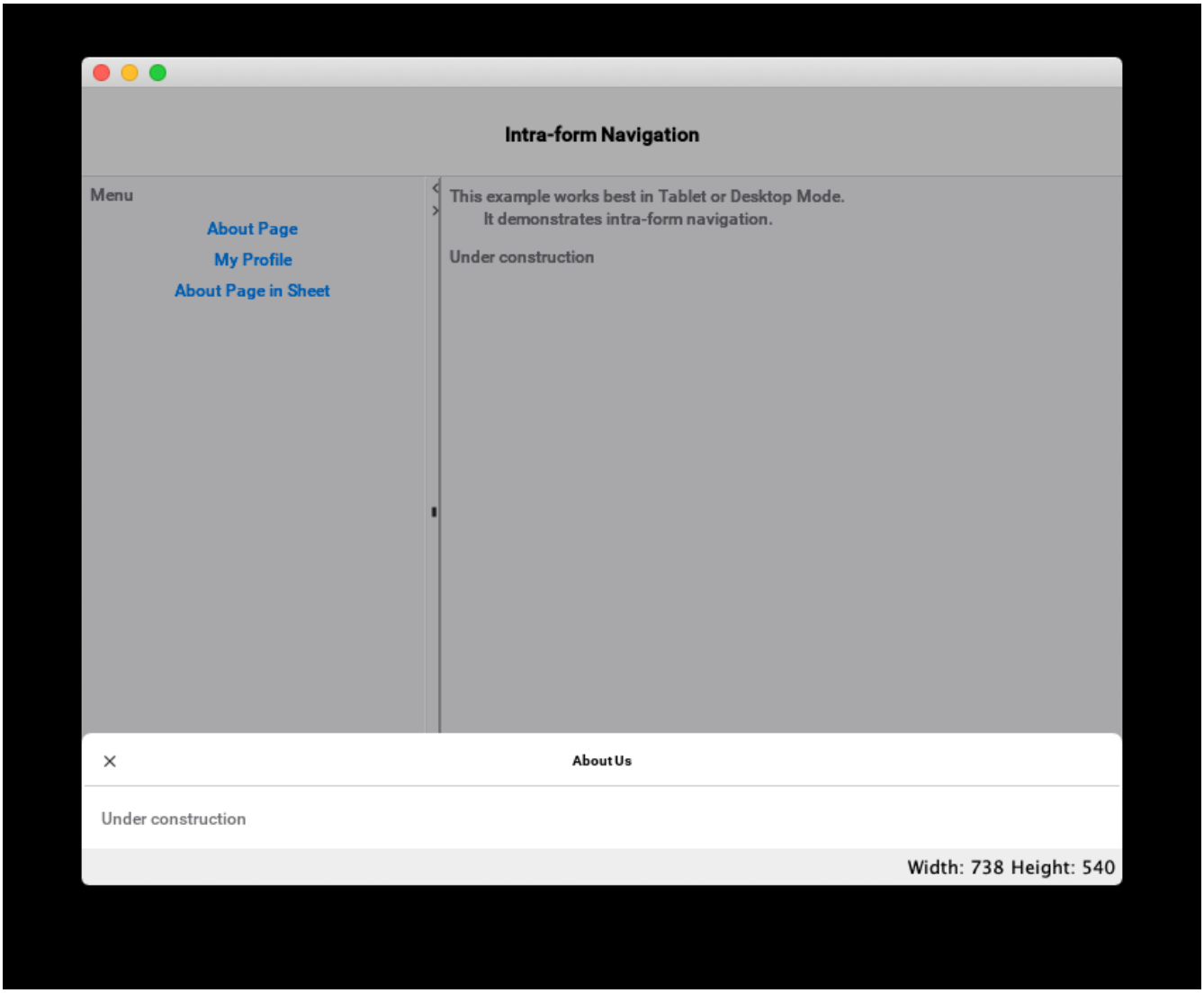
Then the app should appear something like the followign screenshot:



Click on the *My Profile* link on the left, and you should see the user profile page appear on the right, as shown below.



If you click on the *About Page in Sheet* button, it will load the *AboutPage* view inside a sheet as shown here.



You can also control the position of where the sheet will be shown by using one `sheet-top`, `sheet-left`, `sheet-right`, or `sheet-center` instead of the `sheet` option that we used in this example.

2.13.1. Navigation Transitions

You can use the `rad-transition` attribute in conjuction with the `rad-href` attribute also, to specify a transition to be used when replacing a container's content with a new view.

To demonstrate this, let's add a `rad-transition` attribute to each button in the menu from the previous example. Change the first *AboutPage* button to:

```
<button rad-href="#AboutPage sel:#ContentPanel"
            rad-transition="rad-href 0.5s flip"
        >About Page</button>
```

Now, when you click this button, it should transition the *AboutPage* in with a *flip* transition with a duration of 0.5 seconds.

Some other transition types include *fade*, *slide*, *cover*, and *uncover* with variants to specify direction, such as *slide-up*, *slide-down*, *slide-left*, etc...

See [chapter-transitions] for more details and examples using transitions.

2.14. Custom View Controllers

Up until now, we haven't created any custom controllers for our views, other than the application controller (the *MyRadApp* class). Since all events propagate up the controller hierarchy, it is possible just to handle all of the events in the application controller, as we've been doing. Keeping all of our logic inside a single *application-wide* controller has some benefits for small, example apps, but for most real-world apps, you'll want to be intentional about your application's architecture.

Best practice is to create a *ViewController* for each view, which will be responsible for handling application logic pertaining to that view. This practice will promote better modularity, which will make it easier to maintain your code, and to reuse components in other projects.

You can assign a custom view controller to a view by adding a `view-controller` attribute to the root element of your view. E.g.

```
<?xml version="1.0" ?>
<y view-controller="com.example.myapp.controllers.StartPageViewController">
   .....
</y>
```

TIP

If your controller class is covered by an *import* directive in your view, then you could just provide the *simple* name of the controller class, rather than the fully-qualified name. E.g. the following would also work:

```
<?xml version="1.0" ?>
<y view-controller="StartPageViewController">
  <import>
    import com.example.myapp.controllers.*;
  </import>
  ...
</y>
```

Let's expand this to a complete example.

In our sample app project, create a new package in the *common/src/main/java* directory named *com.example.myapp.controllers*, and create a new Java class in this package named "StartPageViewController.java" with the following contents:

common/src/main/java/com/example/myapp/controllers/StartPageViewController.java

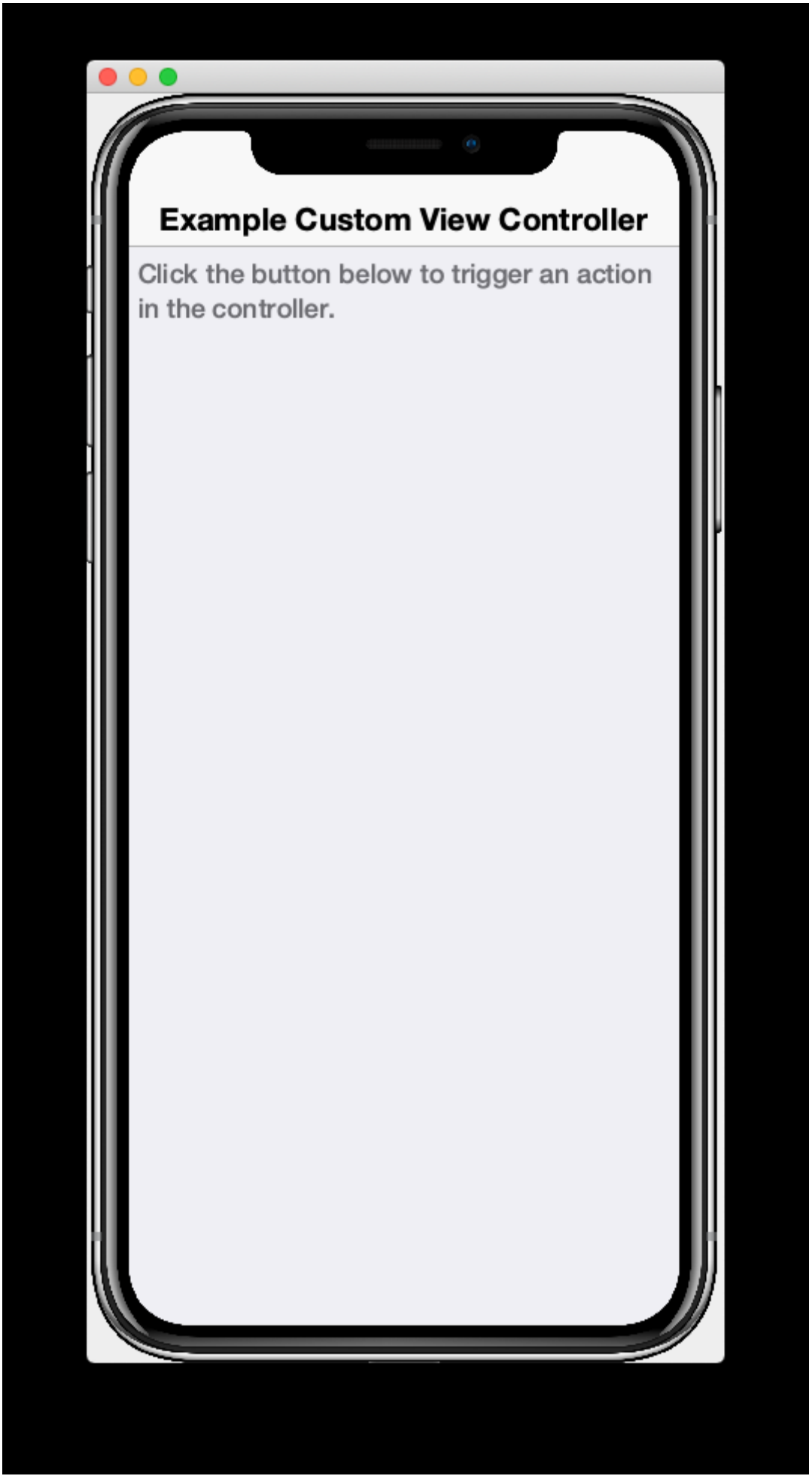
```
package com.example.myapp.controllers;

import com.codename1.rad.controllers.Controller;
import com.codename1.rad.controllers.ViewController;

public class StartPageViewController extends ViewController {
    /**
     * Creates a new ViewController with the given parent controller.
     *
     * @param parent
     */
    public StartPageViewController(Controller parent) {
        super(parent);
    }
}
```

Now, change the *StartPage.xml* template to the following content:

common/src/main/rad/views/com/example/myapp/StartPage.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<y view-controller="com.example.myapp.controllers.StartPageViewController"
   xsi:noNamespaceSchemaLocation="StartPage.xsd" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
    <title>Example Custom View Controller</title>

    <!-- Define an action category for the controller to
        receive events when the "Hello" button is clicked -->
    <define-category name="HELLO_CLICKED"/>

    <spanLabel>Click the button below to trigger an action in the
controller.</spanLabel>

    <button>
        <bind-action category="HELLO_CLICKED"/>
    </button>


</y>
```

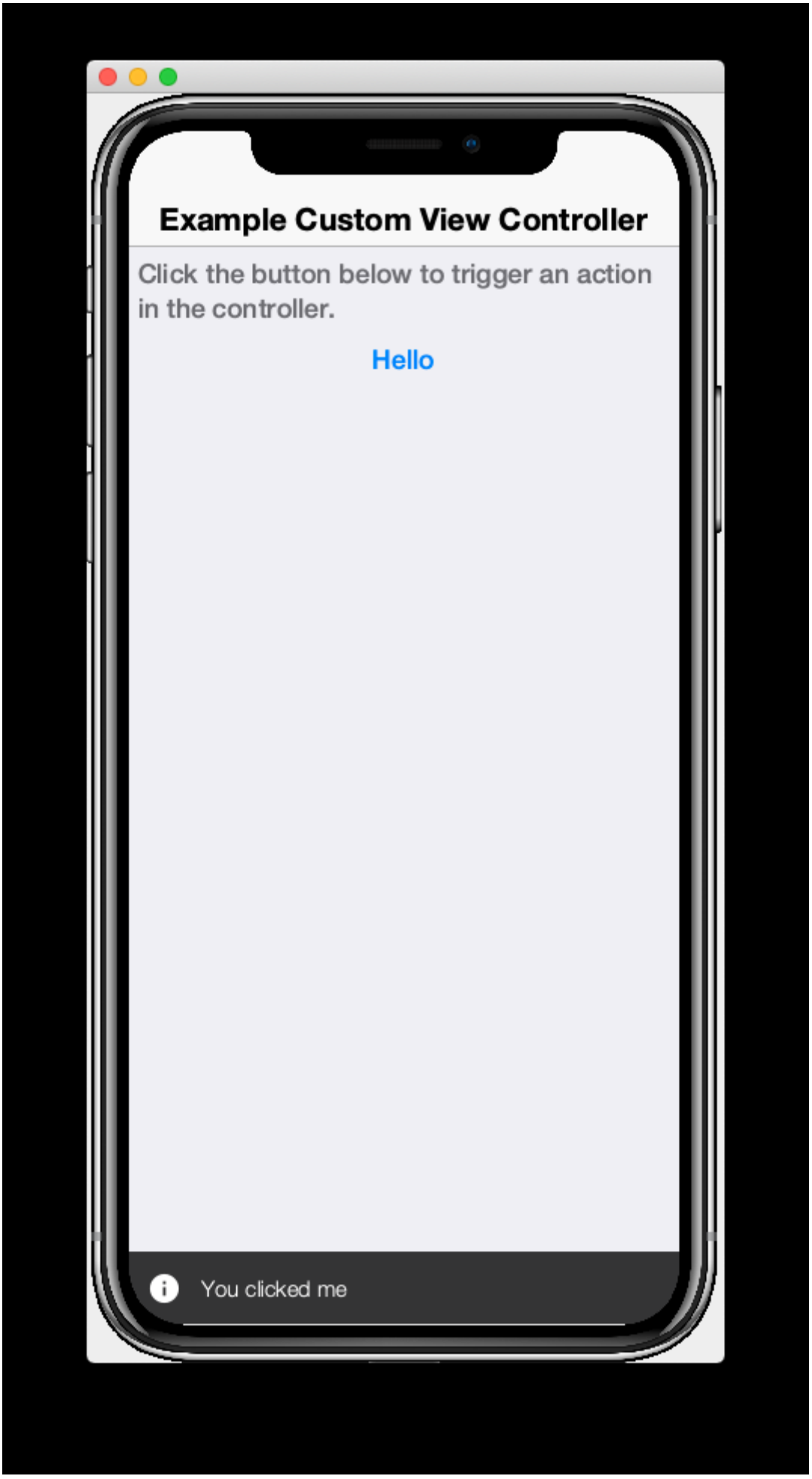
Now try running the example in the simulator.


Example Custom View Controller
Click the button below to trigger an action
in the controller.

Our button is conspicuously missing from this form. This is because it is bound to the *HELLO_CLICKED* action category, but our controller hasn't added any actions to this category yet. Let's add an action to our view controller now.

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

        // Register an action with HELLO_CLICKED category so that the view
        // will bind it to the button.
        ActionNode.builder()
                .label("Hello")
                .addToController(this, StartPage.HELLO_CLICKED, evt -> {

                    // Consume the event so that it doesn't propagage up the
controller
                    // hierarchy.
                    evt.consume();

                    // Show a message to confirm that we received the event.
                    ToastBar.showInfoMessage("You clicked me");
                });
    }
```

The simulator should automatically reload upon saving the controller file, and the "Hello" button should appear. Click "Hello" to confirm that our *ToastBar* info message appears as shown below:


Example Custom View Controller
Click the button below to trigger an action in the controller.
Hello
You clicked me

2.15. Views within Views

Since RAD views are Components themselves, they can be used inside other views, just like other components are. To demonstrate this, let's create create a form to allow users to enter contact information such as name, email, billing address, and shipping address. Since the billing address and shipping address will likely use the same fields, we'll create a *AddressView* view and use it from the main form.

Create a new view in the same package as our existing views named *AddressForm.xml* with the following contents:

```
<?xml version="1.0"?>
<y xsi:noNamespaceSchemaLocation="AddressForm.xsd" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
    <define-tag name="streetAddress" value="PostalAddress.streetAddress"/>
    <define-tag name="city" value="PostalAddress.addressLocality"/>
    <define-tag name="province" value="PostalAddress.addressRegion"/>
    <define-tag name="country" value="PostalAddress.addressCountry"/>
    <define-tag name="postalCode" value="PostalAddress.postalCode"/>

    <radTextField tag="streetAddress"
                  component.hint="Street Address"/>
    <radTextField tag="city"
        component.hint="City"
    />
    <radTextField tag="province"
        component.hint="Province"
    />
    <radTextField tag="country"
        component.hint="Country"
    />
    <radTextField tag="postalCode"
        component.hint="Postal Code"
    />

</y>
```

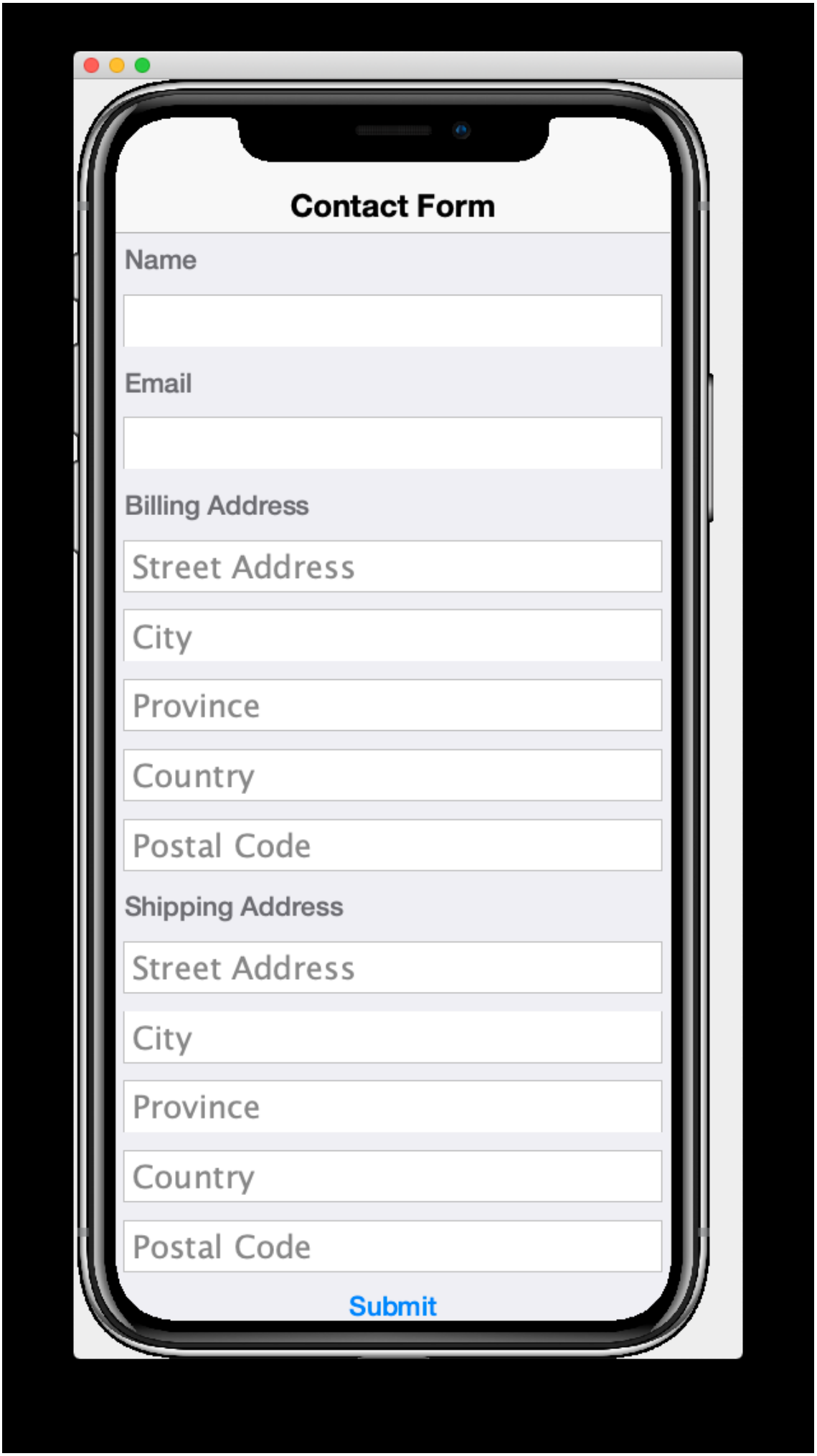
Now create another view called *ContactForm.xml* in the same directory with the following contents.

```
<?xml version="1.0"?>
<y scrollableY="true" xsi:noNamespaceSchemaLocation="ContactForm.xsd" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
    <title>Contact Form</title>

    <!-- Define some tags for the view model -->
    <define-tag name="name" value="Person.name"/>
    <define-tag name="email" value="Person.email"/>

    <label>Name</label>
    <radTextField tag="name"></radTextField>

    <label>Email</label>
    <radTextField tag="email"></radTextField>

    <!-- Embed an AddressForm view for the billing address -->
    <label>Billing Address</label>
    <addressForm view-model="new"/>

    <!-- Embed another AddressForm view for the shipping address -->
    <label>Shipping Address</label>
    <addressForm view-model="new"/>

    <!-- Submit button .. doesn't do anything yet -->
    <button text="Submit"/>

</y>
```

And finally, modify the `actionPerformed()` method of the `MyRADApp` class to display the contact form we just created on start:

```
public void actionPerformed(ControllerEvent evt) {

        with(evt, StartEvent.class, startEvent -> {
            if (true) {
                // Temporarily making ContactForm the default form.
                new ContactFormController(this).show();
                return;
            }
            ...
        });
        super.actionPerformed(evt);
    }
```

If you fire up the simulator, the app should look something like the following screenshot:


Contact Form
Name
Email
Billing Address
Street Address
City
Province
Country
Postal Code
Shipping Address
Street Address
City
Province
Country
Postal Code
Submit

Notice that the contact form includes two embedded address forms. One for the *billing address* and the other for the *shipping address*. Let's walk through the *ContactForm.xml* source to see what is happening here.

You've seen most of the tags before in previous examples. The new part that I'd like to highlight here are the two `<addressForm>` tags:

```
<!-- Embed an AddressForm view for the billing address -->
    <label>Billing Address</label>
    <addressForm view-model="new"/>

    <!-- Embed another AddressForm view for the shipping address -->
    <label>Shipping Address</label>
    <addressForm view-model="new"/>
```

These create instances of the `AddressForm` view that we defined in the `AddressForm.xml`file. The `view-model` attribute is necessary to specify the view model that should be used for the address form. The value "new" is a special value that indicates that the view should create a new view model for itself. If this were omitted, it would attempt to use the view model of the current view which we don't want here, because the model for the ContactForm isn't compatible with the model for the AddressForm.

This is roughly equivalent to:

```
<addressForm view-model="new AddressFormModelImpl()"/>
```

With our current contact form, the *AddressForm* view models aren't connected to the *ContactForm* view model, which isn't idea. Let's improve this by defining tags for *billingAddress* and *shippingAddress* in the *ContactForm* view model:

```
<define-tag name="shippingAddress" type="AddressFormModel" initialValue="new"/>
<define-tag name="billingAddress" type="AddressFormModel" initialValue="new"/>
```

TIP

The `initialValue` attribute here specifies the initial value that new model objects should assign to the property. In this case we use the special value "new", which is equivalent here to `initialValue="new AddressFormModelImpl()"`. If we omit this `initialValue` it will leave the properties as `null` until we explicitly set them, which might bite us later on.

Now, change the `view-model` attribute of the `<addressForm>` tags to use the *shippingAddress* and *billingAddress* properties respectively:

```
<!-- Embed an AddressForm view for the billing address -->
<label>Billing Address</label>
<addressForm view-model="context.getEntity().getBillingAddress()"/>

<!-- Embed another AddressForm view for the shipping address -->
<label>Shipping Address</label>
<addressForm view-model="context.getEntity().getShippingAddress()"/>
```

Notice that I used the explicit `getBillingAddress()` and `getShippingAddress()` methods on the *ContactForm* view model. I could also have used the more generic `getEntity(TAG)` method:

```
context.getEntity().getEntity(billingAddress)

context.getEntity().getEntity(shippingAddress)
```

Both are fine here, but I chose to use the explicit getters as it is more succinct and easier to understand.

Later on you'll learn another, more succinct, way to access properties of the view model using RAD property macros. E.g. The following is also equivalent:

```
<!-- Embed an AddressForm view for the billing address -->
<label>Billing Address</label>
<addressForm view-model="${billingAddress}"/>

<!-- Embed another AddressForm view for the shipping address -->
<label>Shipping Address</label>
<addressForm view-model="${shippingAddress}"/>
```

NOTE

Fun Fact: You can also use the same model for both views. E.g.:

```
<!-- Embed an AddressForm view for the billing address -->
<label>Billing Address</label>
<addressForm view-model="${billingAddress}"/>

<!-- Embed another AddressForm view for the shipping address -->
<label>Shipping Address</label>
<addressForm view-model="${billingAddress}"/>
```

In this case, if you start typing into any fields in *billing address*, it will also update the corresponding fields in *shipping address*.

2.16. Developing Custom Components

Since you can use *any* component (i.e. `com.codename1.ui.Component` subclass) in your views, it follows that you can also develop your own components and use them in your views. You've already seen a special case of this in Views within Views, where we used a view that we created from another view.

The only thing you *may* need to do in order to use your custom component from a *view* is add an *import* statement for your component's class.

TIP	RAD views automatically import several of the core packages containing components, such as `com.codename1.ui.*`, `com.codename1.rad.ui.entityviews.*`, etc... You only need to explicitly *import* packages and classes that aren't among these default packages.

If your component has a *no-arg* constructor, then it should *just* work. If it doesn't have a *no-arg* constructor, or it has some special requirements for how it is used, then you may need to also implement a *ComponentBuilder* for your component. Later on, I'll also show you how to use dependency injection to have certain properties and arguments automatically "injected" into your component at runtime.

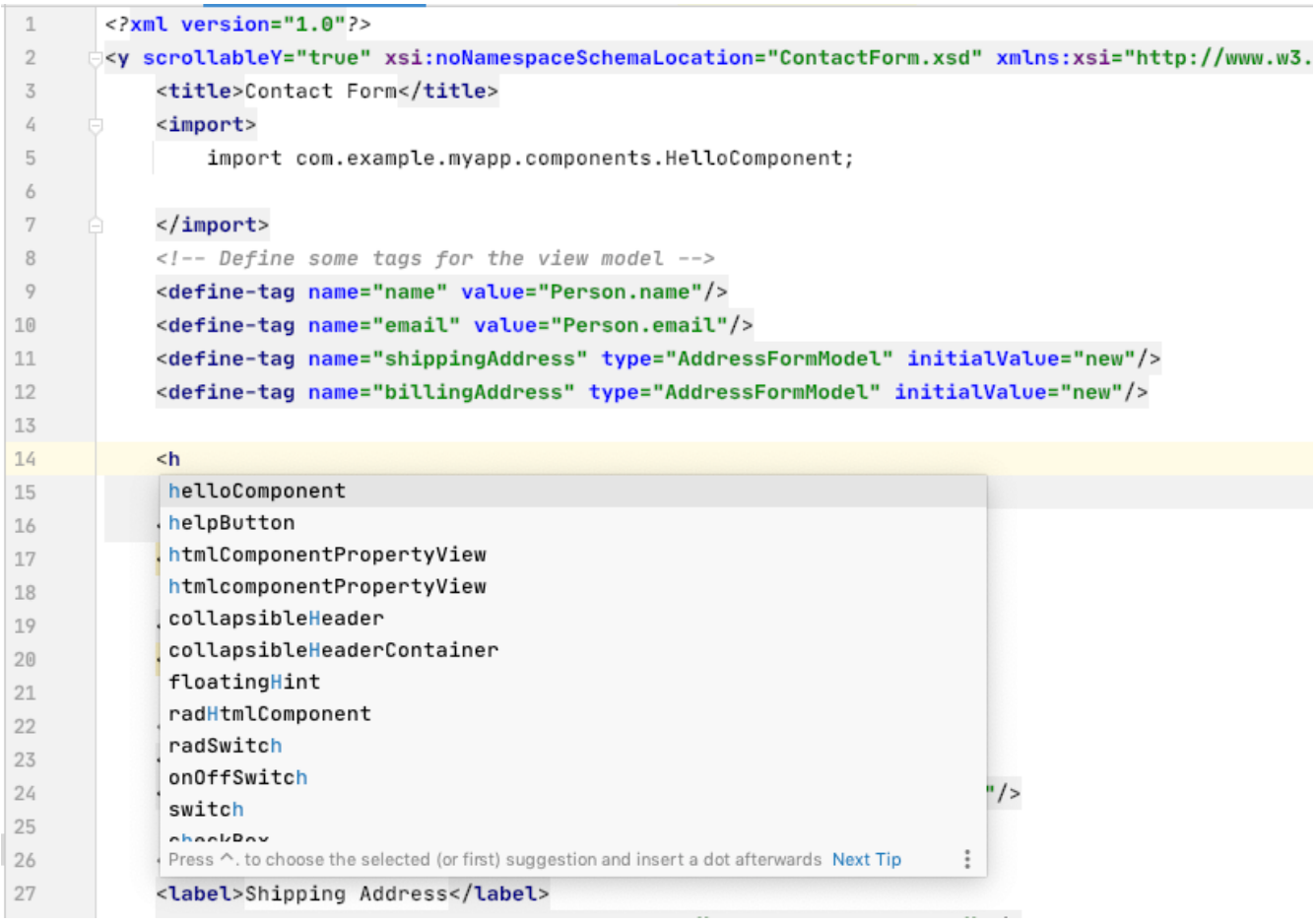
To demonstrate this, create a Java class named `HelloComponent` in the package `com.example.myapp.components`, with the following content:

```
package com.example.myapp.components;

import com.codename1.ui.Container;
import com.codename1.ui.Label;
import com.codename1.ui.layouts.BorderLayout;

/**
 * A custom component that displays a hello message.
 */
public class HelloComponent extends Container {
    private Label helloLabel = new Label();
    private String helloMessage;

    public HelloComponent() {
        super(new BorderLayout());
        add(BorderLayout.CENTER, helloLabel);
    }

    /**
     * Set the hello message to display.
     * @param helloMessage
     */
    public void setHelloMessage(String helloMessage) {
        this.helloMessage = helloMessage;
        helloLabel.setText("Hello " + helloMessage);
    }


    /**
     * Gets the Hello Message.
     * @return
     */
    public String getHelloMessage() {
        return helloMessage;
    }
}
```

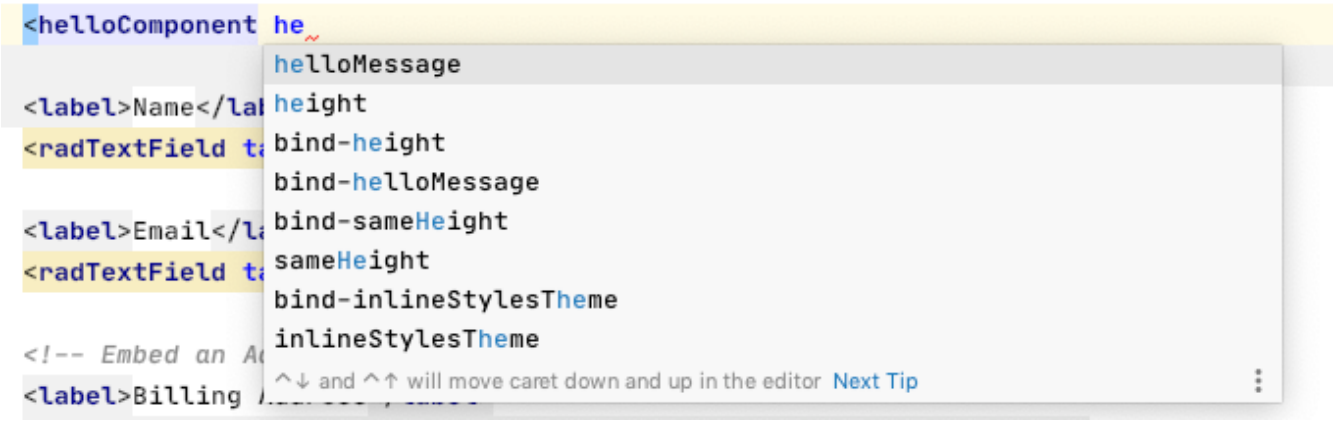
No open the contact form from the previous example and add an import statement for our new package:

```
<import>
    import com.example.myapp.components.HelloComponent;
</import>
```

If you save the file, it will automatically recompile the XML schema so that the `<helloComponent>` tag will be available for type hinting/autocompletion in a few moments.



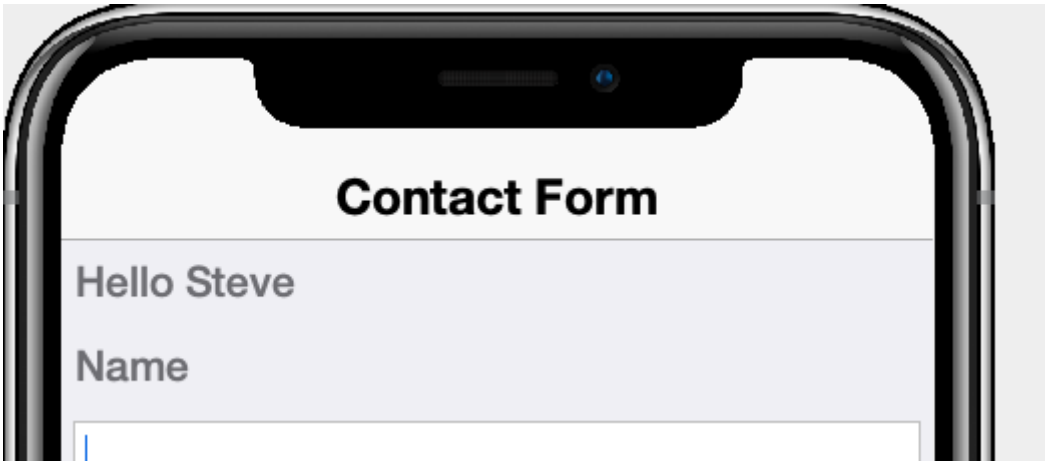
Add the `helloMessage` attribute to set the message in our component. Notice that IntelliJ provides type hints for this property also. It automatically picks up all the setters and getters for your class and converts them into XML attributes.



Let's add an instance that says "Hello Steve":

```
<helloComponent helloMessage="Steve"/>
```

The result:



2.16.1. EntityView and PropertyView

If your component is meant to "bind" to a model, then you should consider extending either `AbstractEntityView` or `PropertyView`, as these include built-in support for binding to entities. If your component is meant to be a view for a single property, then `PropertyView` would make sense. If, however, it is meant to bind to multiple properties on a model, then you should extend `AbstractEntityView`. As we've seen before, in Views within Views, you could create an *EntityView* entirely in XML, RAD views *do* get compiled to subclasses of `AbstractEntityView`. The choice is yours.

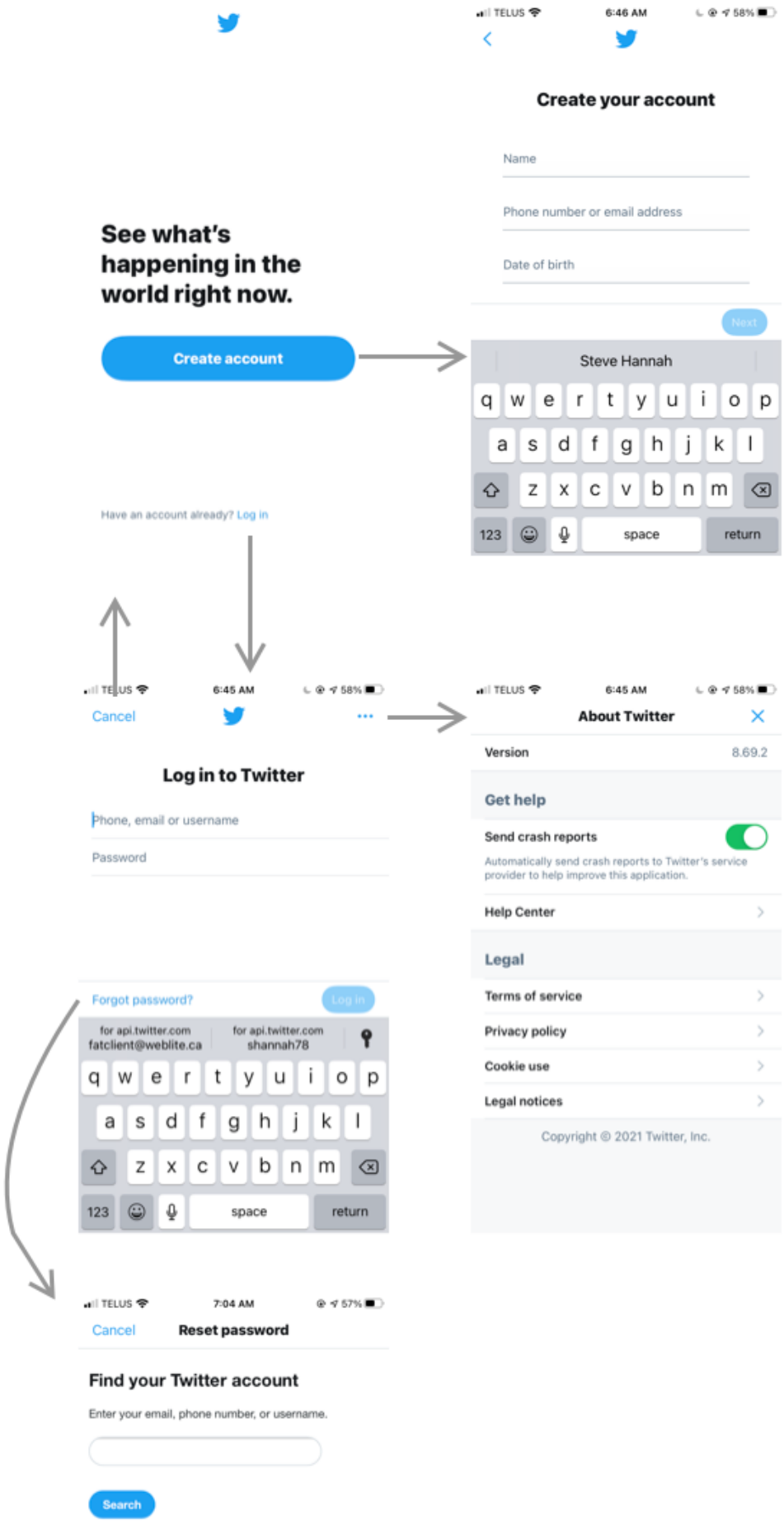
Chapter 3. App Example 1: A Twitter Clone

In Getting Started we covered the basics of CodeRAD. In this chapter we'll put our knowledge to the test as we build a clone of the Twitter mobile app.

As Julie Andrews says, let's start at the very beginning: The start page and signup/signing workflow. Below are some screenshots from the actual Twitter iOS app running on my iPhone 8.

NOTE

In this chapter we will replicate these forms and workflow using CodeRAD. This tutorial will focus only on the *client* portion of the app. Where the *real* app would require server interaction (e.g. login and registration), we will create mock implementations that run on the client. These mock implementations can be extended to support your preferred server-side technology, but that is beyond the scope of this tutorial.


See what's happening in the world right now.
Create account
Have an account already? Log in
Create your account
Name
Phone number or email address
Date of birth
Next
Steve Hannah
Cancel
Log in to Twitter
Phone, email or username
Password
Forgot password?
Log in
About Twitter
Version
8.69.2
Get help
Send crash reports
Automatically send crash reports to Twitter's service provider to help improve this application.
Help Center
Legal
Terms of service
Privacy policy
Cookie use
Legal notices
Copyright © 2021 Twitter, Inc.
Cancel
Reset password
Find your Twitter account
Enter your email, phone number, or username.
Search

3.1. Creating the Project

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

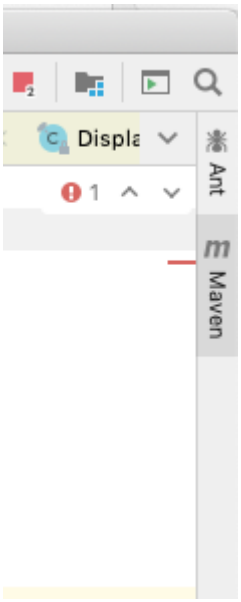
Extract the .zip file and then open the project in IntelliJ IDEA

TIP	This is the same starter project that we used in the Getting Started tutorial. I recommend you go through the Getting Started tutorial before starting this one as it will provide you with the fundamentals required to move forward.

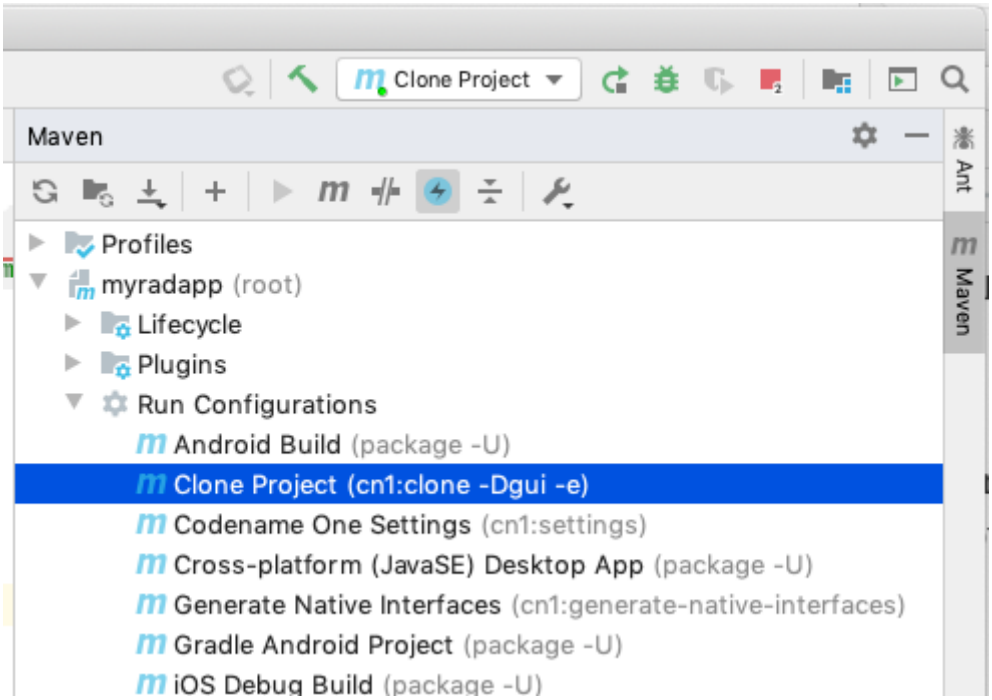
The starter project comes preconfigured with placeholders for the maven `groupId` and `artifactId` properties.

Before we begin, let's change the `groupId` and `archetypeId` to better reflect our application. We'll do this by *cloning* the project using the `cn1:clone` maven goal.

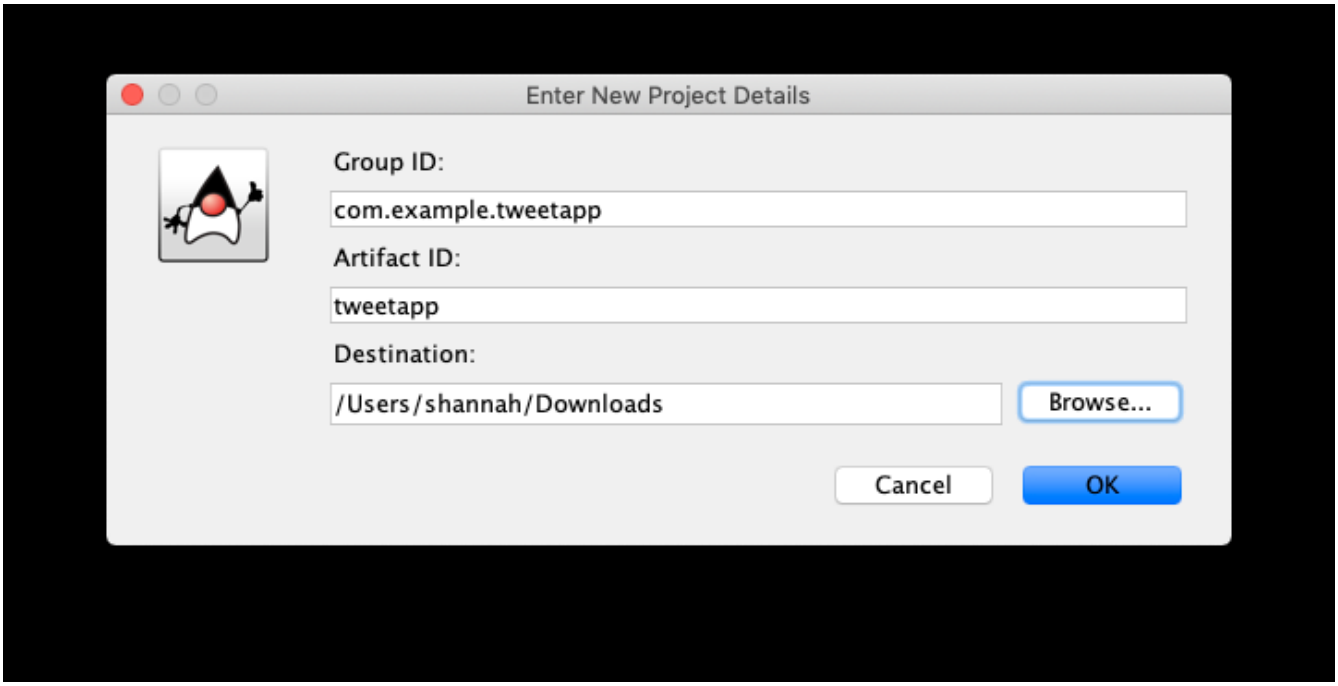
First expand the *Maven* side panel (if it isn't already expanded) by clicking on the "Maven" tab on the right side of the IntelliJ window.



Then expand the *myapp* > *Run Configuration* node, and double click the *Clone Project* option.



In the dialog prompt, enter the new *groupId* and *artifactId* for the project, and select a destination for the new project. My settings are shown in the screenshot below:



After you press *OK* it will create a new project for you at the location you provided. In my case, the project will be found at `$HOME/Downloads/tweetapp`.

Clone the current project, and then open the *cloned* project in IntelliJ.

We are now ready to proceed.

3.2. Creating the Views

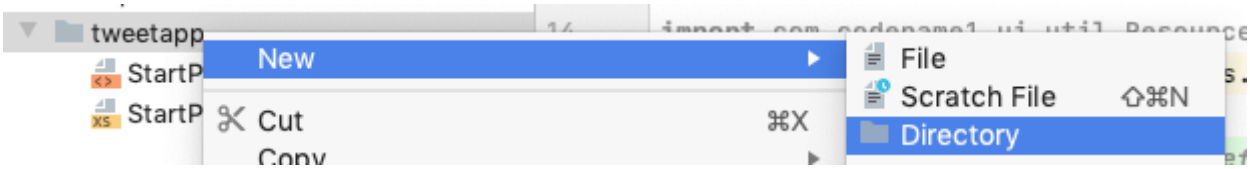
The flow-chart above shows five forms that we will be recreating:

1. Welcome Page
2. Signup Page
3. Signin Page
4. Forgot Password
5. About Twitter

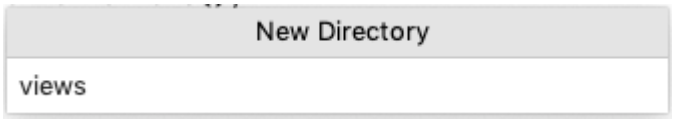
We'll begin by making empty views for each of these forms.

TIP All of the views for this project are located in the *common/src/main/rad/views* directory. See Under the Hood for a brief of the project layout.

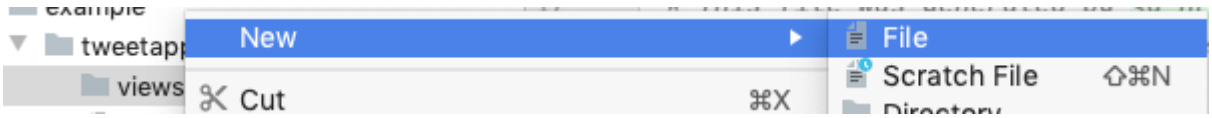
First let's create a new package for our views. Expand the *common/src/main/rad/views/com/example/tweetapp* nodes in the project inspector on the left. Then right-click the *tweetapp* node and select *New* > *Directory*



Enter "views" for the folder name, and press *Enter*.



Then right-click on this new *views* directory node, and select *New* > *File*:



Enter "WelcomePage.xml", in the prompt, and press *Enter*.

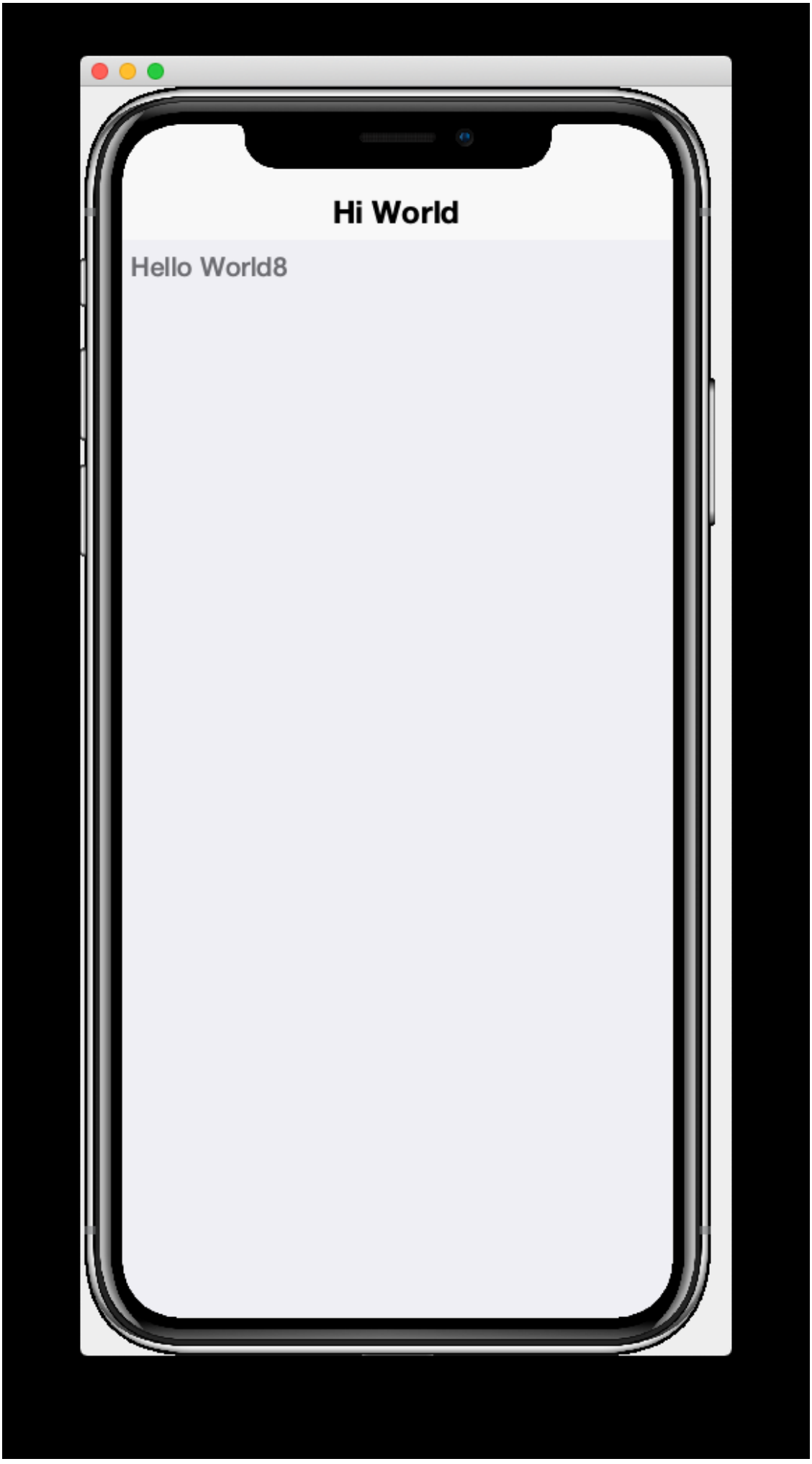
Now, open the *WelcomePage.xml* file, and change the contents to the following:

```
<?xml version="1.0"?>
<y>
    <label>Welcome Page</label>
</y>
```

Before proceeding, let's try running the project in the simulator to make sure that it compiles OK. Running the project will also run the CodeRAD annotation processor which will generate Java classes from our WelcomePage view so that they will be available for type hinting and auto-complete in IntelliJ.

Press the  button on the toolbar. It may take a while to run the first time as it might have to download dependencies. Subsequent runs should only take a few seconds.

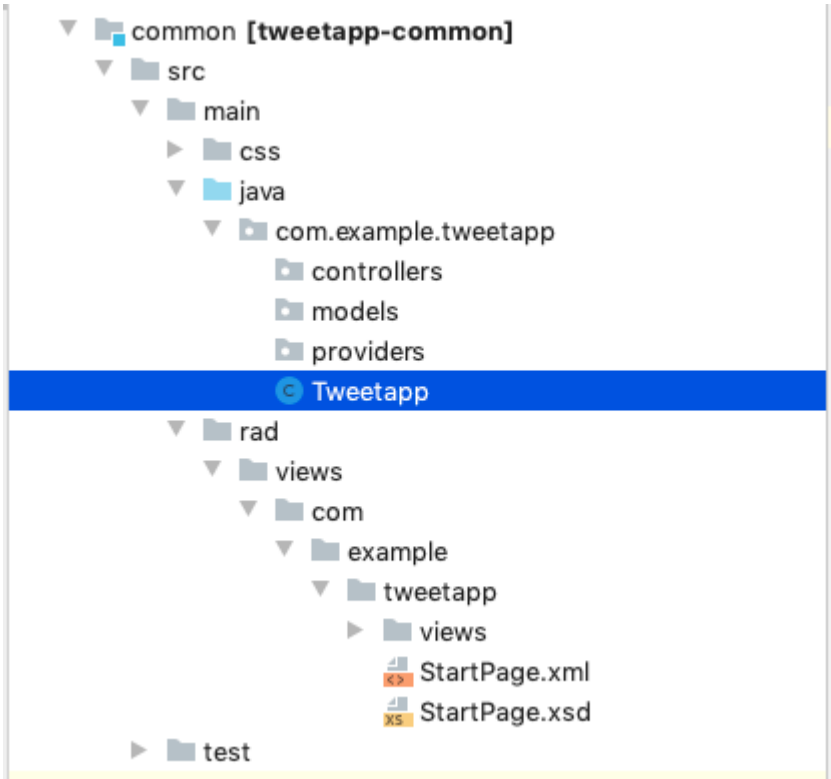
If all goes well, you should see the simulator pop up with a "Hi World" app as shown here:


Hi World
Hello World8

Notice that this isn't showing our *WelcomePage* yet. It is showing the default *StartPage* template from the template.

Let's modify the *ApplicationController* so that it shows our *WelcomePage* view as the default view.

The *ApplicationController* class for this application is defined in the *TweetApp* class inside the *com.example.tweetapp* package (inside the *src/main/java* directory).



Open this file and find the `actionPerformed()` method. It should look something like the following listing:

```
public void actionPerformed(ControllerEvent evt) {

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

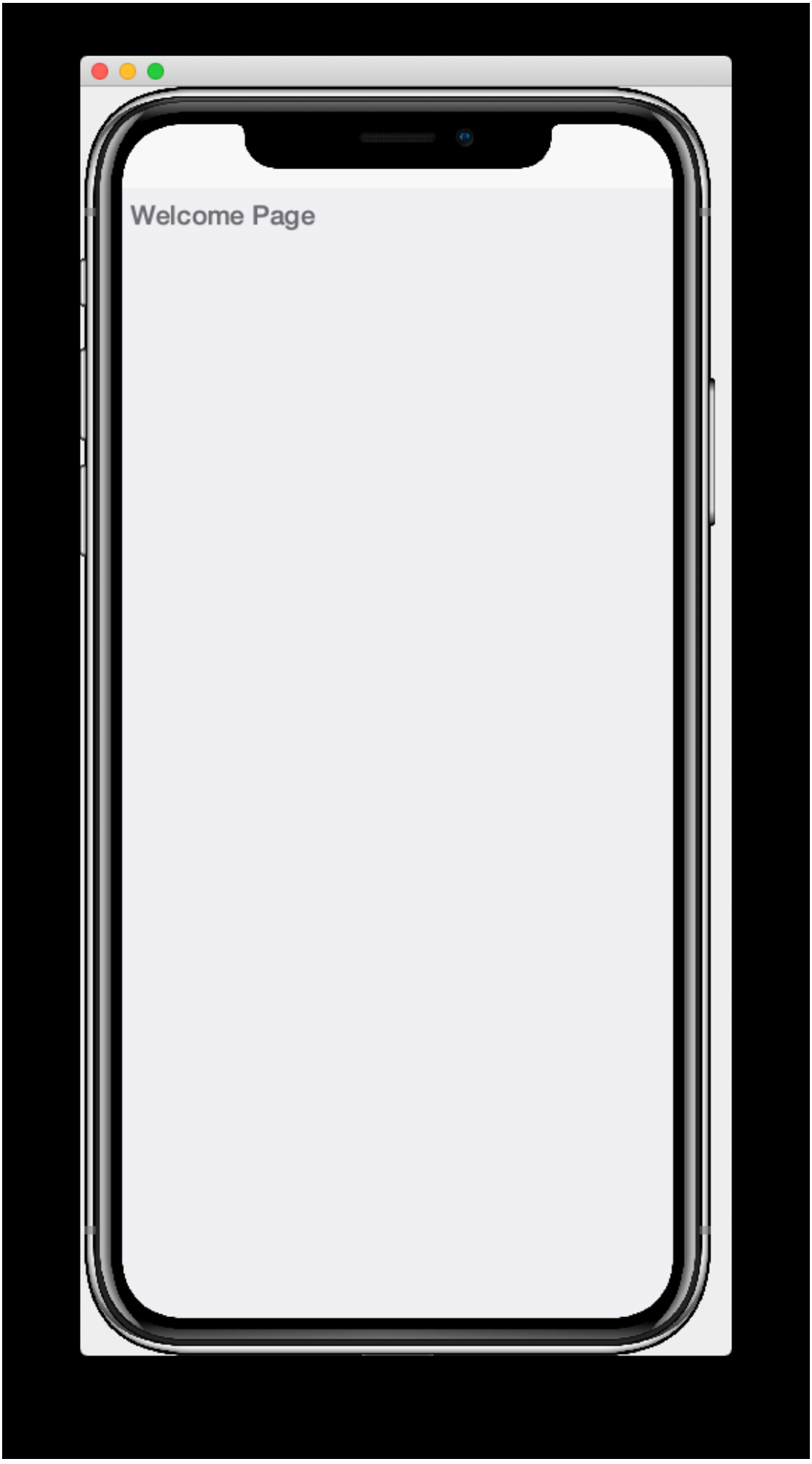
TIP The `actionPerformed()` method is triggered for every event that propagages up to the *ApplicationController*. You can monitor and handle many application events from inside this method.

Let's change the `new StartPageController(this).show()` call to

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

NOTE	The `WelcomePageController` class is a `FormController` subclass that is generated from the `WelcomePage.xml` view by the CodeRAD annotation processor. If you haven't built the project since creating the `WelcomePage.xml` file, then IntelliJ might complain that it can't find the class. Don't worry, about these warnings as they should "fix" themselves when you run or build the project.

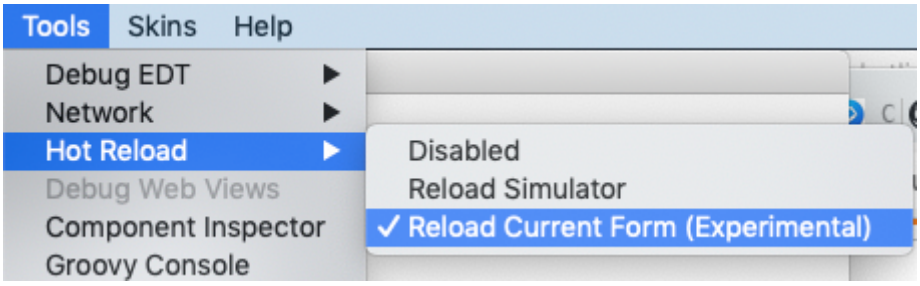
Now, if you restart the simulator, it should show our *WelcomePage* view.


Welcome Page

3.3. Hot Reload

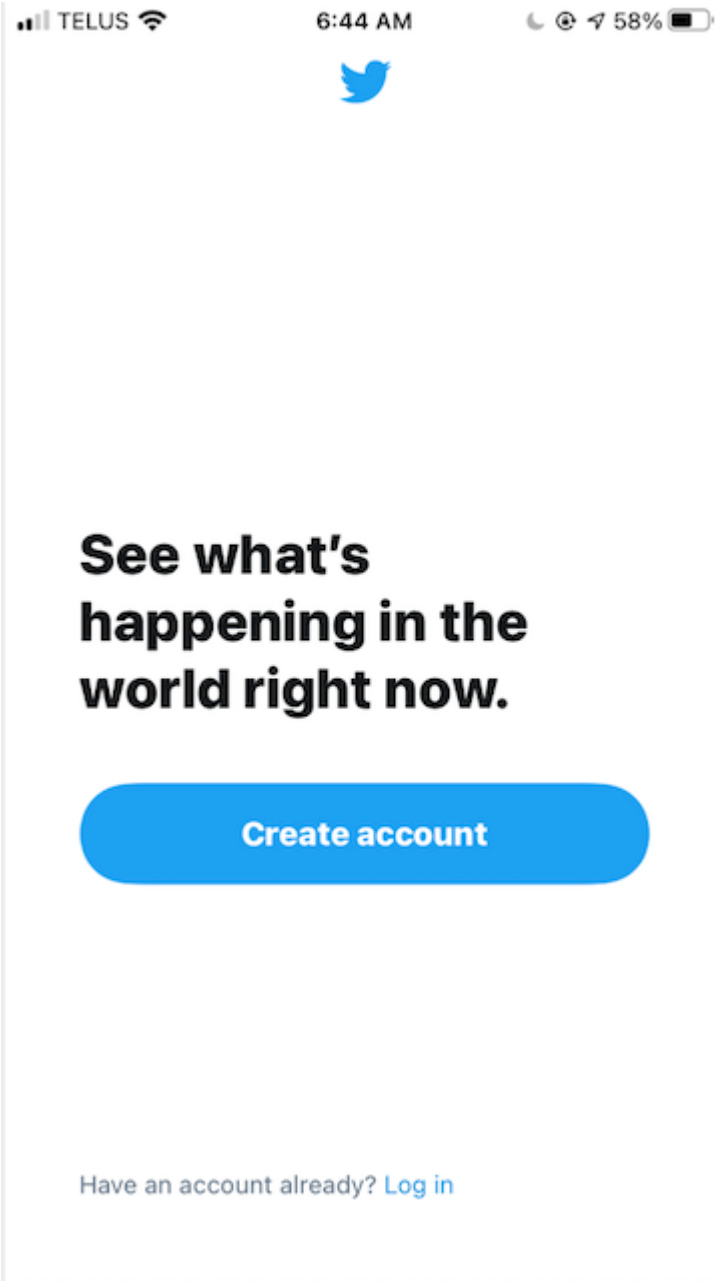
The Codename One simulator has a *Hot Reload* feature that can dramatically improve your development experience by reducing the turnaround time for testing changes to your source code. See Hot Reload for more information about this feature.

For most of this tutorial, I will be using the *Reload Current Form* setting of Hot reload so that the simulator will automatically reload the current form after I make changes to the source.



3.4. The Welcome Page

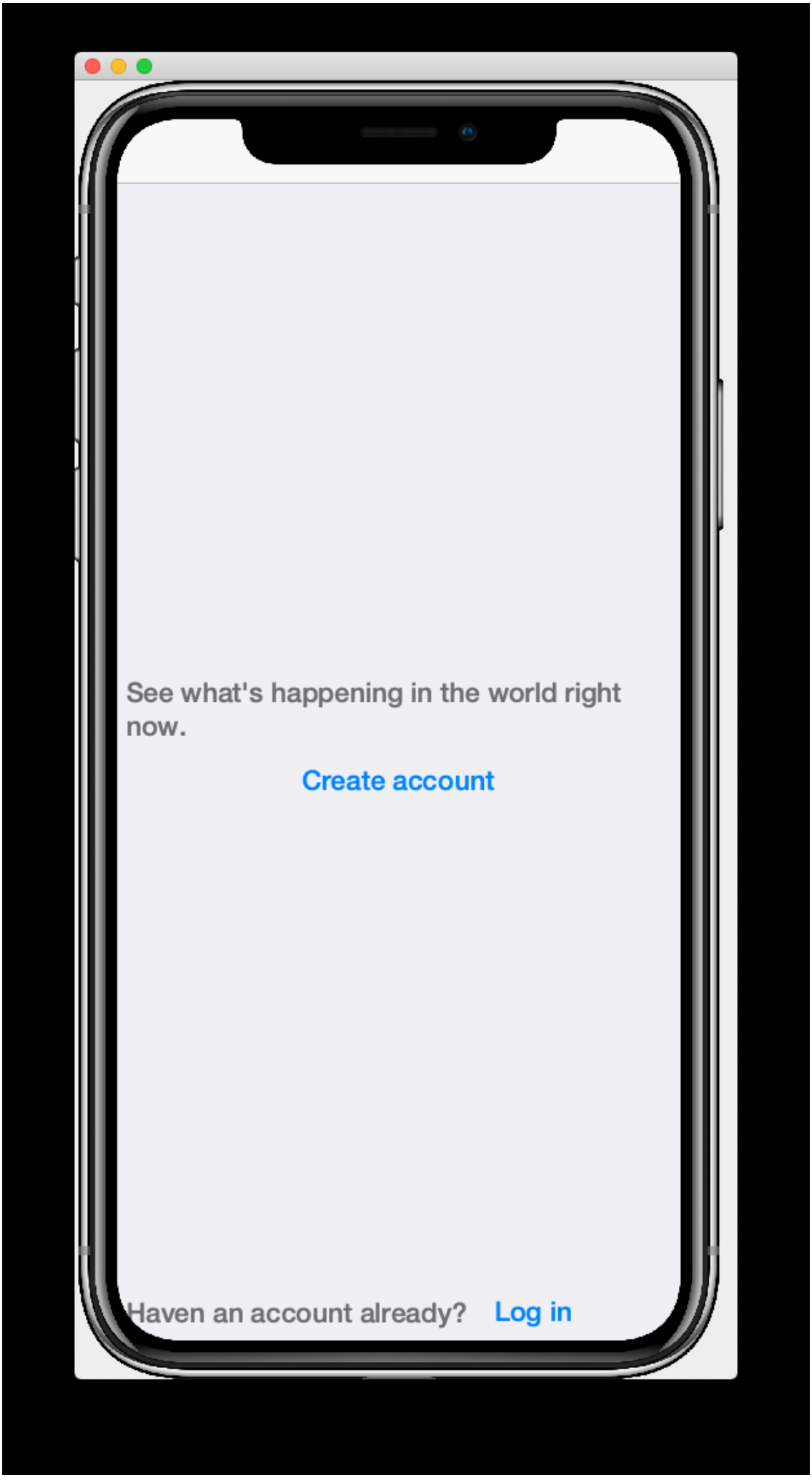
Our welcome page is currently just a placeholder that says "Welcome". Let’s change it to resemble the Twitter welcome page as shown below:



In order to replicate this content and structure, add the following to the *WelcomePage.xml* file:

```
<?xml version="1.0"?>
<borderAbsolute
        xsi:noNamespaceSchemaLocation="WelcomePage.xsd" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
    <y layout-constraint="center">
        <spanLabel>See what's happening in the world right now.</spanLabel>
        <button>Create account</button>
    </y>

    <flow layout-constraint="south">
        <label>Haven an account already?</label>
        <button>Log in</button>
    </flow>
</borderAbsolute>
```

Nothing fancy here. I'm just trying to *roughly* replicate how the form is laid out using Codename One's layout managers. Now reload the simulator (if you have *Hot Reload* enabled, then the simulator will reload automatically), and you'll see something that looks like:


See what's happening in the world right now.
Create account
Haven an account already?
Log in

Now that the structure is there, let's work on the style.

Let's start with the *Create Account* button. According a web search, the *Twitter Blue* color is `#1DA1F2`, so let's make the button background this *Twitter Blue* and the foreground color white. We'll do this by creating a UIID named `TwitterButton` in our stylesheet.

3.4.1. The *Create Account* Button

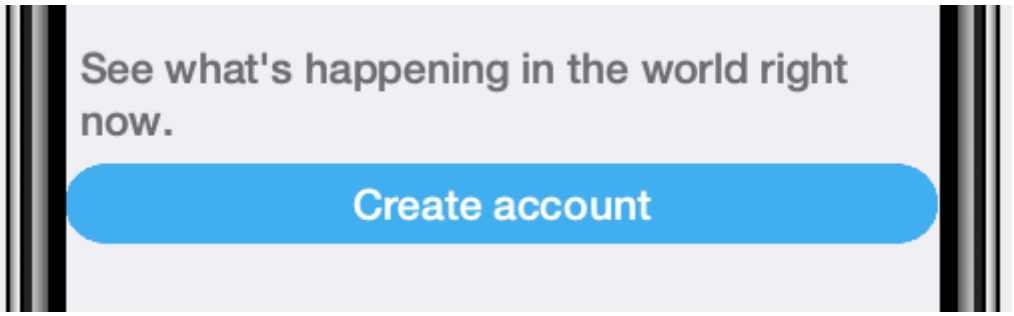
Open the stylesheet (located at *src/main/css/theme.css*) and add the following:

```
TwitterButton {
    cn1-derive: Button;
    background-color: #55acee;
    color: white;
    border: cn1-pill-border;
}
```

And add `uiid="TwitterButton"` to the *Create Account* button:

```
<button uiid="TwitterButton">Create account</button>
```

You should see the simulator update within a couple seconds to show you the result of this change:



This is getting closer, but the button needs a bit more padding.

Just eye-balling it, I'd say the button has about an equal amount of padding as the text size. So we'll try padding of `1rem`.

After some trial and error, I found that the it looks best with a padding of 0.7rem.

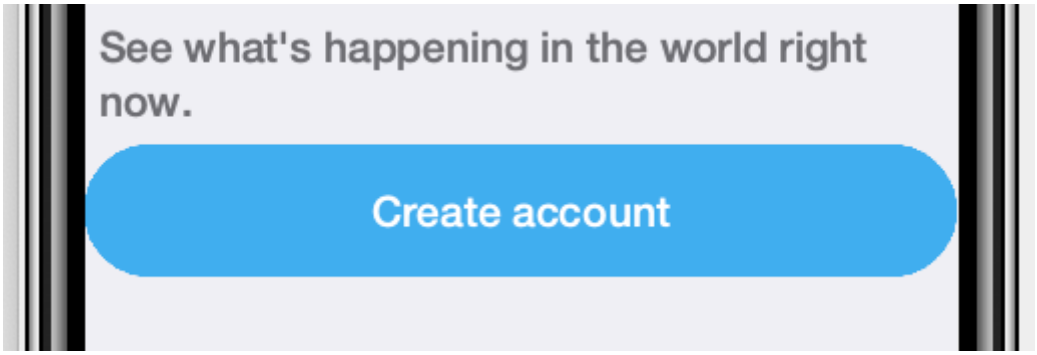
TIP

The `rem` unit corresponds to the height of the default system font. You can also use other units such as `mm` (millimetres), `pt` (points = 1/72nd of an inch), `px` = pixels, `vh` = percent of the display height, `vw` = percent of the display width, `vmin` = percent of the minimum of the display height and width, or `vmax` = the percent of the maxium of the display height and width.

So our CSS becomes:

```
TwitterButton {
    cn1-derive: Button;
    background-color: #1DA1F2;
    color: white;
    border: cn1-pill-border;
    padding: 0.7rem;
}
```

And the result looks like the following:



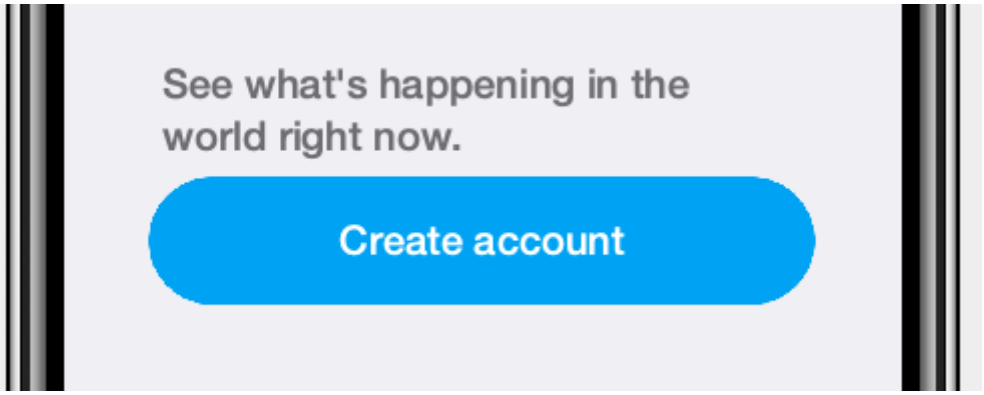
3.4.2. The Form Padding

We also need to add some padding to the form to match the design. Again, I'm eye-balling it, but it looks like their form has about 10% of the display width.

Create a new UIID in the stylesheet and call it TwitterContentPanem as follows:

```
TwitterContentPane {
    padding:10vw;
}
```

You should see the result instantly in the simulator:



It's getting closer. The font isn't exactly right (I'm just using the defualt font right now), but that's OK. We can circle back and refine the fonts later.

3.4.3. The Heading Text

The next obvious thing is the *See what's happening...* text. This needs to be larger and black.

I'll create a style named *TwitterHeading1* for this style. Set this style as the `textUIID` attribute on the `<spanLabel>` tag:

```
<spanLabel textUIID="TwitterHeading1">See what's happening in the world right
now.</spanLabel>
```

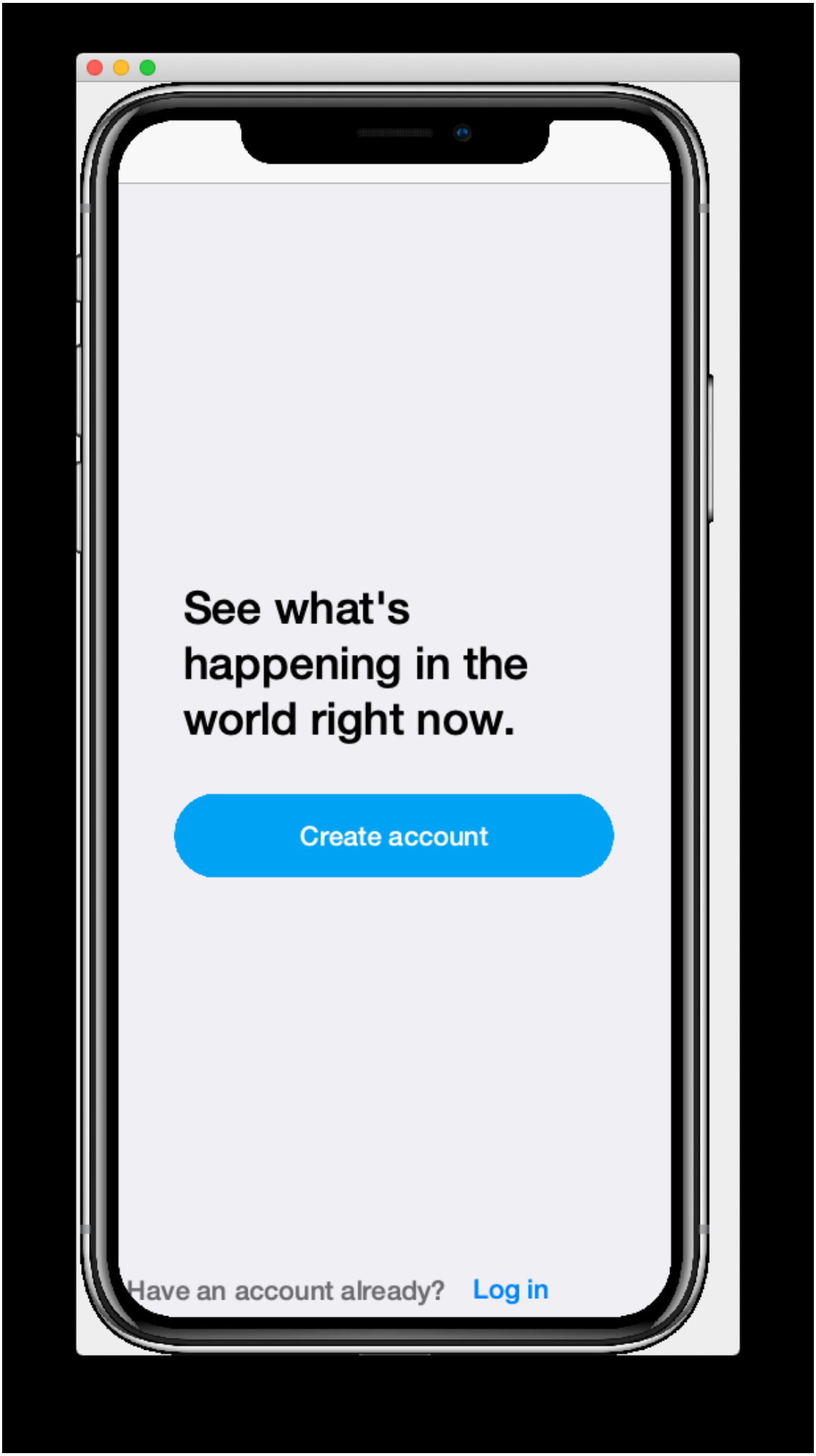
NOTE	We set the `textUIID` attribute instead of the usual `uiid` attribute because the *SpanLabel* component is a compound component that contains an inner component for rendering the text. The `uiid` attribute, in this case, only pertains to the outer `SpanLabel` container - so things like borders, backgrounds, and padding, will work as expected there - but the *font* needs to be applied to the inner component.

And add this style to the stylesheet:

```
TwitterHeading1 {
    font-size: 1.2rem;
    color: black;
    margin-bottom: 1rem;
}
```

These sizes and margins were arrived at by trial and error.

According to the simulator, we're getting closer to our destination:


See what's happening in the world right now.
Create account
Have an account already? Log in

3.4.4. The Login Link

The footer text and login link are currently too big, and require some padding. They also highlight a problem that we will face when app is displayed on a phone that has rounded corners and notches, like the iPhone X.

We'll add the `safeArea="true"` attribute on the view's container to ensure that it provides enough padding so that its contents don't get clipped by the corners and notches.

```
<borderAbsolute safeArea="true" ...>
```

We'll also add some styles for the bottom labels and links, we'll call them, `TwitterSmallLabel` and `TwitterSmallLink` respectively.

```
TwitterSmallLabel {
    cn1-derive: Label;
    font-size: 0.5rem;
    padding:0;
    margin:0;
    color: #66757f;
    margin-right: 1mm;
}

TwitterSmallLink {
    cn1-derive: Button;
    font-size: 0.5rem;
    padding:0;
    margin:0;
    color: #1DA1F2;
}
```

These values were arrived at via trial-and-error, per usual.

NOTE The `cn1-derive` directive means that this style inherits all of the styles from the given style. E.g. `TwitterSmallLabel` *extends* the `Label` style, which is defined in the native theme for the platform.

3.4.5. Hiding the Title Area

The design doesn't include a typical title bar, but our view currently displays a small white area across the top of the form that from the title that we aren't using. We can hide this title area by adding:

```
<title hidden="true"/>
```

3.4.6. A few Inline Styles

For the remaining styles that require tweaking, I'm not going to create UIIDs for. I'll just add them directly the view XML.

We'll add some padding to the south container to match the padding in the center container (`10vw`).

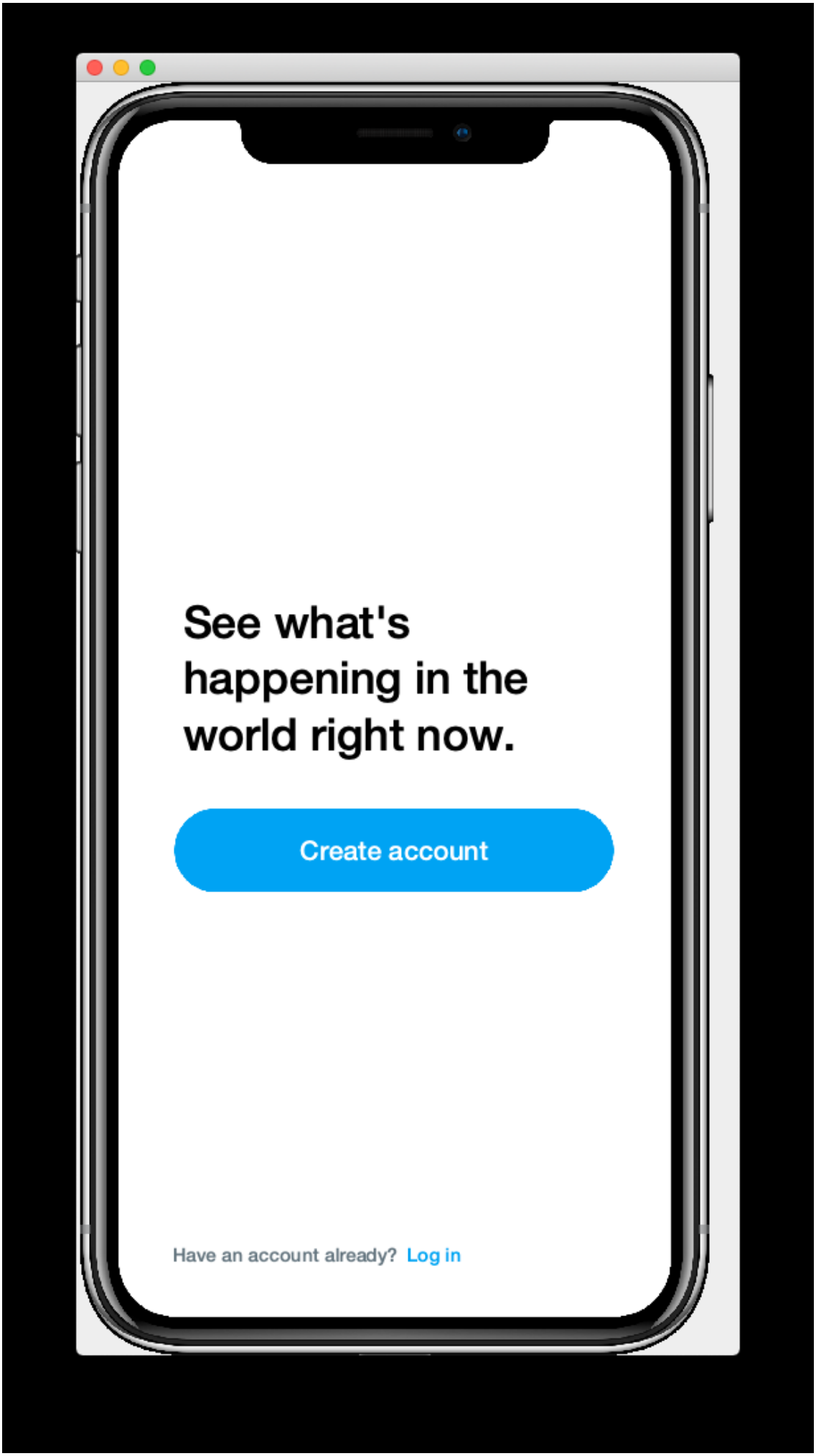
```
<flow style.paddingLeft="10vw" style.paddingRight="10vw" layout-constraint="south">
...
```

And we'll set the form background color to white:

```
<borderAbsolute
        safeArea="true"
        style.bgColor="0xffffff" style.bgTransparency="0xff" ... >
```

NOTE	We set both `style.bgColor` *and* `style.bgTransparency` to ensure that the background color is rendered. If we only set the `style.bgColor` but the native theme set the container to be transparent, then the backgroun color wouldn't be visible.

The result so far:


See what's happening in the world right now.
Create account
Have an account already? Log in

3.4.7. The Title Bar Icon

While this view doesn't have a conventional title bar, it does display the *Twitter* icon in the *title* position at the top of the form. Rather than copy the *real* twitter icon I had our designer make up a custom icon for our tweet app:

At my request, he wrapped this icon in a TTF file so that it can be used as an icon font within my application.

> **TIP** My designer used IcoMoon to convert his vector image into a truetype font.

TODO: Add a link to download the .ttf font

To use this font, I created a *fonts* directory inside the *css* directory, and copied the font (named *icomoon.ttf*) there, so that the font is located at *src/main/css/fonts/icomoon.ttf*.

To use this font in the app, I need to add a `@font-face` directive for the font inside the stylesheet as follows:

```
@font-face {
  font-family: 'icomoon';
  src: url('fonts/icomoon.ttf');
}
```

I also need to create a style that uses this font:

```
TwitterIcon {
    font-family: icomoon;
    font-size: 1.4rem;
    color: #1DA1F2;
}
```

Now, I can finally add a label to my view that uses this icon font, as a means to display the icon.

```
<center layout-constraint="north">
    <label iconUIID="TwitterIcon" fontIcon="(char)0xe902" ></label>
</center>
```

NOTE	In this version it was necessary to cast the `0xe902` to `char` to avoid a compiler error. In future versions, this cast will no longer be required.

I use the `iconUIID` attribute to set the UIID of the label's icon so that it uses the our font icon. The `fontIcon` attribute specifies the character code of the glyph in the font to display. In this case it is the unicode character 0xe902, which I was able to extract from the files provided by IcoMoon.

3.4.8. Final Result

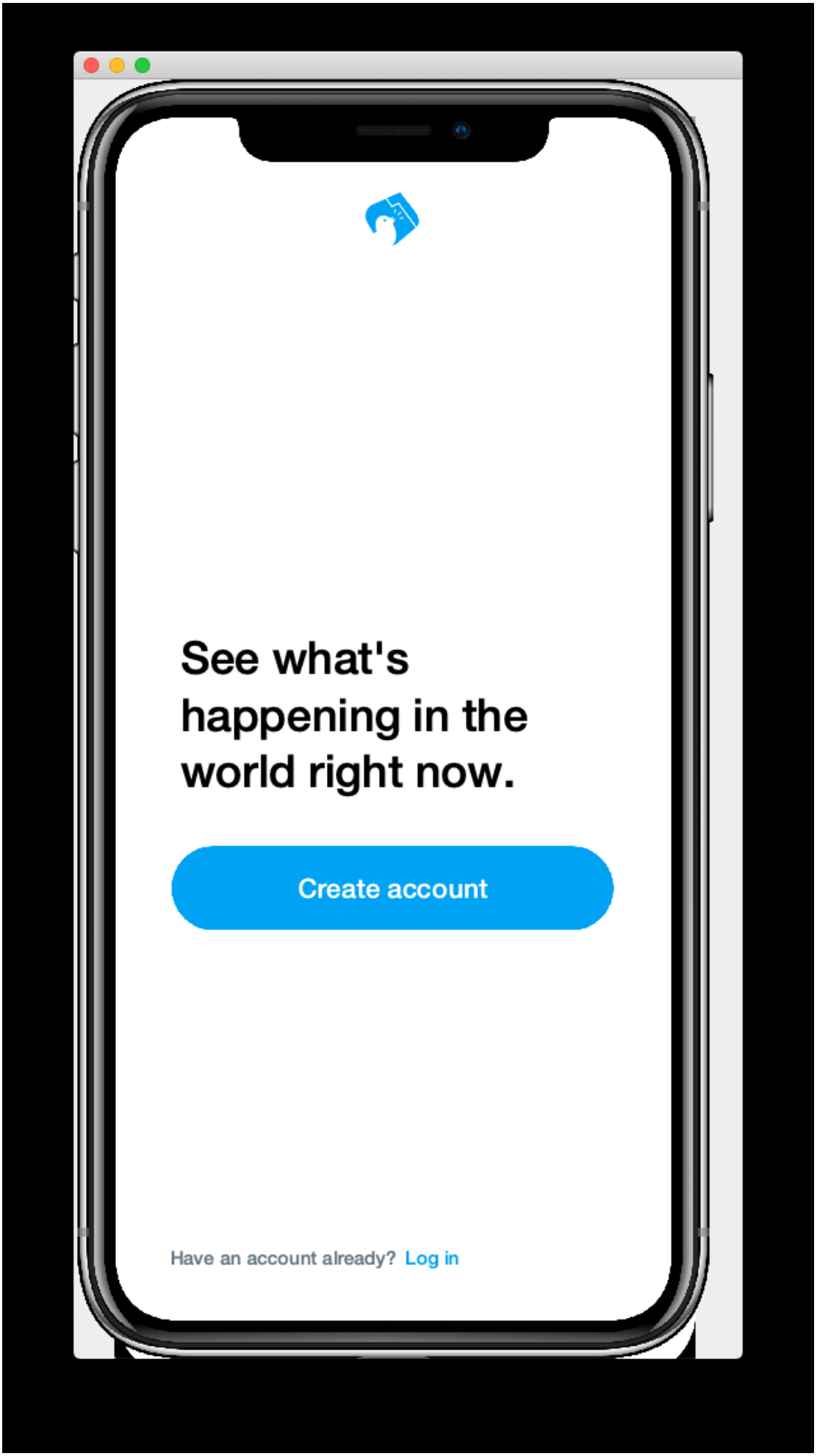

See what's happening in the world right now.
Create account
Have an account already? Log in

Figure 10. The final result of our WelcomePage

WelcomePage.xml

```
<?xml version="1.0"?>
<borderAbsolute
        safeArea="true"
        style.bgColor="0xffffff" style.bgTransparency="0xff"
        xsi:noNamespaceSchemaLocation="WelcomePage.xsd" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">

    <title hidden="true"/>
    <center layout-constraint="north">
        <label iconUIID="TwitterIcon" fontIcon="(char)0xe902" ></label>
    </center>

    <y layout-constraint="center" uiid="TwitterContentPane">
        <spanLabel textUIID="TwitterHeading1">See what's happening in the world right
now.</spanLabel>
        <button uiid="TwitterButton">Create account</button>

    </y>

    <flow style.paddingLeft="10vw" style.paddingRight="10vw" layout-constraint="south
">
        <label uiid="TwitterSmallLabel">Have an account already?</label>
        <button uiid="TwitterSmallLink">Log in</button>

    </flow>
</borderAbsolute>
```

theme.css

```
#Constants {
    includeNativeBool: true;
}

@font-face {
  font-family: 'icomoon';
  src: url('fonts/icomoon.ttf');
}

TwitterButton {
    cn1-derive: Button;
    background-color: #1DA1F2;
    color: white;
    border: cn1-pill-border;
    padding: 0.7rem;
}

TwitterContentPane {
    padding:10vw;
}

TwitterHeading1 {
    font-size: 1.2rem;
    color: black;
    margin-bottom: 1rem;
}

TwitterSmallLabel {
    cn1-derive: Label;
    font-size: 0.5rem;
    padding:0;
    margin:0;
    color: #66757f;
    margin-right: 1mm;
}

TwitterSmallLink {
    cn1-derive: Button;
    font-size: 0.5rem;
    padding:0;
    margin:0;
    color: #1DA1F2;
}

TwitterIcon {
    font-family: icomoon;
    font-size: 1.4rem;
    color: #1DA1F2;
}
```

Chapter 4. Controllers

4.1. Application Structure and Form Navigation Pre-CodeRAD

How do you structure your Codename One apps? How do you share data between different forms? How do you maintain state between forms? How do you manage navigation between forms? E.g. When the user presses "Back", how do you decide what to do?

Codename One is not opinionated on these questions. Essentially, you may structure your app however you like. The "old" GUI builder used a `StateMachine` as a monolithic controller that managed state and navigation. All forms used this single controller for all of their event handling and callbacks. This worked well for small to mid-sized apps, but on larger apps, it soon became necessary to refactor the logic into separate files.

The new GUI builder moved to a less opinionated approach. A GUI builder form would map to either a "Form" or a "Container". I.e. just the view. You were free to use these views in your application however you liked. You could structure your app however you liked.

The other day I was asked by a new developer "How do you push and pop forms?". The question seems to assume that there must be some sort of navigation stack for forms, but there isn't really. The Codename One API leaves you free to implement a navigation stack however you like. The `Form` API allows you to show a form with `show()` (and `showBack()` to show it with a reverse transition), but you are free to organize your form navigation stack however you like.

A common, though quick and dirty, approach is to grab a reference to the "current" form when you create a new form, and use this for your "back" command. E.g.

Sample code using the "current" form for the back command on a new form.

```
Form f2 = new Form("Result", new BorderLayout());
Form f = CN.getCurrentForm();
if (f != null) {
    f2.setBackCommand("Back", null, evt->{
        f.showBack();
    });
    f2.getToolbar().addCommandToLeftBar("Back", null, evt->{
        f.showBack();
    });
}
f2.show();
```

There's nothing wrong with this approach, other than the fact that it gets tedious doing this on every single form. We could reduce the boilerplate code here by abstracting this behaviour out into a separate class, so that we can "reuse" our solution on every form. Let's look at such an abstraction:

A very simple abstraction for "Back" functionality on forms.

```
public class MyFormController {
    private Form form;

    public MyFormController(Form form) {
        this.form = form;
        Form backForm = CN.getCurrentForm();
        if (backForm != null) {
            form.setBackCommand("Back", null, evt->{
                backForm.showBack();
            });
            form.getToolbar().addCommandToLeftBar("Back", null, evt->{
                backForm.showBack();
            });
        }
    }

    public void show() {
        form.show();
    }

    public void showBack() {
        form.showBack();
    }
}
```

What we've created here is a very simple controller, as in the "C" in MVC. This controller implements the "back" navigation for our forms, so that the form itself doesn't need to contain the logic itself.

4.2. App Structure as a Tree

Once you start walking down the path extracting navigation logic into a "controller" class, you can start to think about your application's structure more clearly. For example, we naturally think of UI component hierarchy as a tree-like structure, where the `Form` is the root, and the containers and components are the nodes. Thinking in these terms makes it easy to visualize the structure of a form:

1. Containers have child components
2. Components have a parent container
3. Events can propagate up from a child component, to its parent, and ultimately up to the root - the `Form` itself.
4. Containers can be laid out and painted, which will propagate layout and painting down to its children, and their children, all the way down to the leaves of the container's branch.
5. The state of a container can affect the state of its children. E.g.:
    a. Shifting a container 5 pixels to the right, will shift all children's absolute position 5 to the

right also.

b. Changing the alpha or transform setting while painting a container will affect the alpha or transform on its children.

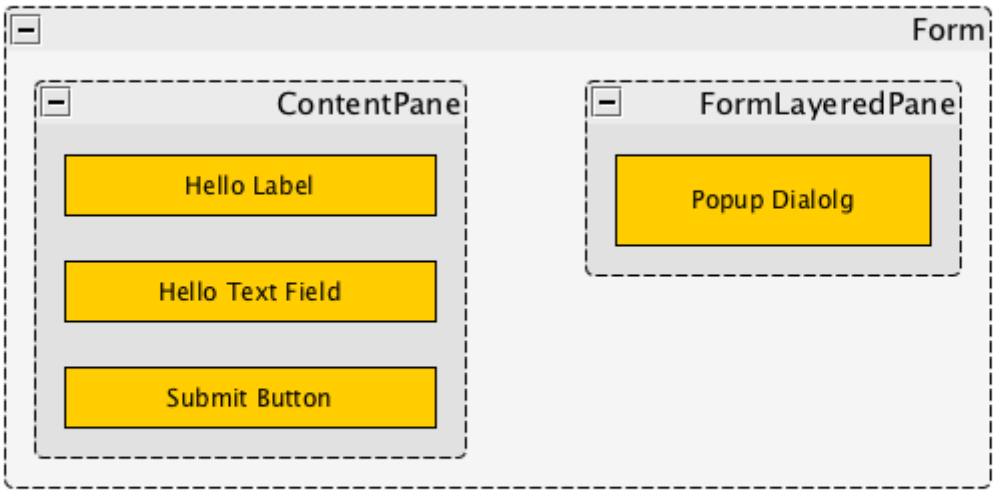


Figure 11. A simple Form UI presented here as a diagram that emphasizes containment. The next figure shows the same UI expressed in a tree diagram.

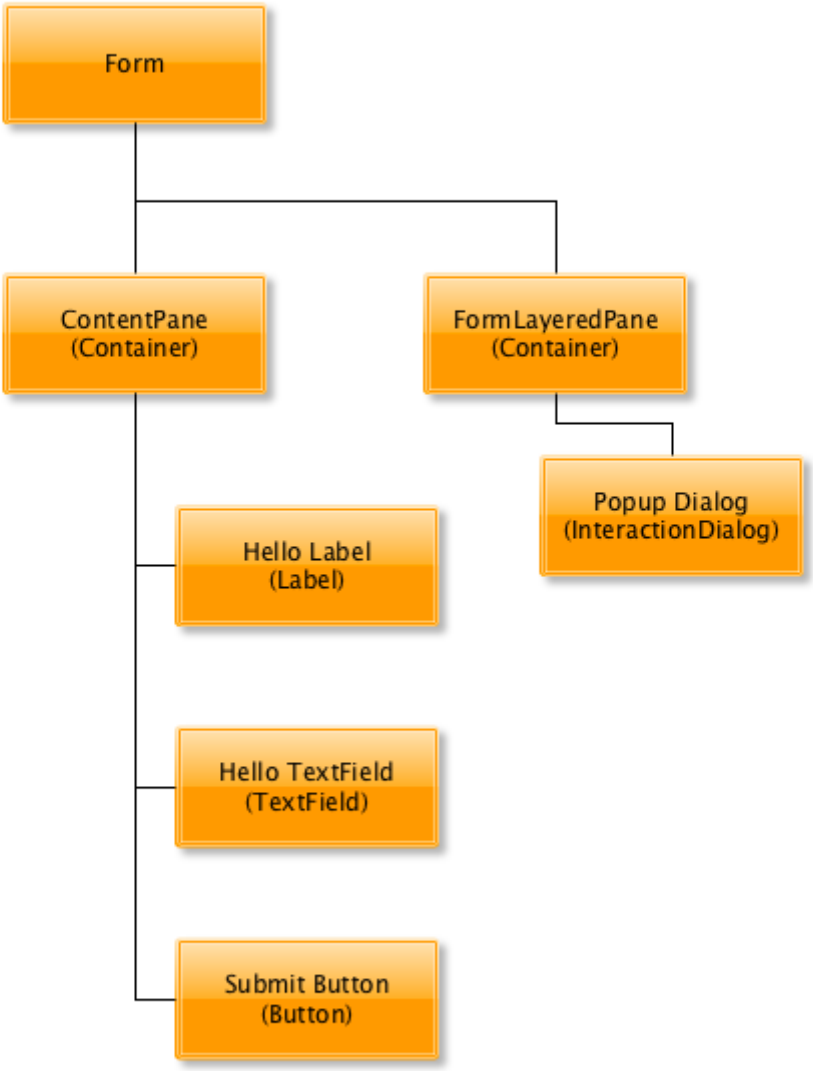


Figure 12. A UI component visualized as a tree, with the Form as the "root" node, each container a branch, and atomic components (e.g. Label, Button, TextField) as leaf nodes.

In fact, we derive many benefits from the fact that the UI can be mapped to a tree data structure. Trees are nice to work with because they lend themselves nicely to software. If you can write a function that operates on a single node of the tree, you can easily expand this function to operate on the entire tree.

Now, try to imagine your application as a tree structure. Each node of the tree is a controller - the root node is the "Application", and most of the other nodes are Forms. The "child" nodes of a Form node are those forms that you navigate to from a given form. This implies that the "Back" command for any given node will navigate back to the "parent" Form node.

In order to properly encapsulate all of this structure without it bleeding into the views themselves, we implement these nodes as "Controller" classes.

As an example, consider the Twitter mobile app. A few of its screens are shown below:

Figure 13. Screenshots from the Twitter mobile app.

The app structure of Twitter has 4 top-level FormControllers corresponding to the tabs at the bottom of the app: Home, Search, Alerys, and Inbox. The "Home" tab contains a list of tweets, and clicking on any of these tweets will bring the user to a "Tweet Details" form. Hence in the controller hierarchy, the "Home" tab is a top-level form (a child of the ApplicationController), and it has at least one child, the "Tweet Details" form controller.

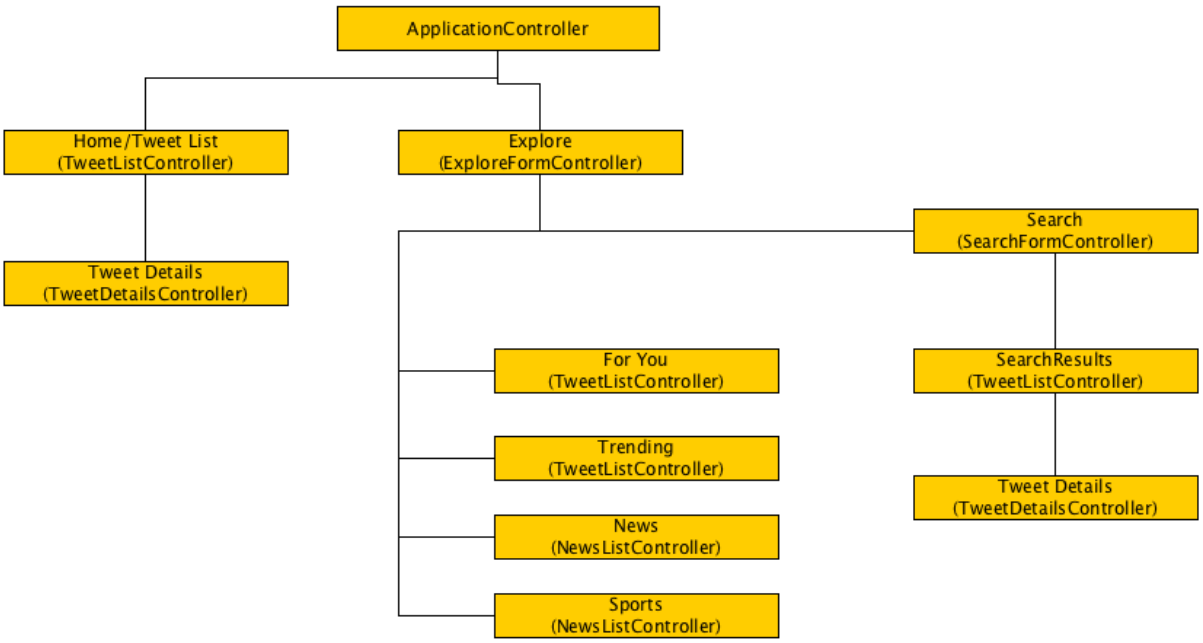


Figure 14. A partial controller hierarchy for the Twitter mobile app. **Note that this is not a class hierarchy**, *as many of these nodes are instances of the same class. It reflects the structure of the app in the sense that a node A is a child of node "B" if the user can navigate from node "A" to node "B".*

4.3. CodeRAD Core Controllers

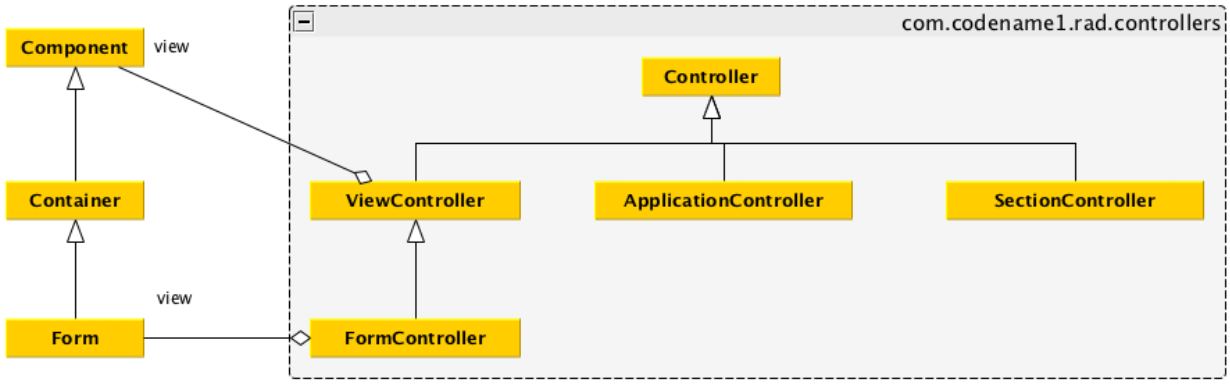


Figure 15. The main controller classes provided by CodeRAD.

CodeRAD provides a set of fundamental controller classes to help structure your app in this way:

1. `ApplicationController` - The root application controller.
2. `SectionController` - A controller for a section of the app. This can be a useful parent node to a set of related form nodes, to help group them together.
3. `FormController` - A controller for a single form.
4. `ViewController` - A controller for a single Component on a form. (This is a superclass of `FormController`, and in most cases `ViewController` on sub-components of a form aren't necessary. Only in complex cases).

4.4. Event Propagation

Mapping our app structure onto a controller tree brings other benefits as well. Just as we can propagate events up a UI component hierarchy for handling. We can propagate events up a controller hierarchy as well. In GUI applications, events are a very effective way of passing information between different parts of the application without introducing coupling.

The `Controller` class implements `ActionListener`, and all `ControllerEvents` that are fired in the UI, will be dispatched to the nearest `ViewController`, and then propagated up the controller hierarchy until it is consumed by a `Controller`, or until it reaches the root controller. Additionally, `Controller` implements `ActionSource` (i.e. implements `addActionListener()`) so that interested parties can subscribe to receive notifications about these controller events.

Event propagation is used to great effect internally in the CodeRAD library for such things as "Back" events, but it can and should also be utilized by application code to help reduce coupling and keep code clean.

4.5. Code-sharing / Lookups

CodeRAD controllers also exploit their tree structure to facilitate sharing of data from a "parent" controller to its children via the "lookup" mechanism. For example, perhaps you need to keep track of the current logged-in user for your app. One way to do it is to use static globals (this includes using a Singleton). But a more elegant approach is to use lookups in your ApplicationController, which are available to be "looked up" from all of its children.

E.g.

```
//.. somewhere in the application controller
UserProfile loggedInUser = getLoggedInUser();
addLookup(loggedInUser);

...

//.. in a child controller
UserProfile loggedInUser = lookup(UserProfile.class);
```

The `lookup()` method exploits the tree structure of the controller hierarchy by checking the current controller for an object of type `UserProfile.class`, and if it doesn't exist, it checks the parent controller. And so on until it reaches the root.

This allows us to completely encapsulate all functionality related to the logged in user in the ApplicationController, or a specific sub-controller, and it is made available to all parts of the app.

4.6. Example Controllers

NOTE	The following examples all use an ApplicationController as the root of the controller hierarchy, but it is also possible to use a controller in isolation. You just won't receive the full benefits of an application-wide controller hierarchy.

The easiest way to implement an application controller in your app is for your app's main lifecycle class (i.e. the class with `init()`, `start()`, `stop()` and `destroy()`) to extend `ApplicationController`. The `ApplicationController` class implements all of the lifecycle methods with sensible defaults. Therefore, you can simply override the start() method as shown below:

MyApplication.java - A minimal application controller. This controller overrides the `start()` method and instantiates a `FormController`, then shows it.

```
public class MyApplication extends ApplicationController {
    @Override
    public void start() {
        super.start();
        new MainFormController(this).show();
    }
}
```

MainFormController.java - A minimal FormController.

```
public class MainFormController extends FormController {
    public MainFormController(Controller parent) {
        super(parent);
        Form f = new Form("Hello");
        f.add(new Label("Hello World"));
        setView(f);
    }
}
```

4.7. Form Navigation

As we've already seen, "back" functionality is handled automatically by the `FormController`. If there is a "parent" FormController of the form, then it will add a "back" button to the toolbar and set a "back" command to go to that toolbar. However, you can also explicitly trigger a "back" event by firing a `FormBackEvent` (an internal class in `FormController`).

For example, we might want to create a reusable view called "BackButton", which is just a Button, that, when pressed, will fire a "FormBackEvent".

View that fires a `FormBackEvent`. This view can be added anywhere in the UI, and clicking it cause the app to navigate back to the "parent" (previous) form.

```
class BackButton extends Button {
    public BackButton() {
        super("Go Back");
        addActionListener(evt->{
            evt.consume();
            ActionSupport.dispatchEvent(new FormBackEvent(this));
        });
    }
}
```

This `BackButton` can be added anywhere in the UI. You don't need to write any "handling" code to catch this `FormBackEvent` because it will propagate up the component hierarchy until it finds the nearest ViewController, then continue to propagate up to the FormController, which will consume the event.

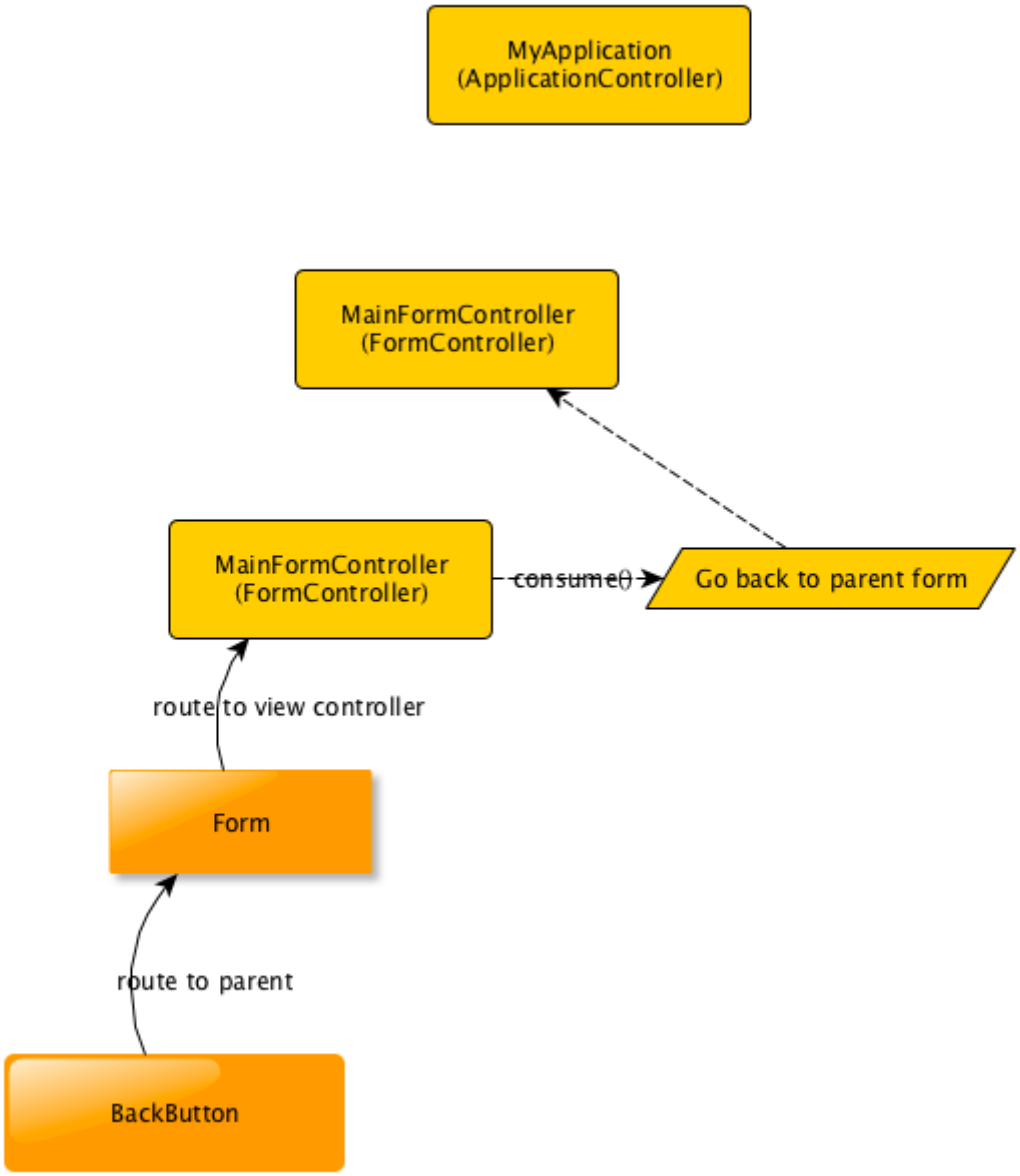


Figure 16. The flow of the `FormBackEvent` *as it propagates up to the first FormController, which consumes the event, and navigates back to the previous/parent form.*

Chapter 5. Serialization: Working with XML and JSON

CodeRAD entities' loose coupling presents advantages beyond the user interface. Just as you can loosely bind the properties of an entity to a UI View, you can also bind them to elements of and XML or JSON document. The `ResultParser` class provides a clean and succinct syntax for parsing XML and JSON into entity graphs.

In addition to the ResultParser, CodeRAD provides some convenient utility classes for parsing and querying JSON and XML documents. This chapter will introduce you to some of these utilities, and provide concrete examples of how they can be used to load data from a JSON or XML web service.

5.1. Useful Classes

Some of the more useful classes related to XML and JSON parsing are as:

`com.codename1.rad.processing.Result`

A derivation of the core Codename One Result class, which provides an expression language similar to XPath for querying XML and JSON documents. The CodeRAD version of this class fixes a number of bugs, and expands the expression capability with support for additional expressions. The decision was made to "fork" the Codename One class, rather than fix bugs because legacy applications may be relying on those "bugs", so changing its behaviour would have been too risky.

`com.codename1.rad.io.ElementSelector`

A utility for querying XML documents using syntax similar to CSS selectors. This class works similarly to the ComponentSelector class, except that this is optimized to work with XML elements rather than Codename One components.

`com.codename1.rad.io.ResultParser`

A class that is able to parse XML and JSON documents into entity graphs. This uses the `Result` class' expression language (similar to XPath syntax) for mapping properties of XML tags and JSON data structures, onto Entity properties.

These classes complement the existing data parsing classes that already exist in the Codename One ecosystem, such as `XMLParser`, `JSONParser`, and the `CN1JSON` cn1lib.

5.2. Starting at the end… XML to Entity

Before wading through the details XML, JSON, expression languages, and CSS selectors, let's take a peek at our end goal, which is to convert XML or JSON data into Entities.

Consider the following sample XML file:

Some sample XML data that we will be parsing into entities using `ResultParser`.

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

```
<catalog>
   <book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
      <publish_date>2000-10-01</publish_date>
      <description>An in-depth look at creating applications
      with XML.</description>
   </book>
   <book id="bk102">
      <author>Ralls, Kim</author>
      <title>Midnight Rain</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2000-12-16</publish_date>
      <description>A former architect battles corporate zombies,
      an evil sorceress, and her own childhood to become queen
      of the world.</description>
   </book>
   <book id="bk103">
      <author>Corets, Eva</author>
      <title>Maeve Ascendant</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2000-11-17</publish_date>
      <description>After the collapse of a nanotechnology
      society in England, the young survivors lay the
      foundation for a new society.</description>
   </book>
   <book id="bk104">
      <author>Corets, Eva</author>
      <title>Oberon's Legacy</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2001-03-10</publish_date>
      <description>In post-apocalypse England, the mysterious
      agent known only as Oberon helps to create a new life
      for the inhabitants of London. Sequel to Maeve
      Ascendant.</description>
   </book>
   <book id="bk105">
      <author>Corets, Eva</author>
      <title>The Sundered Grail</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2001-09-10</publish_date>
      <description>The two daughters of Maeve, half-sisters,
      battle one another for control of England. Sequel to
      Oberon's Legacy.</description>
   </book>
```

```
    <book id="bk106">
       <author>Randall, Cynthia</author>
       <title>Lover Birds</title>
       <genre>Romance</genre>
       <price>4.95</price>
       <publish_date>2000-09-02</publish_date>
       <description>When Carla meets Paul at an ornithology
       conference, tempers fly as feathers get ruffled.</description>
    </book>
    <book id="bk107">
       <author>Thurman, Paula</author>
       <title>Splish Splash</title>
       <genre>Romance</genre>
       <price>4.95</price>
       <publish_date>2000-11-02</publish_date>
       <description>A deep sea diver finds true love twenty
       thousand leagues beneath the sea.</description>
    </book>
    <book id="bk108">
       <author>Knorr, Stefan</author>
       <title>Creepy Crawlies</title>
       <genre>Horror</genre>
       <price>4.95</price>
       <publish_date>2000-12-06</publish_date>
       <description>An anthology of horror stories about roaches,
       centipedes, scorpions  and other insects.</description>
    </book>
    <book id="bk109">
       <author>Kress, Peter</author>
       <title>Paradox Lost</title>
       <genre>Science Fiction</genre>
       <price>6.95</price>
       <publish_date>2000-11-02</publish_date>
       <description>After an inadvertant trip through a Heisenberg
       Uncertainty Device, James Salway discovers the problems
       of being quantum.</description>
    </book>
    <book id="bk110">
       <author>O'Brien, Tim</author>
       <title>Microsoft .NET: The Programming Bible</title>
       <genre>Computer</genre>
       <price>36.95</price>
       <publish_date>2000-12-09</publish_date>
       <description>Microsoft's .NET initiative is explored in
       detail in this deep programmer's reference.</description>
    </book>
    <book id="bk111">
       <author>O'Brien, Tim</author>
       <title>MSXML3: A Comprehensive Guide</title>
       <genre>Computer</genre>
       <price>36.95</price>
```

```
      <publish_date>2000-12-01</publish_date>
      <description>The Microsoft MSXML3 parser is covered in
      detail, with attention to XML DOM interfaces, XSLT processing,
      SAX and more.</description>
   </book>
   <book id="bk112">
      <author>Galos, Mike</author>
      <title>Visual Studio 7: A Comprehensive Guide</title>
      <genre>Computer</genre>
      <price>49.95</price>
      <publish_date>2001-04-16</publish_date>
      <description>Microsoft Visual Studio 7 is explored in depth,
      looking at how Visual Basic, Visual C++, C#, and ASP+ are
      integrated into a comprehensive development
      environment.</description>
   </book>
</catalog>
```

And suppose our application includes the following entities:

Minimal source code for a `Book` *entity.*

```
public class Book extends Entity {
    public static final EntityType TYPE = new EntityTypeBuilder()
        .string(Thing.identifier)
        .string(Thing.name)
        .string(Thing.description)
        .build();
    {
        setEntityType(TYPE);
    }
}
```

Minimal source code for a `Books` *entity. I find it helps for API clarity to create subclasses entity-specific collection types rather than just using generic EntityLists. This makes it easier for things like the* `ResultParser` *to introspect the data model and produce better results.*

```
public class Books extends EntityList<Book> {}
```

Minimal source code for a `Catalog` entity.

```
public class Catalog extends Entity {
    public static final Tag BOOKS = new Tag("Books");
    public static final EntityType TYPE = new EntityTypeBuilder()
        .list(Books.class, BOOKS)
        .build();
    {
        setEntityType(TYPE);
    }

}
```

We can convert this XML document into our entities using:

Using `ResultParser` to parse an XML document into Entities and EntityTypes.

```
ResultParser parser = new ResultParser(Catalog.TYPE) ①
    .property("./book", Catalog.BOOKS) ②
    .entity(Book.TYPE) ③
    .property("@id", Thing.identifier) ④
    .property("title", Thing.name) ⑤
    .property("description", Thing.description); ⑥

Catalog catalog = (Catalog)parser.parseXML(xmlContent); ⑦
for (Book book : (Books)catalog.get(Catalog.BOOKS)) {
    System.out.println("Name: " + book.get(Thing.name));
}
```

① Constructor takes the `Catalog.TYPE` entity type, which is the assigned entity type for the `Catalog` class, thus ensuring that this result parser will map the "root" tag of an XML document to a `Catalog` entity.

② "./book" is an expression language selector matching `<book>` elements that are direct children of the "current" element. `Catalog.BOOKS` indicates that the `<book>` elements should be mapped to the `Catalog.BOOKS` property of the `Catalog` entity.

③ `entity(Book.TYPE)`, create a new `ResultParser` for mapping the `Book` entity. This `entity()` method creates the "sub" parser, registers it with the "root" Catalog parser, and returns itself so that subsequent chained method calls are actually performed on the "Book" parser.

④ Map the "id" attribute of the `<book>` tag to the `Thing.identifier` property of the `Book` entity.

⑤ Map the contents of the `<title>` child tag to the `Thing.name` property of the `Book` entity.

⑥ Map the contents of the `<description>` child tag to the `Thing.description` property of the `Book` entity.

⑦ `parser.parseXML(xmlContent)` parses the provided XML content as a `Catalog` object.

This short example demonstrates how easy it is to parse arbitrary XML into Java entities without dictating any structural requirements on the XML data. The `ResultParser` uses the Result expression language to specify how the XML data should be mapped to entities. This example, being chosen for

clarity and small code-size primarily maps to entities that have the same structure as the XML data, but API is flexible enough to map different structures together. It also includes advanced facilities for custom content parsing and formatting. For example, you can provide a `DateFormatter` object to help format dates and time data.

5.3. JSON to Entity

Lest you think that the ResultParser is geared to XML data input exclusively, here is a motivating example the demonstrates the parsing of JSON data into entities.

```
{
  "colors": [
    {
      "color": "black",
      "category": "hue",
      "type": "primary",
      "code": {
        "rgba": [255,255,255,1],
        "hex": "#000"
      }
    },
    {
      "color": "white",
      "category": "value",
      "code": {
        "rgba": [0,0,0,1],
        "hex": "#FFF"
      }
    },
    {
      "color": "red",
      "category": "hue",
      "type": "primary",
      "code": {
        "rgba": [255,0,0,1],
        "hex": "#FF0"
      }
    },
    {
      "color": "blue",
      "category": "hue",
      "type": "primary",
      "code": {
        "rgba": [0,0,255,1],
        "hex": "#00F"
      }
    },
    {
      "color": "yellow",
      "category": "hue",
```

```
      "type": "primary",
      "code": {
        "rgba": [255,255,0,1],
        "hex": "#FF0"
      }
    },
    {
      "color": "green",
      "category": "hue",
      "type": "secondary",
      "code": {
        "rgba": [0,255,0,1],
        "hex": "#0F0"
      }
    },
  ]
}
```

The `Color` entity which will encapsulate a "row" of data in the JSON.

```
/**
 * Class to encapsulate a color.
 */
class Color extends Entity {

    /**
     * Define some tags which we'll use for properties
     * in our class.
     */
    public static final Tag type = new Tag("type"),
        red = new Tag("red"),
        green = new Tag("green"),
        blue = new Tag("blue"),
        alpha = new Tag("alpha");

    /**
     * Define the entity type.
     */
    public static final EntityType TYPE = entityTypeBuilder(Color.class)
        .string(Thing.name)
        .string(Product.category)
        .string(type)
        .Integer(red)
        .Integer(green)
        .Integer(blue)
        .Integer(alpha)
        .build();
}
```

The `Colors` entity, which encapsulates a list of colors.

```
/**
 * Encapsulates a list of colors.
 */
class Colors extends EntityList<Color>{
    static {
        // Register this EntityType
        EntityType.registerList(
            Colors.class, // Class used for list
            Color.class  // Class of row type
        );
    }
}
```

The `ColorSet` entity which will encapsulate the "root" of the data. This is largely created to make it easier to map the JSON data onto our entities.

```
/**
 * An entity type to model the root of the data set.
 */
class ColorSet extends Entity {
    public static final Tag colors = new Tag("colors");
    public static final EntityType TYPE = entityTypeBuilder(ColorSet.class)
        .list(Colors.class, colors)
        .factory(cls -> {return new ColorSet();})
        .build();

}
```

With all of the entity definitions out of the way, let's finally parse the JSON into entities:

```
ResultParser parser = resultParser(ColorSet.class)
        .property("colors", colors)
        .entityType(Color.class)
        .string("color", Thing.name)
        .string("category", Product.category)
        .string("type", type)
        .Integer("code/rgba[0]", red)
        .Integer("code/rgba[1]", green)
        .Integer("code/rgba[2]", blue)
        .Integer("code/rgba[3]", alpha)
        ;

ColorSet colorSet = (ColorSet)parser.parseJSON(jsonData, new ColorSet());
```

This looks very similar to the XML parsing example, but there are some notable differences. One key difference is how "attributes" are addressed in JSON vs XML. For XML, attributes are prefixed

with "@" in the expression language, whereas they are not for JSON. E.g. For the tag `<person name="Ted"/>` we would reference the name using "@name", whereas for the JSON object `{"name" : "Ted"}`, we would reference the name using "name".

This example is intended to provide a glimpse of how you would use CodeRAD's XML and JSON parsing facilities in a real app. I'll go over the details such as the expression language and APIs in subsequent sections.

5.4. A Bird's-eye View

Codename One's data parsing APIs can be grouped in the following three categories, listed in increasing order by level of abstraction:

1. **Low-level parsers** - E.g. `XMLParser` and `JSONParser` which parse String data into generic data structures like `Element`, `Map`, `List`, etc..
2. **Query and Selection** - E.g. `Result` and `ElementSelector` which provide a way to access data in an XML or JSON data set using simple expression languages resembling XPath and CSS selectors.
3. **Entity Mapping** - E.g. `ResultParser` which provides a way to convert JSON and XML data directly into Entity objects.

For low-level parsing, CodeRAD uses the core `XMLParser` and `JSONParser` classes.

For query and selection, CodeRAD provides its own APIs. The `Result` class and its XPath-like expression language are used internally by the `ResultParser` class for mapping JSON/XML into entities, but you can also use it directly query XML and JSON data directly.

The `ElementSelector` class provides an alternate syntax, resembling CSS selectors, for querying data in an XML data set. It is currently XML-only, and it leverages XML-specific characteristics to provide a fluid experience. This API is modeled after the `ComponentSelector` class, which, itself, is inspired by the popular **jQuery** javascript library.

5.5. Parsing XML and JSON

Low-level XML and JSON parsing can be performed using the core XMLParser and JSONParser classes respectively. For information on using these classes, refer to their javdoc pages.

5.6. Querying XML and JSON Data Using `Result`

The Result class provides a powerful expression language for accessing content from an XML document or JSON dataset. You can create a `Result` object to wrap a data set using the `Result.fromContent(…)` methods which accepts content either as XML/JSON strings, streams, or readers. You will also accept pre-parsed data in the form of an `Element` object (for XML data), or `Map` object (for JSON data).

The following is a simple usage example:

Sample code pulled from the CodeRAD unit tests demonstrating the use of Result to query XML data.

```
String xml = "<?xml version='1.0'?>\n"
                + "<person name=\"Paul\" email=\"paul@example.com\" dob=\"December 27,
1978\">"
                + "  <children>\n"
                + "    <person name=\"Jim\" email=\"jim@example.com\" dob=\"January
10, 1979\"/>\n"
                + "    <person name=\"Jill\" email=\"jill@example.com\" dob=\"January
11, 1979\"/>\n"
                + "  </children>\n"
                + "</person>";

Result r = Result.fromContent(xml, Result.XML);

r.get("/person/@name"); // "Paul"
r.getAsString("./@name"); // "Paul"
r.getAsString("@name"); // "Paul"
r.get("/person[0]/@name"); // "Paul"
r.get("./children/person[0]/@name"); // "Jim"
r.getAsString("./children/person/@name"); // "Jim"
r.getAsString("./children[0]/person/@name"); // "Jim"
r.getAsStringArray("./children/person/@name").length; // 2
r.get("/person/children/person[0]/@name"); // "Jim"
r.getAsString("/person[0]/children/person/@name"); "Jim"
r.getAsString("children[0]/person/@name"); // Jim
r.getAsStringArray("children/person/@name").length; // 2
```

In the above example, we parse an XML string directly using `Result`. The various `get(…)`, `getAsString(…)`, and `getAsStringArray(…)` method give you a flavour for the expression language. This example retrieved all data in `String` format, but the API includes methods for retrieving data for all of the types supported by JSON. Specifically:

1. `getAsBoolean(expr)`
2. `getAsBooleanArray(expr)`
3. `getAsDouble(expr)`
4. `getAsDoubleArray(expr)`
5. `getAsInteger(expr)`
6. `getAsIntegerArray(expr)`
7. `getAsLong(expr)`
8. `getAsLongArray(expr)`

Working with JSON is very similar, but there are some differences, which are related to the inherent differences between JSON data and XML data.

Let's look at an equivalent example, this time using JSON as the source format:

```
String json = "{\"name\":\"Paul\", \"email\":\"paul@example.com\", \"dob\":\"December
27, 1978\" "
        + ", \"children\": [{\"name\":\"Jim\", \"email\":\"jim@example.com\", \"dob\"
:\"January 10, 1979\"},"
        + "{\"name\"=\"Jill\", \"email\"=\"jill@example.com\", \"dob\":\"January 11,
1979\"}]}";


Result r = Result.fromContent(json, Result.JSON);

r.get("name"); // "Paul"
r.getAsString("name"); // "Paul"
r.get("name"); // "Paul"
r.get("./children[0]/name"); // "Jim"
r.get("children[0]/name"); // "Jim"
r.get("./children/person[0]/name"); // null
r.getAsString("./children/person/name"); // null
r.getAsString("./children[0]/name"); // "Jim"
r.getAsStringArray("./children/name").length; // 2
r.getAsStringArray("./children/name"); // String[]{"Jim", "Jill"}
r.get("./children/name"); // "Jim"
r.getAsString("children/person/name");  // null
r.getAsString("children[0]/person/name"); // null
r.getAsStringArray("children/person/name").length; // 0
```

Even though this JSON data is roughly equivalent to the XML data in the above example, we see that some of the expressions we use will differ. In XML the XML tag is generally used as part of the expression language query. In JSON there are no "tags", only Map properties and array indices. Also, XML uses the "@" prefix for addressing attributes to distinguish them from child tags.

E.g. Given the following XML tag:

```
<root name="John">
    <name>Jack</name>
</root>
```

We would have the following:

```
Result r = Result.fromContent(xmlString, Result.XML);
r.get("name"); // "Jack"
r.get("@name"); // "John"
```

5.6.1. The Expression Language

The expression language used by `Result` is very similar to XPath. E.g.

1. `/` is a path separator.

2. `//` is a "glob" path separator.
3. `@attributeName` Refers to an attribute named "attributeName".
4. `[..]` Is used for array indices, and attribute filters.
5. `.` Refers to the current element
6. `..` Refers to the parent element

See the Result javadocs for more details documentation on the expression language an API.

5.7. `ResultParser` - From XML/JSON to Entities

The ResultParser class uses the Result expression language to map XML and JSON data into Entities.

See the ResultParser javadocs for usage examples.

5.8. Asynchronous Parsing

In Codename One, most things are done on the EDT. The main app lifecycle methods (`start()`, `stop()`, etc..) are run on the EDT, as are all event handlers. Any code that mutates the UI hierarchy **must** be run on the EDT, so we often just stay on the EDT to make things simpler. No thread race conditions, etc.. Some things, however, would be better done in a background thread so that they don't interfer with the app's UI performance. Data parsing, especially when parsing large amounts of data, is one of those tasks that can potentially get in the way of a smooth user experience. This is for two reasons:

1. Parsing large amounts of data is processor intensive and can get in the way of the app's ability to **"draw" itself at 60fps.**
2. Parsing large amounts of data may involve a the creation of lots of short-lived objects (usually strings), which **cause the garbage collector to churn**. On iOS, in particular, and likely on other platforms, to varying extents, the garbage collector will only "stop the world", in extreme circumstances, where memory is at a critical level, but it **may** need to stop individual threads temporarily that are producing large amounts of "garbage" while it catches up with the backlog. It is better for everyone involved if excess garbage related to data parsing happens on a thread **other than** the EDT, so that the garbage collector isn't faced with the sophie's choice of whether to "lock" the EDT, or risk running out of memory.

The ParsingService class makes it easier to parse data off the EDT.

A simple example:

Adapting the example from JSON to Entity to perform asynchronouze parsing off the EDT

```
ParsingService parserService = new ParsingService();

parserService.parseJSON(jsonData, parser, new ColorSet()).ready(colorSet -> {
    // Callback passed the resulting colorSet object
    // once parsing is complete
});
```

IMPORTANT	Each `ParsingService` instance creates and wraps a Thread which is used to perform the parsing. When you are finished using a particular `ParsingService` instance, you should call its `stop()` method to stop the thread.

Refer to the ParsingService javadocs for more examples and details about the API.