Table of Contents

TornadoFX Guide

This is a continual effort to fully document the TornadoFX framework in the format of a book.

Part 1: TornadoFX Fundamentals

This section will cover everything you will need to get started with TornadoFX. These sections are somewhat designed to be read sequentially, as concepts may build on top of each other.

Introduction

User interfaces are becoming increasingly critical to the success of consumer and business applications. With the rise of consumer mobile apps and web applications, business users are increasingly holding enterprise applications to a higher standard of quality. They want rich, feature-packed user interfaces that provide immediate insight and navigate complex screens intuitively. More importantly, they want the application to adapt quickly to business changes on a frequent basis. For the developer, this means the application must not only be maintainable but also evolvable. TornadoFX seeks to assist all these objectives and greatly streamline the development of JavaFX UI's.

While much of the enterprise IT world is pushing HTML5 and cloud-based applications, many businesses are still using desktop UI frameworks like JavaFX. While it does not distribute to large audiences as easily as web applications, JavaFX works well for "in-house" business applications. Its high-performance with large datasets (and the fact it is native Java) make it a practical choice for applications used behind the corporate firewall.

JavaFX, like many UI frameworks, can quickly become verbose and difficult to maintain. Fortunately, the rapidly growing Kotlin platform has allowed an opportunity to rethink how JavaFX applications are built.

Why TornadoFX?

In February 2016, JetBrains released Kotlin, a new JVM language that emphasizes pragmatism over convention. Kotlin works at a higher level of abstraction and provides practical language features not available in Java. One of the more important features of Kotlin is its 100% interoperability with existing Java libraries and codebases, including JavaFX. Even more important is in 2017, Google backed Kotlin as an official language for Android. This gives Kotlin a bright future that has already extended beyond mobile apps.

While JavaFX can be used with Kotlin in the same manner as Java, some believed Kotlin had language features that could streamline and simplify JavaFX development. Well before Kotlin's beta, Eugen Kiss prototyped JavaFX "builders" with KotlinFX. In January 2016, Edvin Syse rebooted the initiative and released TornadoFX.

TornadoFX seeks to greatly minimize the amount of code needed to build JavaFX applications. It not only includes type-safe builders to quickly lay out controls and user interfaces, but also features dependency injection, delegated properties, control extension functions, and other practical features enabled by Kotlin. TornadoFX is a fine showcase of

how Kotlin can simplify codebases, and it tackles the verbosity of UI code with elegance and simplicity. It can work in conjunction with other popular JavaFX libraries such as ControlsFX and JFXtras. It works especially well with reactive frameworks such as ReactFX as well as RxJava and friends (including RxJavaFX, RxKotlin, and RxKotlinFX).

Reader Requirements

This book expects readers to have some knowledge of Kotlin and have spent some time getting acquainted with it. There will be some coverage of Kotlin language features but only to a certain extent. If you have not done so already, read the JetBrains Kotlin Reference and spend a good few hours studying it.

It helps to be familiar with JavaFX but it is not a requirement. Many Kotlin developers reported using TornadoFX successfully without any JavaFX knowledge. What is particularly important to know about JavaFX is its concepts of `ObservableValue` and `Bindings`, which this guide will cover to a good degree.

A Motivational Example

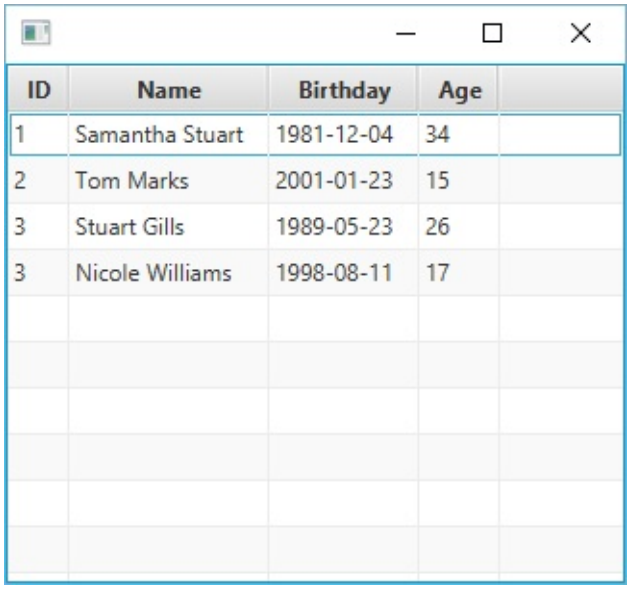
If you have worked with JavaFX before, you might have created a `TableView` at some point. Say you have a given domain type `Person`. TornadoFX allows you to much more concisely create the JavaBeans-like convention used for the JavaFX binding.

```
class Person(id: Int, name: String, birthday: LocalDate) {
    val idProperty = SimpleIntegerProperty(id)
    var id by idProperty

    val nameProperty = SimpleStringProperty(name)
    var name by nameProperty

    val birthdayProperty = SimpleObjectProperty(birthday)
    var birthday by birthdayProperty

    val age: Int get() = Period.between(birthday, LocalDate.now()).years
}
```

You can then build an entire " `View` " containing a `TableView` with a small code footprint.

```
class MyView : View() {

    private val persons = listOf(
            Person(1, "Samantha Stuart", LocalDate.of(1981,12,4)),
            Person(2, "Tom Marks", LocalDate.of(2001,1,23)),
            Person(3, "Stuart Gills", LocalDate.of(1989,5,23)),
            Person(3, "Nicole Williams", LocalDate.of(1998,8,11))
    ).observable()

    override val root = tableview(persons) {
        column("ID", Person::idProperty)
        column("Name", Person::nameProperty)
        column("Birthday", Person::birthdayProperty)
        column("Age", Person::age)
    }
}
```

RENDERED OUTPUT:



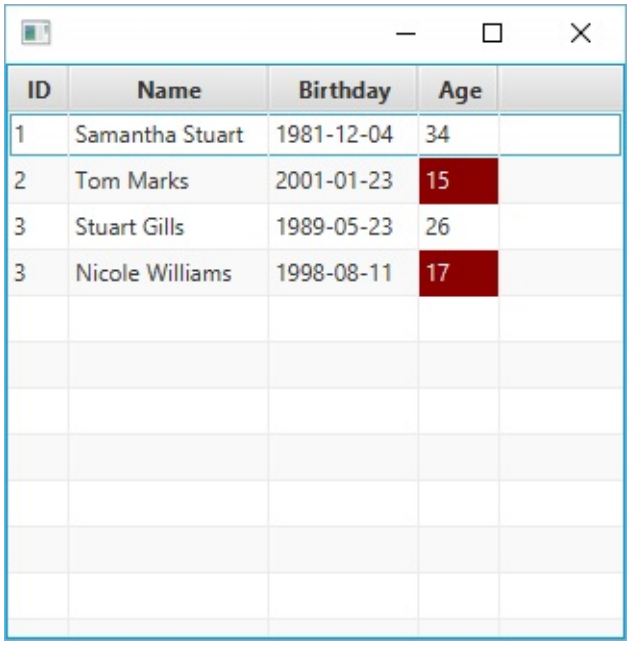
Half of that code was just initializing sample data! If you hone in on just the part declaring the `TableView` with four columns (shown below), you will see it took a simple functional construct to build a `TableView`. It will automatically support edits to the fields as well.

```
tableview(persons) {
    column("ID", Person::idProperty)
    column("Name", Person::nameProperty)
    column("Birthday", Person::birthdayProperty)
    column("Age", Person::age)
}
```

As shown below, we can use the `cellFormat()` extension function on a `TableColumn`, and create conditional formatting for "Age" values that are less than `18`.

```
tableview<Person> {
    items = persons
    column("ID", Person::idProperty)
    column("Name", Person::nameProperty)
    column("Birthday", Person::birthdayProperty)
    column("Age", Person::age).cellFormat {
        text = it.toString()
        style {
            if (it < 18) {
                backgroundColor += c("#8b0000")
                textFill = Color.WHITE
            }
        }
    }
}
```

RENDERED OUTPUT:



These declarations are pure Kotlin code, and TornadoFX is packed with expressive power for dozens of cases like this. This allows you to focus on creating solutions rather than engineering UI code. Your JavaFX applications will not only be turned around more quickly, but also be maintainable and evolvable.

Setting Up

To use TornadoFX, there are several options to set up the dependency for your project. Mainstream build automation tools like Gradle and Maven are supported and should have no issues in getting set up.

Please note that TornadoFX is a Kotlin library, and therefore your project needs to be configured to use Kotlin. For Gradle and Maven configurations, please refer to the Kotlin Gradle Setup and Kotlin Maven Setup guides. Make sure your development environment or IDE is equipped to work with Kotlin and has the proper plugins and compilers.

This guide will use Intellij IDEA to walk through certain examples. IDEA is the IDE of choice to work with Kotlin, although Eclipse has a plugin as well.

Gradle

For Gradle, you can set up the dependency directly from Maven Central. Provide the desired version number for the `x.y.z` placeholder.

```
repositories {
    mavenCentral()
}

// Minimum jvmTarget of 1.8 needed since Kotlin 1.1
compileKotlin {
    kotlinOptions.jvmTarget= 1.8
}

dependencies {
    compile 'no.tornado:tornadofx:x.y.z'
}
```

Maven

To import TornadoFX with Maven, add the following dependency to your POM file. Provide the desired version number for the `x.y.z` placeholder.

Goes into kotlin-maven-plugin block:

```
<configuration>
    <jvmTarget>1.8</jvmTarget>
</configuration>
```

Then this goes into `dependencies` block:

```
<dependency>
    <groupId>no.tornado</groupId>
    <artifactId>tornadofx</artifactId>
    <version>x.y.z</version>
</dependency>
```

Other Build Automation Solutions

For instructions on how to use TornadoFX with other build automation solutions, please refer to the [TornadoFX page at the Central Repository]
([http://search.maven.org/#search|gav|1|g%3A"no.tornado]
(http://search.maven.org/#search|gav|1|g%3A"no.tornado)" AND a%3A"tornadofx")

Manual Import

To manually download and import the JAR file, go to the TornadoFX release page or the Central Repository. Download the JAR file and configure it into your project.

Components

JavaFX uses a theatrical analogy to organize an `Application` with `Stage` and `Scene` components.

TornadoFX builds on this by providing `View`, `Controller`, and `Fragment` components. While the `Stage`, and `Scene` are used by TornadoFX, the `View`, `Controller`, and `Fragment` introduces new concepts that streamline development. Many of these components are automatically maintained as singletons, and can communicate to each other through TornadoFX's simple dependency injections or direct instantiation.

You also have the option to utilize FXML which will be discussed in Chapter 10. But first, lets extend `App` to create an entry point that launches a TornadoFX application.

App and View Basics

To create a TornadoFX application, you must have at least one class that extends `App`. An **App** is the entry point to the application and specifies the initial `View`. It does in fact extend JavaFX `Application`, but you do not necessarily need to specify a `start()` or `main()` method.

Extend `App` to create your own implementation and specify the primary view as the first constructor argument.

```
class MyApp: App(MyView::class)
```

A **View** contains display logic as well as a layout of Nodes, similar to the JavaFX `Stage`. It is automatically managed as a singleton. When you declare a `View` you must specify a `root` property which can be any `Parent` type, and that will hold the View's content.

In the same Kotlin file or in a new file, extend a class off of `View`. Override the abstract `root` property and assign it `VBox` or any other `Node` you choose.

```
class MyView: View() {
    override val root = VBox()
}
```

However, we might want to populate this `VBox` acting as the `root` control. Using the initializer block, let's add a JavaFX `Button` and a `Label`. You can use the "plus assign" `+=` operators to add children, such as a `Button` and `Label`.

```
class MyView: View() {
    override val root = VBox()

    init {
        root += Button("Press Me")
        root += Label("Waiting")
    }
}
```

While it is pretty clear what is going on just looking at this code, TornadoFX provides a builder syntax that will streamline your UI code further. We will gradually move into builder syntax in the next chapter.

Next we will see how to run this application.

Starting a TornadoFX Application

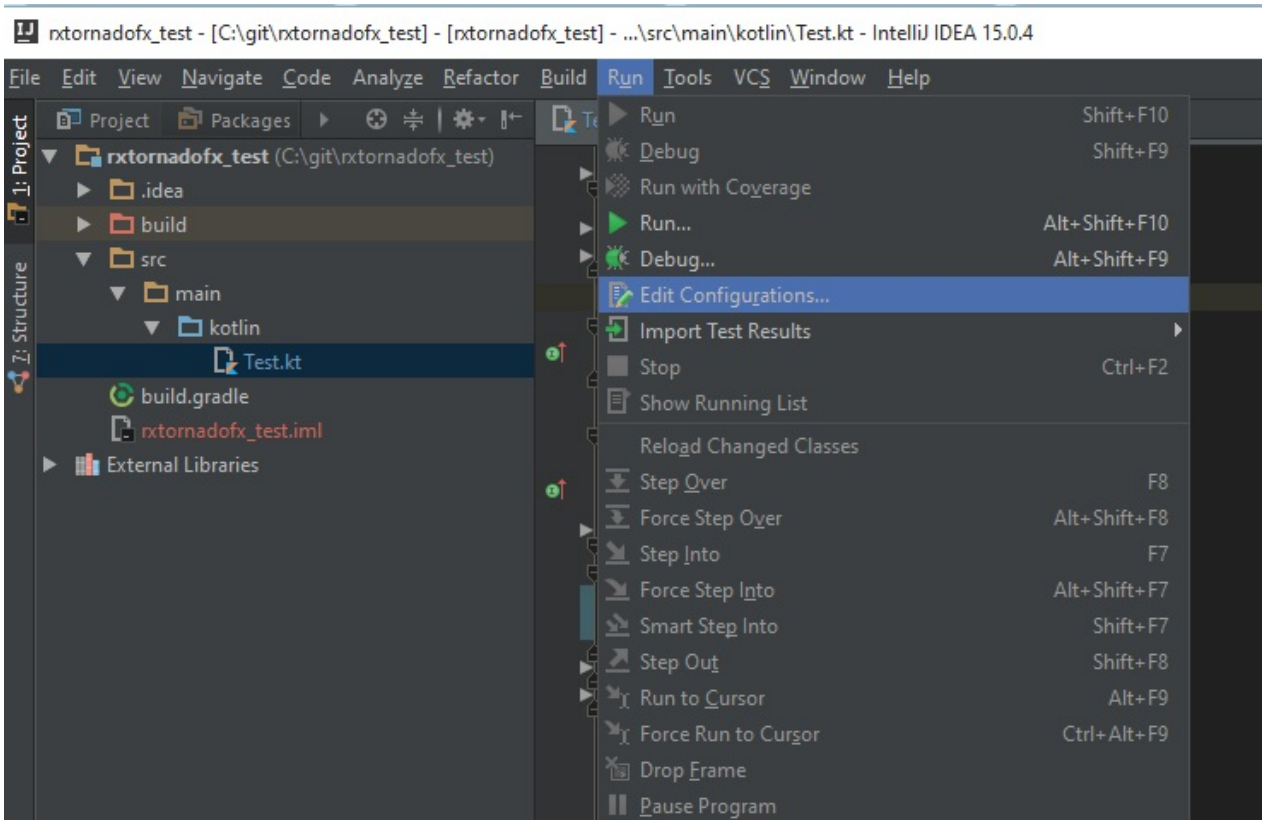
Newer versions of the JVM know how to start JavaFX applications without a `main()` method. A JavaFX application, and by extension a TornadoFX application, is any class that extends `javafx.application.Application`. Since `tornadofx.App` extends `javafx.application.Application`, TornadoFX apps are no different. Therefore you would start the app by referencing `com.example.app.MyApp`, and you do not necessarily need a `main()` function (unless you need to supply command line arguments). In that case you would add a package level main function to the `MyApp.kt` file:

```
fun main(args: Array<String>) {
  Application.launch(MyApp::class.java, *args)
}
```

This main function would be compiled to `com.example.app.MyAppKt`. Notice the `Kt` at the end. When you create a packagelevel main function, it will always have a class name of the fully qualified package, plus the file name, appended with `Kt`.

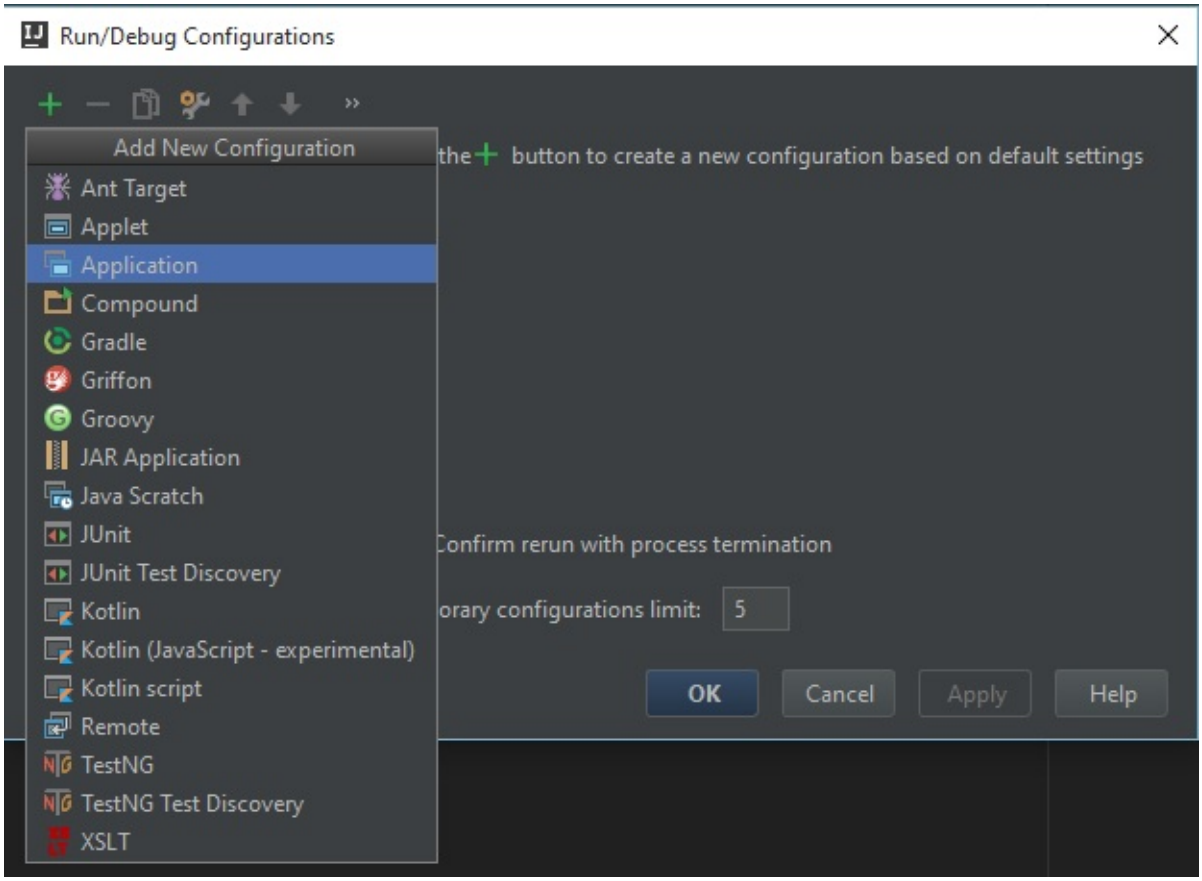
For launching and testing the `App`, we will use Intellij IDEA. Navigate to *Run→Edit Configurations* (Figure 3.1).

Figure 3.1



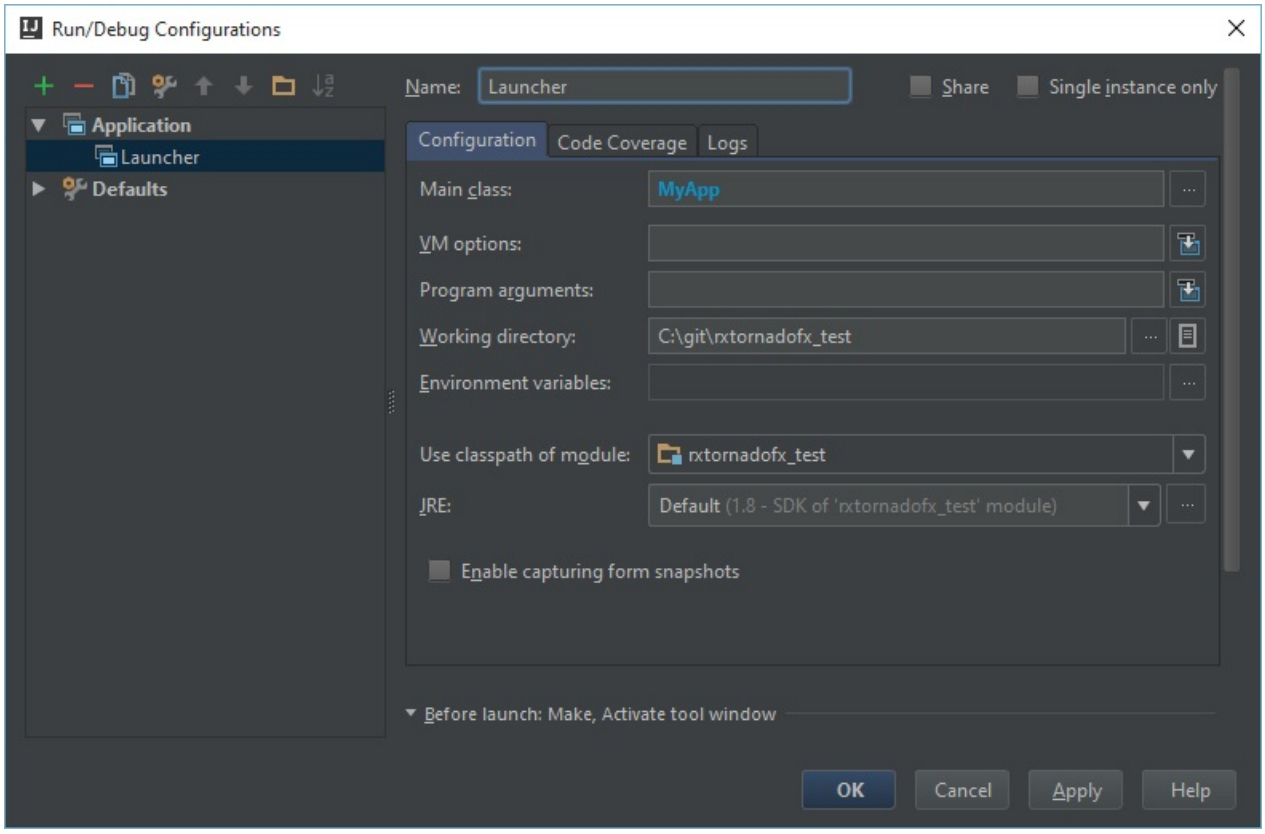
Click the green "+" sign and create a new Application configuration (Figure 3.2).

Figure 3.2



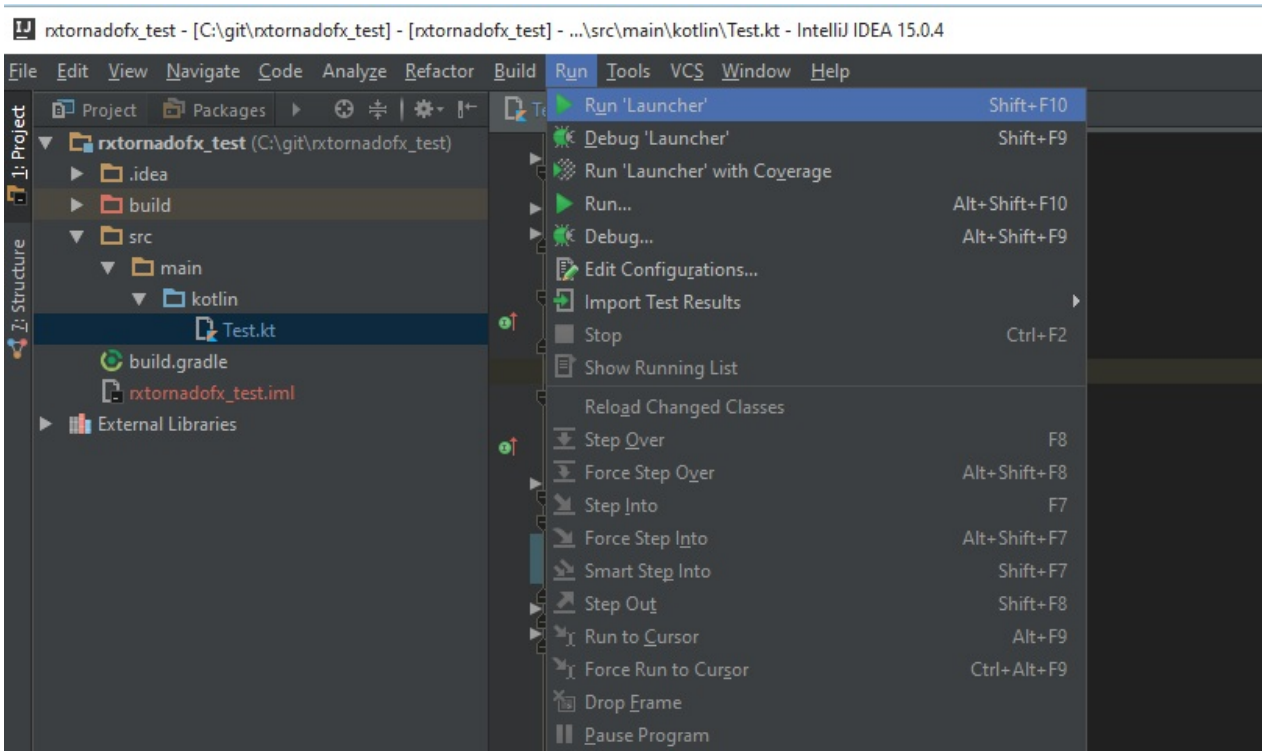
Specify the name of your "Main class" which should be your `App` class. You will also need to specify the module it resides in. Give the configuration a meaningful name such as "Launcher". After that click "OK" (Figure 3.3).

Figure 3.3



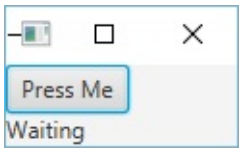
You can run your TornadoFX application by selecting *Run→Run 'Launcher'* or whatever you named the configuration (Figure 3.4).

Figure 3.4



You should now see your application launch (Figure 3.5)

Figure 3.5



Congratulations! You have written your first (albeit simple) TornadoFX application. It may not look like much right now, but as we cover more of TornadoFX's powerful features we will be creating large, impressive user interfaces with little code in no time. But first let's understand a little better what is happening between `App` and `View`.

Understanding Views

Let's dive a little deeper into how a `View` works and how it can be invoked. Take a look at the `App` and `View` classes we just built.

```
class MyApp: App(MyView::class)

class MyView: View() {
    override val root = VBox()

    init {
        root += Button("Press Me")
        root += Label("Waiting")
    }
}
```

A `View` contains a hierarchy of JavaFX Nodes and is injected by name wherever it is called. In the next section we will learn how to leverage powerful builders to create these `Node` hierarchies quickly. There is only one instance of `MyView` maintained by TornadoFX, effectively making it a singleton. TornadoFX also supports scopes, which can group together a collection of `View` s, `Fragment` s and `Controller` s in separate instances, resulting in a `View` only being a singleton inside that scope. This is great for Multiple-Document Interface applications and other advanced use cases.

Using inject() and Embedding Views

You can also inject one or more Views into another `View` . Below we embed a `TopView` and `BottomView` into a `MasterView` . Note we use the `inject()` delegate property to lazily inject the `TopView` and `BottomView` instances. Then we call each "child" View's `root` to assign them to the `BorderPane` (Figure 3.6).

```
import javafx.scene.control.Label
import javafx.scene.layout.BorderPane
import tornadofx.*


class MasterView: View() {
    val topView: TopView by inject()
    val bottomView: BottomView by inject()

    override val root = BorderPane()

    init {
        with(root) {
            top = topView.root
            bottom = bottomView.root
        }
    }
}

class TopView: View() {
    override val root = Label("Top View")
}

class BottomView: View() {
    override val root = Label("Bottom View")
}
```

Injection Using find()

The `inject()` delegate will lazily assign a given component to a property. The first time that component is called is when it will be retrieved. Alternatively, instead of using the `inject()` delegate you can use the `find()` function to retrieve a singleton instance of a `View` or other components.

```
import javafx.scene.control.Label
import javafx.scene.layout.BorderPane
import tornadofx.*


class MasterView : View() {
    override val root = BorderPane()

    val topView = find(TopView::class)
    val bottomView = find(BottomView::class)

    init {
        with(root) {
            top = topView.root
            bottom = bottomView.root
        }
    }
}

class TopView: View() {
    override val root = Label("Top View")
}

class BottomView: View() {
    override val root = Label("Bottom View")
}
```

You can use either `find()` or `inject()`, but using `inject()` delegates is the most idiomatic means to perform dependency injection.

A Brief Intro to Builders

While we will cover builders more in depth in the next chapter, it is time to reveal that the above example can be written in a much more concise and expressive syntax:

```
import javafx.scene.control.Label
import tornadofx.*


class MasterView : View() {
    override val root = borderpane {
        top(TopView::class)
        bottom(BottomView::class)
    }
}

class TopView: View() {
    override val root = Label("Top View")
}

class BottomView: View() {
    override val root = Label("Bottom View")
}
```

Instead of injecting the `TopView` and `BottomView` and then assigning their respective root nodes to the `BorderPane` s `top` and `bottom` property, we specify the `BorderPane` with the builder syntax (all lower case) and then declaratively tell TornadoFX to pull in the two subviews and assign them to the `top` and `bottom` properties. Hopefully you find this is much more expressive, with a lot less boiler plate. This is one of the most important principles TornadoFX tries to live by: reduce boiler plate and increase readability. The end result is often less code and less bugs.

Controllers

In many cases, it is considered a good practice to separate a UI into three distinct parts:

1. **Model** - The business code layer that holds core logic and data
2. **View** - The visual display with various input and output controls
3. **Controller** - The "middleman" mediating events between the Model and the View

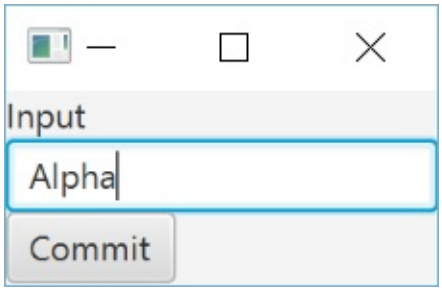
> There are other flavors of MVC like MVVM and MVP, all of which can be leveraged in TornadoFX.

While you could put all logic from the Model and Controller right into the view, it is often cleaner to separate these three pieces distinctly to maximize reusability. One commonly used pattern to accomplish this is the MVC pattern. In TornadoFX, a `Controller` can be injected to support a `View` .

Here is a simple example. Create a simple `View` with a `TextField` whose value is written to a "database" when a `Button` is clicked. We can inject a `Controller` that handles interacting with the model that writes to the database. Since this example is simplified, there will be no database but a printed message will serve as a placeholder (Figure 3.7).

```
import tornadofx.*


class MyView : View() {

    val controller: MyController by inject()

    override val root = vbox {

        label("Input")

        val inputField = textfield()

        button("Commit") {
            action {
                controller.writeToDb(inputField.text)
                inputField.clear()
            }
        }
    }
}


class MyController: Controller() {
    fun writeToDb(inputValue: String) {
        println("Writing $inputValue to database!")
    }
}
```

Figure 3.7



When we build the UI, we make sure to add a reference to the `inputField` so that it can be referenced from the `onClick` event handler of the "Commit" button later, hence why we save it to a variable. When the "Commit" button is clicked, you will see the Controller prints a line to the console.

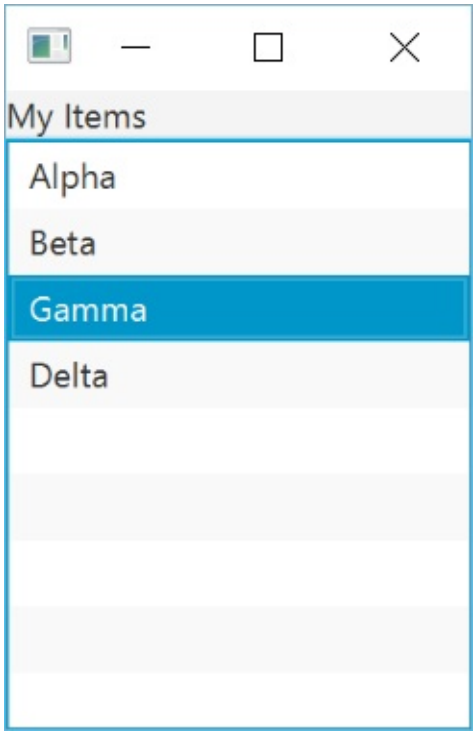
```
Writing Alpha to database!
```

It is important to note that while the above code works, and may even look pretty good, it is a good practice to avoid referencing other UI elements directly. Your code will be much easier to refactor if you bind your UI elements to properties and manipulate the properties instead. We will introduce the `ViewModel` later, which provides even easier ways to deal with this type of interaction.

You can also use Controllers to provide data to a `View` (Figure 3.8).

```
import javafx.collections.FXCollections
import tornadofx.*


class MyView : View() {
    val controller: MyController by inject()

    override val root = vbox {
        label("My items")
        listview(controller.values)
    }
}

class MyController: Controller() {
    val values = FXCollections.observableArrayList("Alpha","Beta","Gamma","Delta")
}
```

Figure 3.8



The `VBox` contains a `Label` and a `ListView`, and the `items` property of the `ListView` is assigned to the `values` property of our `Controller`.

Whether they are reading or writing data, Controllers can have long-running tasks and should not perform work on the JavaFX thread. We will learn how to easily offload work to a worker thread using the `runAsync` construct next.

Long Running Tasks

Whenever you call a function in a controller, you need to determine if that function returns immediately or if it performs potentially long-running tasks. If you call a function on the JavaFX Application Thread, the UI will be unresponsive until the call completes. Unresponsive UI's is a killer for user acceptance, so we need to make sure to run expensive operations in the background. TornadoFX provides the `runAsync` function to help with this.

Code placed inside a `runAsync` block will run in the background. If the result of the background call should update your UI, you must make sure that you apply the changes on the JavaFX Application Thread. The `ui` block, which typically follows, does exactly that.

```
val textfield = textfield()
button("Update text") {
    action {
        runAsync {
            myController.loadText()
        } ui { loadedText ->
            textfield.text = loadedText
        }
    }
}
```

When the button is clicked, the action inside the `action` builder is run. It makes a call out to `myController.loadText()` and applies the result to the text property of the textfield when it returns. The UI stays responsive while the controller function runs.

Under the covers, `runAsync` creates a JavaFX `Task` objects, and spins off a separate thread to run your call inside the `Task`. You can even assign this `Task` to a variable and bind it to a UI to show progress while your operation is running.

In fact, this is so common that there is also an default ViewModel called `TaskStatus` which contains observable values for `running`, `message`, `title`, and `progress`. You can supply the `runAsync` call with a specific instance of the `TaskStatus` object, or use the default. There is also a version of `runAsync` called `runAsyncWithProgress` which will cover the current `Node` with a progress indicator while the long running operation runs.

The TornadoFX sources includes an example usage of this in the `AsyncProgressApp.kt` file.

> If you need to handle a great deal of complex concurrency, you may consider using RxKotlin with RxKotlinFX. Rx also has the ability to handle rapid user inputs and events, and kill previous requests to only chase after the latest.

Fragment

Any `View` you create is a singleton, which means you typically use it in only one place at a time. The reason for this is that the root node of the `View` can only have a single parent in a JavaFX application. If you assign it another parent, it will disappear from its previous parent.

However, if you would like to create a piece of UI that is short-lived or can be used in multiple places, consider using a `Fragment`. A **Fragment** is a special type of `View` designed to have multiple instances. They are particularly useful for popups or as pieces of a larger UI (such as ListCells, which we look at via the `ListCellFragment` later).

Both `View` and `Fragment` support `openModal()`, `openWindow()`, and `openInternalWindow()` functions that will open the root node in a separate Window.

```
import javafx.stage.StageStyle
import tornadofx.*


class MyView : View() {
    override val root = vbox {
        button("Press Me") {
            action {
                find(MyFragment::class).openModal(stageStyle = StageStyle.UTILITY)
            }
        }
    }
}

class MyFragment: Fragment() {
    override val root = label("This is a popup")
}
```

You can also pass optional arguments to `openModal()` to modify a few of its behaviors

Optional Arguments for openModal()

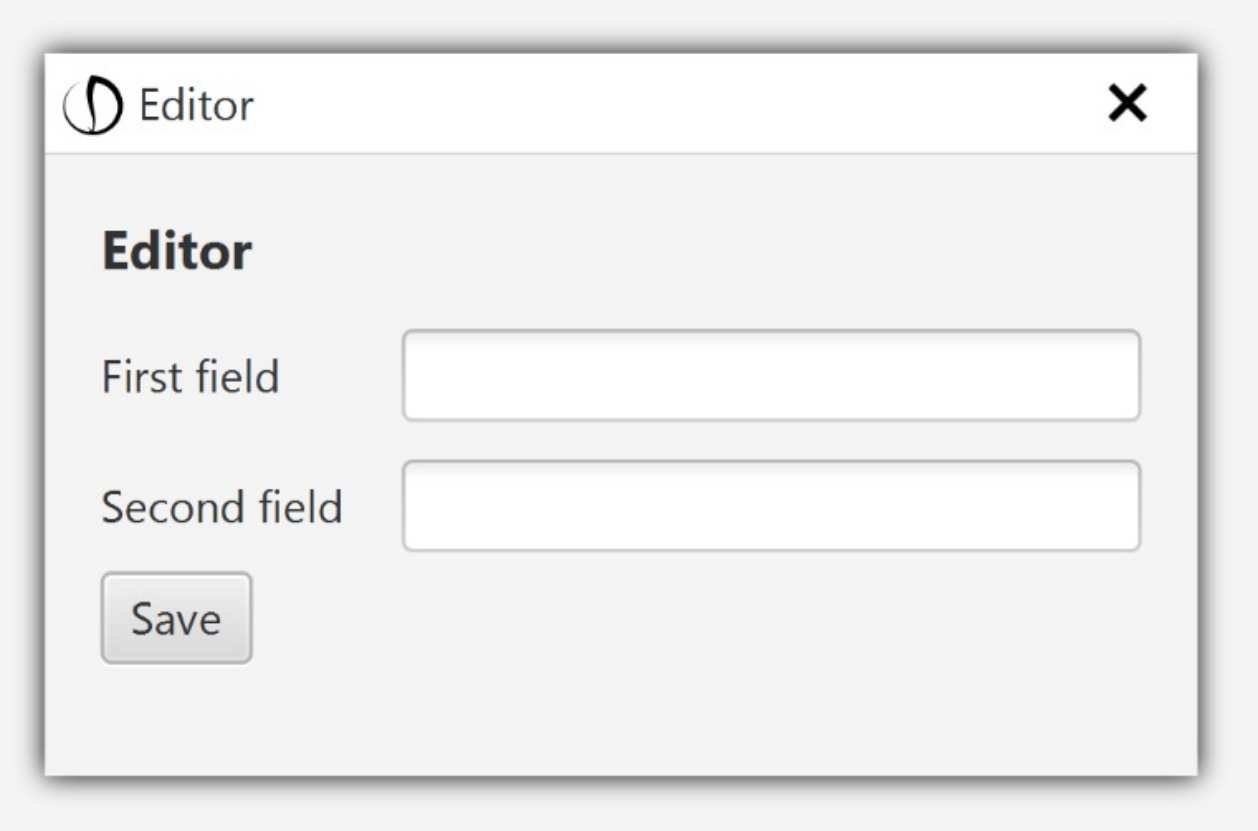
Argument	Type	Description
stageStyle	StageStyle	Defines one of the possible enum styles for `Stage`. Default: `StageStyle.DECORATED`
modality	Modality	Defines one of the possible enum modality types for `Stage`. Default: `Modality.APPLICATION_MODAL`
escapeClosesWindow	Boolean	Sets the `ESC` key to call `closeModal()`. Default: `true`
owner	Window	Specify the owner Window for this Stage`
block	Boolean	Block UI execution until the Window closes. Default: `false`

InternalWindow

While `openModal` opens in a new `Stage`, `openInternalWindow` opens over the current root node, or any other node if you specify it:

```
    button("Open editor") {
        action {
            openInternalWindow(Editor::class)
        }
    }
```

Figure 3.9



A good use case for the internal window is for single stage environments like JPro, or if you want to customize the window trim to make the window appear more in line with the design of your application. The Internal Window can be styled with CSS. Take a look at the `InternalWindow.Styles` class for more information about styleable properties.

The internal window API differs from modal/window in one important aspect. Since the window opens over an existing node, you typically call `openInternalWindow()` from within the `View` you want it to open on top of. You supply the View you want to show, and you can optionally supply what node to open over via the `owner` parameter.

Optional Arguments for openInternalWindow()

Argument	Type	Description
view	UIComponent	The component will be the content of the new window
view	KClass	Alternatively, you can supply the class of the view instead of an instance
icon	Node	Optional window icon
scope	Scope	If you specify the view class, you can also specify the scope used to fetch the view
modal	Boolean	Defines if the covering node should be disabled while the internal window is active. Default: `true`
escapeClosesWindow	Boolean	Sets the `ESC` key to call `close()`. Default: `true`
owner	Node	Specify the owner Node for this window. The window will by default cover the root node of this view.`

Closing Modal Windows

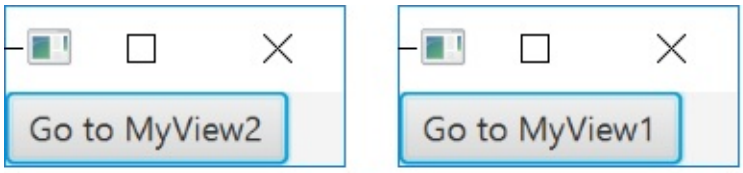
Any `Component` opened using `openModal()`, `openWindow()` or `openInternalWindow()` can be closed by calling `closeModal()`. It is also possible to get to the `InternalWindow` instance directly if needed using `findParentOfType(InternalWindow::class)`.

Replacing Views and Docking Events

With TornadoFX, is easy to swap your current `View` with another `View` using `replaceWith()`, and optionally add a transition. In the example below, a `Button` on each `View` will switch to the other view, which can be `MyView1` or `MyView2` (Figure 3.10).

```
import tornadofx.*


class MyView1: View() {
    override val root = vbox {
        button("Go to MyView2") {
            action {
                replaceWith(MyView2::class)
            }
        }
    }
}

class MyView2: View() {
    override val root = vbox {
        button("Go to MyView1") {
            action {
                replaceWith(MyView1::class)
            }
        }
    }
}
```

Figure 3.10



You also have the option to specify a spiffy animation for the transition between the two Views, as shown below.

```
replaceWith(MyView1::class, ViewTransition.Slide(0.3.seconds, Direction.LEFT)
```

This works by replacing the `root` `Node` on a given `View` with another `View`'s `root`. There are two functions you can override on `View` to leverage when a View's `root` `Node` is connected to a parent (`onDock()`), and when it is disconnected (`onUndock()`). You can leverage these two events to connect and "clean up" whenever a `View` comes in or falls out. You will notice running the code below that whenever a `View` is swapped, it will undock that previous `View` and dock the new one. You can leverage these two events to manage initialization and disposal tasks.

```
import tornadofx.*


class MyView1: View() {
    override val root = vbox {
        button("Go to MyView2") {
            action {
                replaceWith(MyView2::class)
            }
        }
    }

    override fun onDock() {
        println("Docking MyView1!")
    }

    override fun onUndock() {
        println("Undocking MyView1!")
    }
}

class MyView2: View() {
    override val root = vbox {
        button("Go to MyView1") {
            action {
                replaceWith(MyView1::class)
            }
        }
    }

    override fun onDock() {
        println("Docking MyView2!")
    }
    override fun onUndock() {
        println("Undocking MyView2!")
    }
}
```

Passing Parameters to Views

The best way to pass information between views is often an injected `ViewModel`. Even so, it can still be convenient to be able to pass parameters to other components. The `find` and `inject` functions supports varargs of `Pair<String, Any>` which can be used for just this purpose. Consider a customer list that opens a customer editor for the selected customer. The action to edit a customer might look like this:

```
fun editCustomer(customer: Customer) {
    find<CustomerEditor>(mapOf(CustomerEditor::customer to customer).openWindow())
}
```

The parameters are passed as a map, where the key is the property in the view and the value is whatever you want the property to be. This gives you a type safe way of configuring parameters for the target View.

Here we use the Kotlin `to` syntax to create the parameter. This could also have been written as `Pair(CustomerEditor::customer, customer)` if you prefer. The editor can now access the parameter like this:

```
class CustomerEditor : Fragment() {
    val customer: Customer by param()
}
```

If you want to inspect the parameters instead of blindly relying on them to be available, you can either declare them as nullable or consult the `params` map:

```
class CustomerEditor : Fragment() {
    init {
        val customer = params["customer"] as? Customer
        if (customer != null) {
            ...
        }
    }
}
```

If you do not care about type safety you can also pass parameters as `mapOf("customer" to customer)`, but then you miss out on automatic refactoring if you rename a property in the target view.

Accessing the Primary Stage

`View` has a property called `primaryStage` that allows you to manipulate properties of the `Stage` backing it, such as window size. Any `View` or `Fragment` that were opened via `openModal` will also have a `modalStage` property available.

Accessing the Scene

Sometimes it is necessary to get a hold of the current scene from within a `View` or `Fragment`. This can be achieved with `root.scene`, or if you are within a type safe builder, just call `scene`.

Accessing Resources

Lots of JavaFX APIs takes resources as a `URL` or the `toExternalForm` of an URL. To retrieve a resource url one would typically write something like:

```
val myAudioClip = AudioClip(MyView::class.java.getResource("mysound.wav").toExternalFo
rm())
```

However, every `Component` has a `resources` object which can retrieve the external form url of a resource like this:

```
val myAudiClip = AudioClip(resources["mysound.wav"])
```

If you need an actual `URL`, it can be retrieved like this:

```
val myResourceURL = resources.url("mysound.wav")
```

The `resources` helper also has several other helpful functions to help you turn files relative to the `Component` into an object of the type you need:

```
val myJsonObject = resources.json("myobject.json")
val myJsonArray = resources.jsonArray("myarray.json")
val myStream = resources.stream("somefile")
```

> It is worth mentioning that the `json` and `jsonArray` functions are also available on `InputStream` objects.

Resources are relative to the `Component` but you can also retrieve a resource by it's full path, starting with a `/`.

Summary

TornadoFX is filled with simple, streamlined, and powerful injection tools to manage Views and Controllers. It also streamlines dialogs and other small UI pieces using `Fragment`. While the applications we built so far are pretty simple, hopefully you appreciate the simplified

concepts TornadoFX introduces to JavaFX. In the next chapter we will cover what is arguably the most powerful feature of TornadoFX: Type-Safe Builders.

Basic Controls

One of the most exciting features of TornadoFX are the Type-Safe Builders. Configuring and laying out controls for complex UI's can be verbose and difficult, and the code can quickly become messy to maintain. Fortunately, you can use a powerful closure pattern pioneered by Groovy to create structured UI layouts with pure and simple Kotlin code.

While we will learn how to apply FXML later in Chapter 10, you may find builders to be an expressive, robust way to create complex UI's in a fraction of the time. There are no configuration files or compiler magic tricks, and builders are done with pure Kotlin code. The next several chapters will divide the builders into separate categories of controls. Along the way, you will gradually build more complex UI's by integrating these builders together.

But first, let's cover how builders actually work.

How Builders Work

Kotlin's standard library comes with a handful of helpful "block" functions to target items of any type `T`. There is the with() function, which allows you to write code against a control as if you were right inside of its class.

```
import javafx.scene.control.Button
import javafx.scene.layout.VBox
import tornadofx.*

class MyView : View() {

    override val root = VBox()

    init {
        with(root) {
            this += Button("Press Me")
        }
    }
}
```

In the above example, the `with()` function accepts the `root` as an argument. The following closure argument manipulates `root` directly by referring to it as `this`, which is safely interpreted as a `VBox`. A `Button` was added to the `VBox` by calling its `plusAssign()` extended operator.

Alternatively, every type in Kotlin has an apply() function. This is almost the same functionality as `with()` but it is presented as an extended higher-order function.

```
import javafx.scene.control.Button
import javafx.scene.layout.VBox
import tornadofx.*

class MyView : View() {

    override val root = VBox()

    init {
        root.apply {
            this += Button("Press Me")
        }
    }
}
```

Both `with()` and `apply()` accomplish a similar task. They safely interpret the type they are targeting and allow manipulations to be done to it. However, `apply()` returns the item it was targeting. Therefore, if you call `apply()` on a `Button` to manipulate, say, its font color and action, it is helpful the `Button` returns itself so as to not break the declaration and assignment flow.

```
import javafx.scene.control.Button
import javafx.scene.layout.VBox
import javafx.scene.paint.Color
import tornadofx.*

class MyView : View() {

    override val root = VBox()

    init {
        with(root) {
            this += Button("Press Me").apply {
                textFill = Color.RED
                action { println("Button pressed!") }
            }
        }
    }
}
```

The basic concepts of how builders work are expressed above, and there are three tasks being done:

1. A `Button` is created
2. The `Button` is modified

3. The `Button` is added to its "parent", which is a `VBox`

When declaring any `Node`, these three steps are so common that TornadoFX streamlines them for you using strategically placed extension functions, such as `button()` as shown below.

```
import javafx.scene.layout.VBox
import javafx.scene.paint.Color
import tornadofx.*

class MyView : View() {

    override val root = VBox()

    init {
        with(root) {
            button("Press Me") {
                textFill = Color.RED
                action { println("Button pressed!") }
            }
        }
    }
}
```

While this looks much cleaner, you might be wondering: "How did we just get rid of the `this +=` and `apply()` function call? And why are we using a function called `button()` instead of an actual `Button`?"

We will not go too deep on how this is done, and you can always dig into the source code if you are curious. But essentially, the `VBox` (or any targetable component) has an extension function called `button()`. It accepts a text argument and an optional closure targeting a `Button` it will instantiate.

When this function is called, it will create a `Button` with the specified text, apply the closure to it, add it to the `VBox` it was called on, and then return it.

Taking this efficiency further, you can override the `root` in a `View`, but assign it a builder function and avoid needing any `init` or `with()` blocks.

```
import javafx.scene.paint.Color
import tornadofx.*

class MyView : View() {

    override val root = vbox {
        button("Press Me") {
            textFill = Color.RED
            action { println("Button pressed!") }
        }
    }
}
```

The builder pattern becomes especially powerful when you start nesting controls into other controls. Using these builder extension functions, you can easily populate and nest multiple `HBox` instances into a `VBox`, and create UI code that is clearly structured (Figure 4.1).

```
import tornadofx.*

class MyView : View() {

    override val root = vbox {
        hbox {
            label("First Name")
            textfield()
        }
        hbox {
            label("Last Name")
            textfield()
        }
        button("LOGIN") {
            useMaxWidth = true
        }
    }
}
```

Figure 4.1

> Also note we will learn about TornadoFX's proprietary `Form` later, which will make simple input UI's like this even simpler to build.

If you need to save references to controls such as the TextFields, you can save them to variables or properties since the functions return the produced controls. Until we learn more robust modeling techniques, it is recommended you use the `singleAssign()` delegates to ensure the properties are only assigned once.

```
import javafx.scene.control.TextField
import tornadofx.*

class MyView : View() {

    var firstNameField: TextField by singleAssign()
    var lastNameField: TextField by singleAssign()

    override val root = vbox {
        hbox {
            label("First Name")
            firstNameField = textfield()
        }
        hbox {
            label("Last Name")
            lastNameField = textfield()
        }
        button("LOGIN") {
            useMaxWidth = true
            action {
                println("Logging in as ${firstNameField.text} ${lastNameField.text}")
            }
        }
    }
}
```

Note that non-builder extension functions and properties have been added to different controls as well. The `useMaxWidth` is an extended property for `Node`, and it sets the `Node` to occupy the maximum width allowed. We will see more of these helpful extensions throughout the next few chapters. We will also see each corresponding builder for each JavaFX control. With the concepts understood above, you can read about these next chapters start to finish or as a reference.

Builders for Basic Controls

The rest of this chapter will cover builders for common JavaFX controls like `Button`, `Label`, and `TextField`. The next chapter will cover builders for data-driven controls like `ListView`, `TableView`, and `TreeTableView`.

Button

For any `Pane`, you can call its `button()` extension function to add a `Button` to it. You can optionally pass a `text` argument and a `Button.() -> Unit` lambda to modify its properties.

This will add a `Button` with red text and print "Button pressed!" every time it is clicked (Figure 4.2)

```
button("Press Me") {
    textFill = Color.RED
    action {
        println("Button pressed!")
    }
}
```

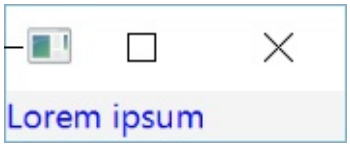
Figure 4.2

Label

You can call the `label()` extension function to add a `Label` to a given `Pane`. Optionally you can provide a text (of type `String` or `Property<String>`), a graphic (of type `Node` or `ObjectProperty<Node>`) and a `Label.() -> Unit` lambda to modify its properties (Figure 4.3).

```
label("Lorem ipsum") {
    textFill = Color.BLUE
}
```

Figure 4.3



TextField

For any target, you can add a `TextField` by calling its `textfield()` extension function (Figure 4.4).

```
textfield()
```

Figure 4.4

You can optionally provide initial text as well as a closure to manipulate the `TextField`. For example, we can add a listener to its `textProperty()` and print its value every time it changes (Figure 4.5).

```
textfield("Input something") {
    textProperty().addListener { obs, old, new ->
        println("You typed: " + new)
    }
}
```

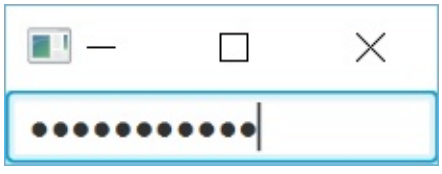
Figure 4.6

PasswordField

If you need a `TextField` to take sensitive information, you might want to consider a `PasswordField` instead. It will show anonymous characters to protect from prying eyes. You can also provide an initial password as an argument and a block to manipulate it (Figure 4.7).

```
passwordfield("password123") {
    requestFocus()
}
```

Figure 4.7

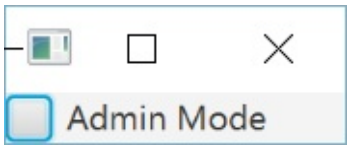


CheckBox

You can create a `CheckBox` to quickly create a true/false state control and optionally manipulate it with a block (Figure 4.8).

```
checkbox("Admin Mode") {
    action { println(isSelected) }
}
```

Figure 4.9



Notice that the action block is wrapped inside the checkbox so you can access its `isSelected` property. If you do not need access to the properties of the `CheckBox`, you can just express it like this.

```
checkbox("Admin Mode").action {
  println(isSelected)
}
```

You can also provide a `Property<Boolean>` that will bind to its selection state.

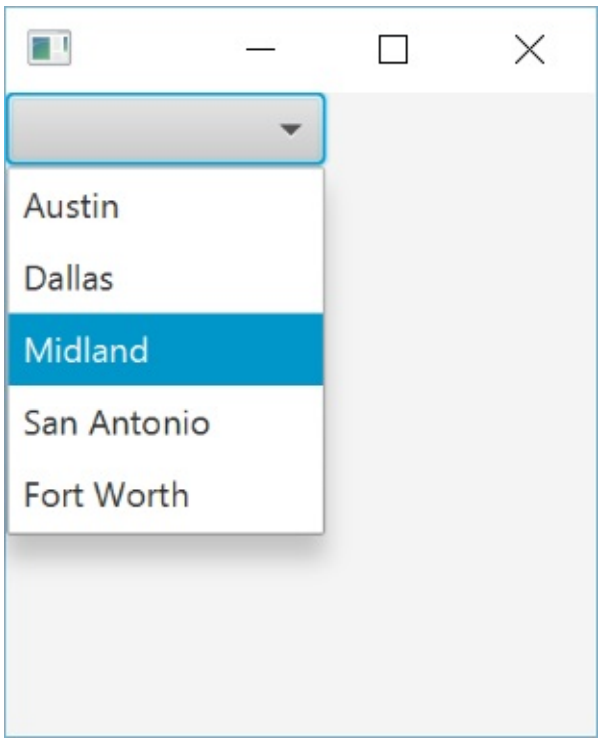
```
val booleanProperty = SimpleBooleanProperty()

checkbox("Admin Mode", booleanProperty).action {
  println(isSelected)
}
```

ComboBox

A `ComboBox` is a drop-down control that allows a fixed set of values to be selected from (Figure 4.10).

```
val texasCities = FXCollections.observableArrayList("Austin",
    "Dallas","Midland", "San Antonio","Fort Worth")

combobox<String> {
    items = texasCities
}
```

Figure 4.10



You do not need to specify the generic type if you declare the `values` as an argument.

```
val texasCities = FXCollections.observableArrayList("Austin",
        "Dallas","Midland","San Antonio","Fort Worth")

combobox(values = texasCities)
```

You can also specify a `Property<T>` to be bound to the selected value.

```
val texasCities = FXCollections.observableArrayList("Austin",
        "Dallas","Midland","San Antonio","Fort Worth")

val selectedCity = SimpleStringProperty()

combobox(selectedCity, texasCities)
```

ToggleButton

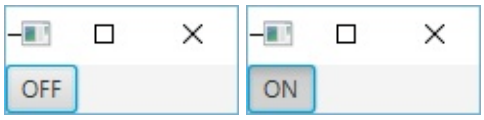
A `ToggleButton` is a button that expresses a true/false state depending on its selection state (Figure 4.11).

```
togglebutton("OFF") {
    action {
        text = if (isSelected) "ON" else "OFF"
    }
}
```

A more idiomatic way to control the button text would be to use a `StringBinding` bound to the `textProperty:`

```
togglebutton {
    val stateText = selectedProperty().stringBinding {
        if (it == true) "ON" else "OFF"
    }
    textProperty().bind(stateText)
}
```

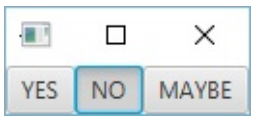
Figure 4.11



You can optionally pass a `ToggleGroup` to the `togglebutton()` function. This will ensure all `ToggleButton`s in that `ToggleGroup` can only have one in a selected state at a time (Figure 4.12).

```
import javafx.scene.control.ToggleGroup
import tornadofx.*

class MyView : View() {

    private val toggleGroup = ToggleGroup()

    override val root = hbox {
        togglebutton("YES", toggleGroup)
        togglebutton("NO", toggleGroup)
        togglebutton("MAYBE", toggleGroup)
    }
}
```

Figure 4.12



RadioButton

A `RadioButton` has the same functionality as a `ToggleButton` but with a different visual style. When it is selected, it "fills" in a circular control (Figure 4.13).

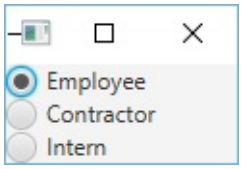
```
radiobutton("Power User Mode") {
    action {
        println("Power User Mode: $isSelected")
    }
}
```

Figure 4.13

Also like the `ToggleButton`, you can set a `RadioButton` to be included in a `ToggleGroup` so that only one item in that group can be selected at a time (Figure 4.14).

```
import javafx.scene.control.ToggleGroup
import tornadofx.*

class MyView : View() {

    private val toggleGroup = ToggleGroup()

    override val root = vbox {
        radiobutton("Employee", toggleGroup)
        radiobutton("Contractor", toggleGroup)
        radiobutton("Intern", toggleGroup)
    }
}
```

Figure 4.14

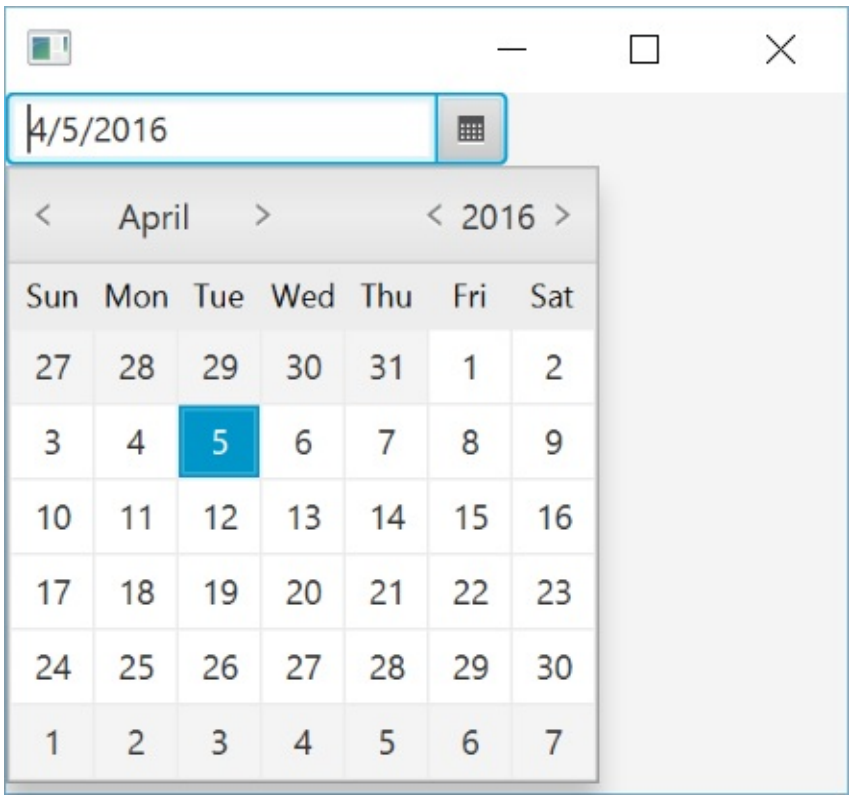


DatePicker

The `DatePicker` allows you to choose a date from a popout calendar control. You can optionally provide a block to manipulate it (Figure 4.15).

```
datepicker {
    value = LocalDate.now()
}
```

Figure 4.15



You can also provide a `Property<LocalDate>` as an argument to bind to its value.

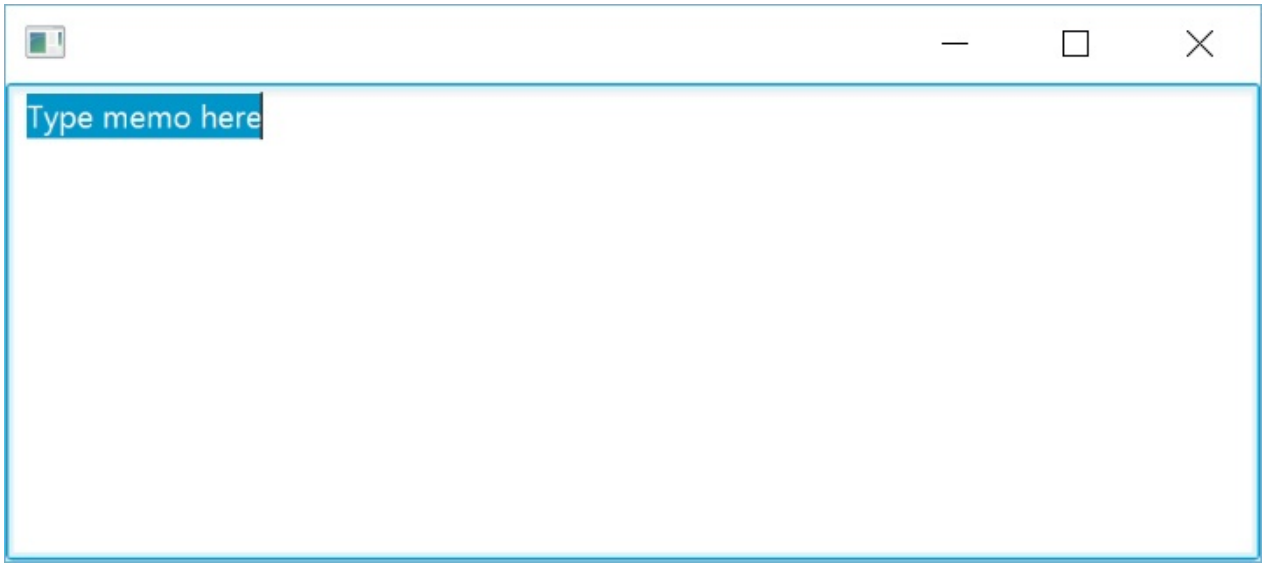
```
val dateProperty = SimpleObjectProperty<LocalDate>()

datepicker(dateProperty) {
    value = LocalDate.now()
}
```

TextArea

The `TextArea` allows you input multiline freeform text. You can optionally provide the initial text `value` as well as a block to manipulate it on declaration (Figure 4.16).

```
textarea("Type memo here") {
    selectAll()
}
```

Figure 4.16



ProgressBar

A `ProgressBar` visualizes progress towards completion of a process. You can optionally provide an initial `Double` value less than or equal to 1.0 indicating percentage of completion (Figure 4.17).

```
progressbar(0.5)
```

Figure 4.17

Here is a more dynamic example simulating progress over a short period of time.

```
progressbar {
    thread {
        for (i in 1..100) {
            Platform.runLater { progress = i.toDouble() / 100.0 }
            Thread.sleep(100)
        }
    }
}
```

You can also pass a `Property<Double>` that will bind the `progress` to its value as well as a block to manipulate the `ProgressBar`.

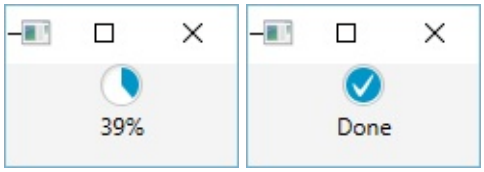
```
progressbar(completion) {
    progressProperty().addListener {
        obsVal, old, new ->  print("VALUE: $new")
    }
}
```

ProgressIndicator

A `ProgressIndicator` is functionally identical to a `ProgressBar` but uses a filling circle instead of a bar (Figure 4.18).

```
progressindicator {
    thread {
        for (i in 1..100) {
            Platform.runLater { progress = i.toDouble() / 100.0 }
            Thread.sleep(100)
        }
    }
}
```

Figure 4.18



Just like the `ProgressBar` you can provide a `Property<Double>` and/or a block as optional arguments (Figure 4.19).

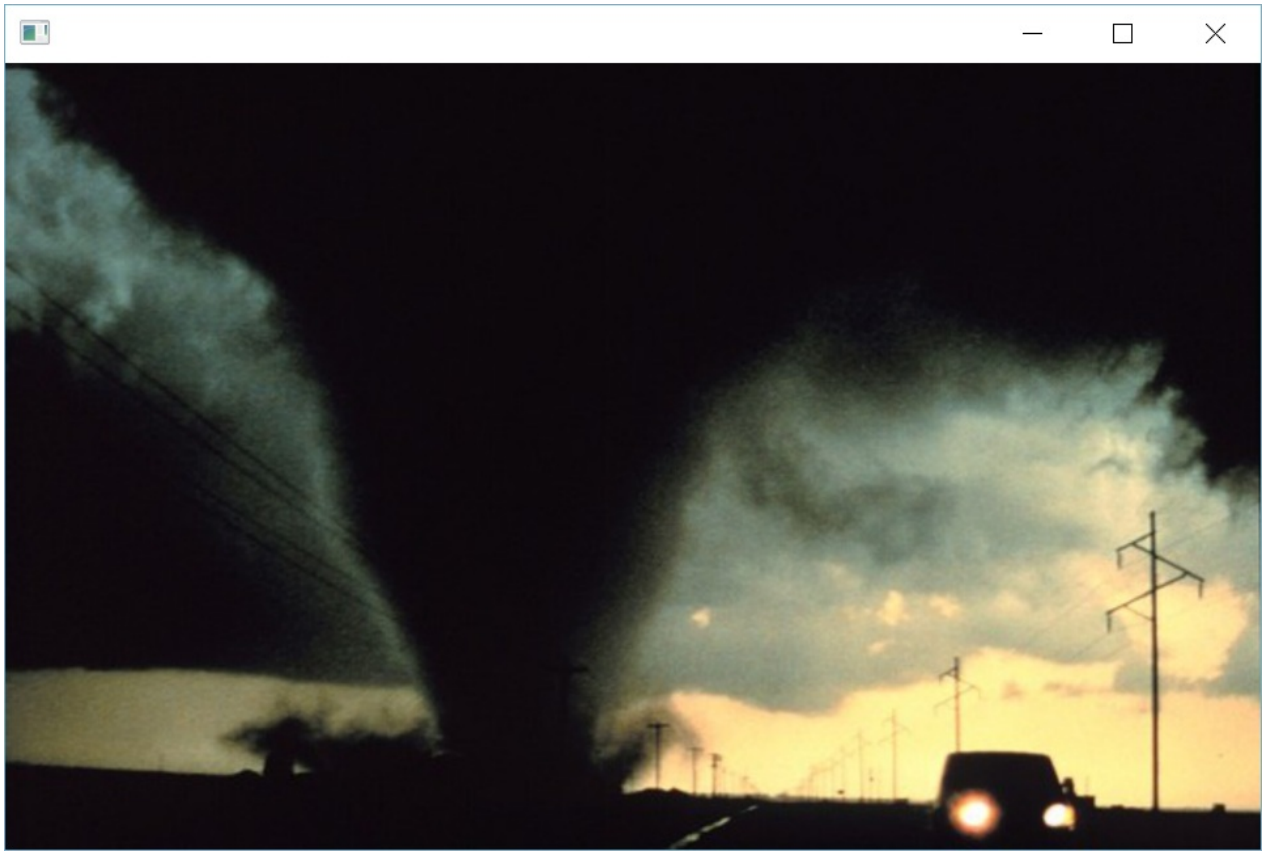
```
val completion = SimpleObjectProperty(0.0)
progressindicator(completion)
```

ImageView

You can embed an image using `imageview()` .

```
imageview("tornado.jpg")
```

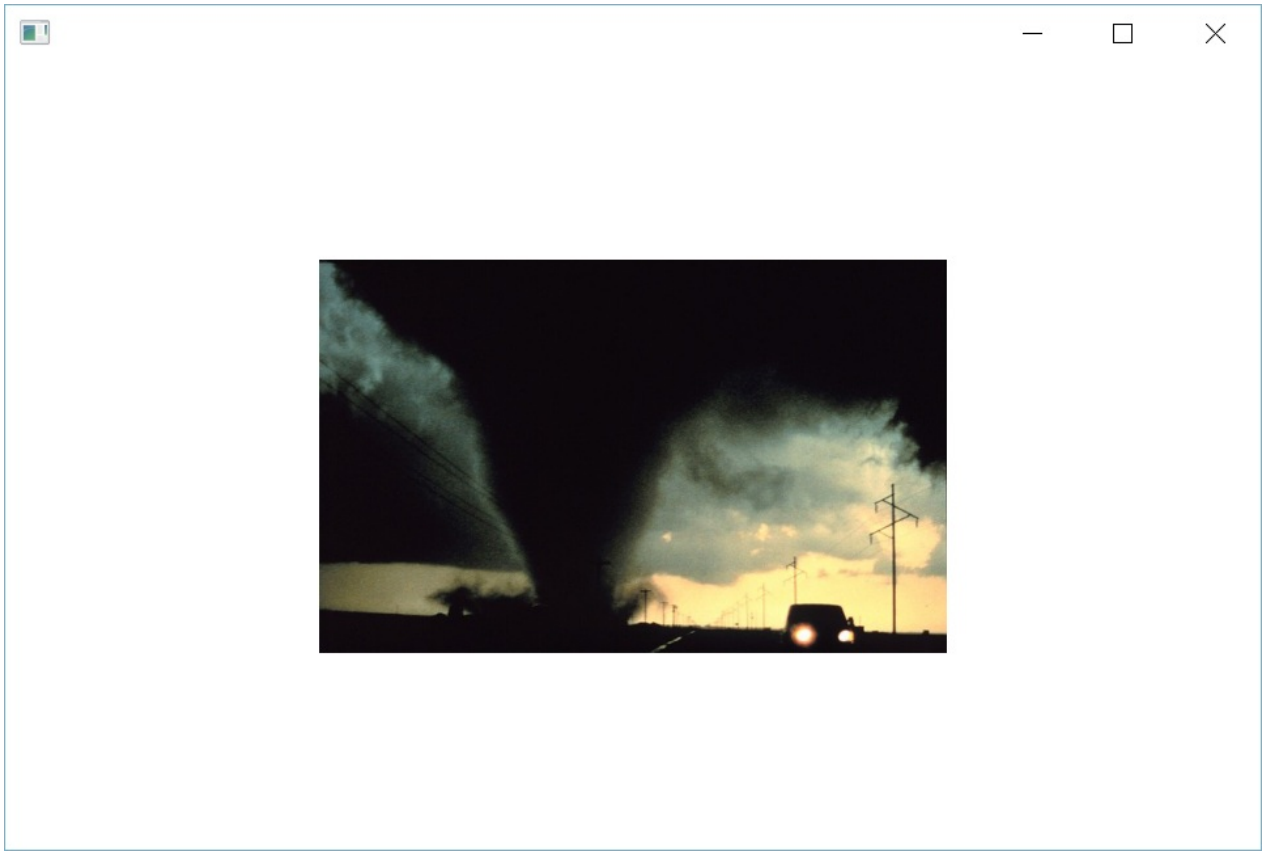
Figure 4.19



Like most other controls, you can use a block to modify its attributes (Figure 4.20).

```
imageview("tornado.jpg") {
    scaleX = .50
    scaleY = .50
}
```

Figure 4.20



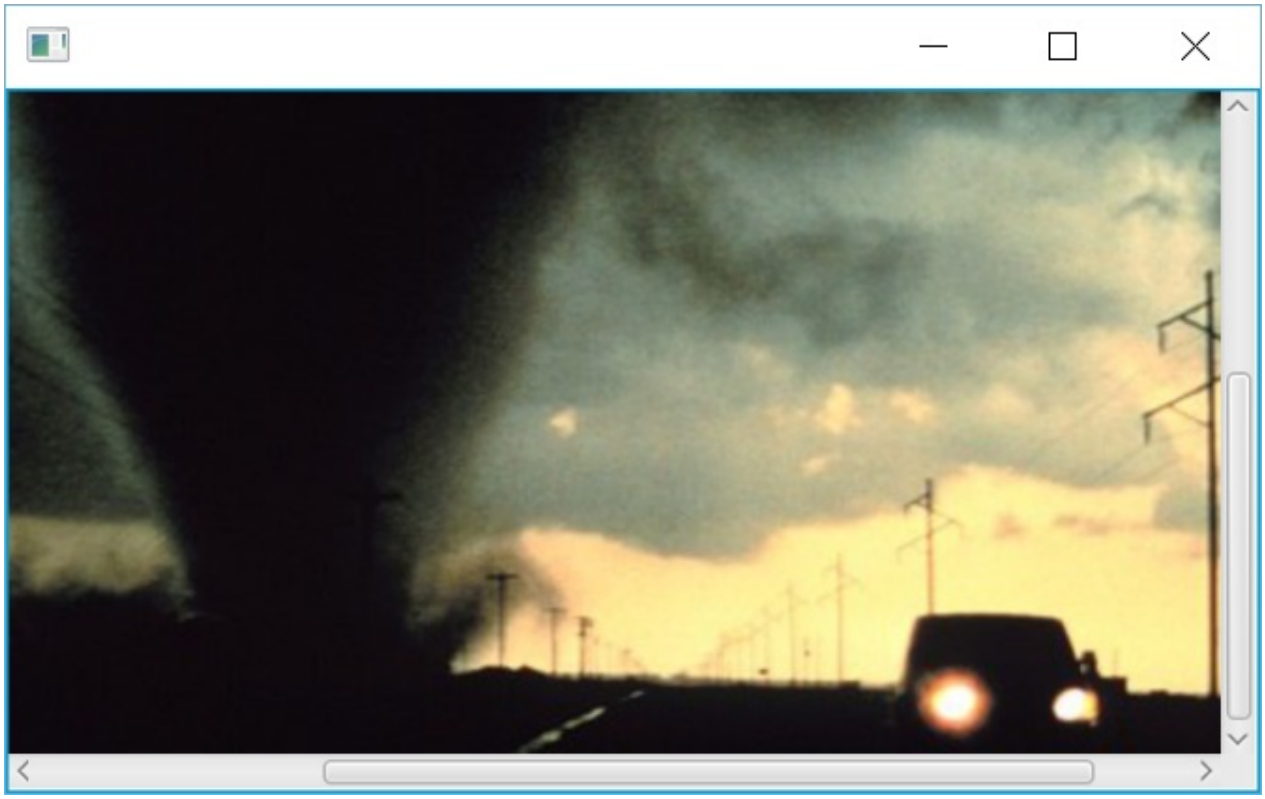
ScrollPane

You can embed a control inside a `ScrollPane` to make it scrollable. When the available area becomes smaller than the control, scrollbars will appear to navigate the control's area.

For instance, you can wrap an `ImageView` inside a `ScrollPane` (Figure 4.21).

```
scrollpane {
    imageview("tornado.jpg")
}
```

Figure 4.21



Keep in mind that many controls like `TableView` and `TreeTableView` already have scroll bars on them, so wrapping them in a `ScrollPane` is not necessary (Figure 4.22).

Hyperlink

You can create a `Hyperlink` control to mimic the behavior of a typical hyperlink to a file, a website, or simply perform an action.

```
hyperlink("Open File").action { println("Opening file...") }
```

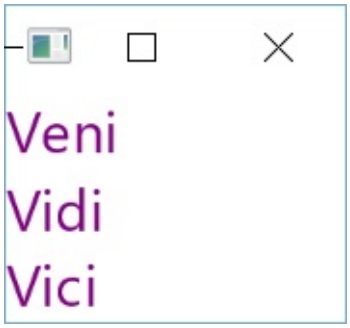
Figure 4.22

Text

You can add a simple piece of `Text` with formatted properties. This control is simpler and rawer than a `Label`, and paragraphs can be separated using `\n` characters (Figure 4.23).

```
text("Veni\nVidi\nVici") {
    fill = Color.PURPLE
    font = Font(20.0)
}
```

Figure 4.23



TextFlow

If you need to concatenate multiple pieces of text with different formats, the `TextFlow` control can be helpful (Figure 4.24).

```
textflow {
    text("Tornado") {
        fill = Color.PURPLE
        font = Font(20.0)
    }
    text("FX") {
        fill = Color.ORANGE
        font = Font(28.0)
    }
}
```

Figure 4.24

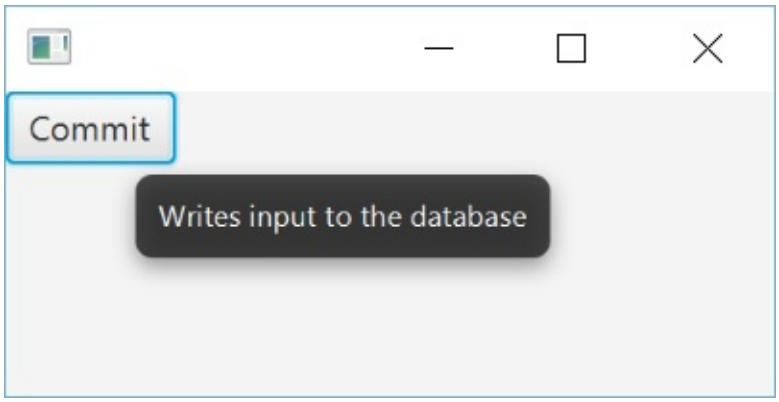
You can add any `Node` to the `textflow`, including images, using the standard builder functions.

Tooltips

Inside any `Node` you can specify a `Tooltip` via the `tooltip()` function (Figure 4.25).

```
button("Commit") {
    tooltip("Writes input to the database")
}
```

Figure 4.25



Like most other builders, you can provide a closure to customize the `Tooltip` itself.

```
button("Commit") {
    tooltip("Writes input to the database") {
        font = Font.font("Verdana")
    }
}
```

Shortcuts and Key Combinations

You can fire actions when certain key combinations are typed. This is done with the `shortcut` function:

```
shortcut(KeyCombination.valueOf("Ctrl+Y")) {
    doSomething()
}
```

There is also a string version of the `shortcut` function that does the same but is less verbose:

```
shortcut("Ctrl+Y")) {
    doSomething()
}
```

You can also add shortcuts to button actions directly:

```
button("Save") {
    action { doSave() }
    shortcut("Ctrl+S")
}
```

Touch Support

JavaFX supports touch out of the box, and TornadoFX makes a few improvements especially for shortpress and longpress durations. It consists of two functions similar to `action`, which can be configured on any `Node`:

```
shortpress { println("Activated on short press") }
longpress { println("Activated on long press") }
```

Both functions accepts a `consume` parameter which by default is `false`. Setting it to true will prevent event bubbling for the press event. The `longpress` function additionally supports a `threshold` parameter which is used to determine when a `longpress` has occurred. It is `700.millis` by default.

SUMMARY

In this chapter we learned about TornadoFX builders and how they work simply by using Kotlin extension functions. We also covered builders for basic controls like `Button`, `TextField` and `ImageView`. In the coming chapters we will learn about builders for tables, layouts, menus, charts, and other controls. As you will see, combining all these builders together creates a powerful way to express complex UI's with very structured and minimal code.

There are many other builder controls, and the maintainers of TornadoFX have strived to create a builder for every JavaFX control. If you need something that is not covered here, use Google to see if its included in JavaFX. Chances are if a control is available in JavaFX, there is a builder with the same name in TornadoFX.

These are not the only control builders in the TornadoFX API, and this guide does its best to keep up. Always check the TornadoFX GitHub to see the latest builders and functionalities available, and file an issue if you see any missing.

We are not done covering builders yet though. In the next section, we will cover more complex controls in the next few sections.

Data Controls

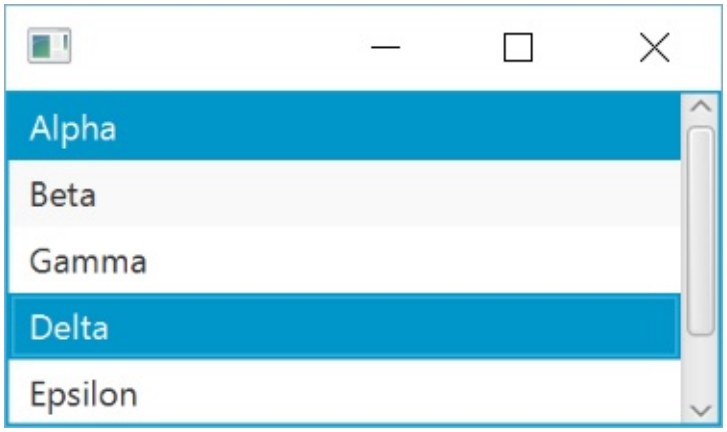
Any significant application works with data, and providing a means for users to view, manipulate, and modify data is not a trivial task for user interface development. Fortunately, TornadoFX streamlines many JavaFX data controls such as `ListView`, `TableView`, `TreeView`, and `TreeTableView`. These controls can be cumbersome to set up in a purely object-oriented way. But using builders through functional declarations, we can code all these controls in a much more streamlined way.

ListView

A `ListView` is similar to a `ComboBox` but it displays all items within a `ScrollView` and has the option of allowing multiple selections, as shown in Figure 5.1

```
listview<String> {
    items.add("Alpha")
    items.add("Beta")
    items.add("Gamma")
    items.add("Delta")
    items.add("Epsilon")
    selectionModel.selectionMode = SelectionMode.MULTIPLE
}
```

Figure 5.1



You can also provide it an `ObservableList` of items up front and omit the type declaration since it can be inferred. Using an `ObservableList` has the benefit that changes to the list will automatically be reflected in the `ListView`.

```
val greekLetters = listOf("Alpha","Beta",
        "Gamma","Delta","Epsilon").observable()

listview(greekLetters) {
    selectionModel.selectionMode = SelectionMode.MULTIPLE
}
```

Like most data controls, keep in mind that by default the `ListView` will call `toString()` to render the text for each item in your domain class. To render anything else, you will need to create your own custom cell formatting.

> To read about custom cell formatting and nodes for a `ListView`, read Appendix A3 - Custom Cell Formatting in ListView

TableView

Probably one of the most significant builders in TornadoFX is the one for `TableView`. If you have worked with JavaFX, you might have experienced building a `TableView` in an object-oriented way. But TornadoFX provides a functional declaration construct pattern using extension functions that greatly simplify the coding of a `TableView`.

Say you have a domain type, such as `Person`.

```
class Person(val id: Int, val name: String, val birthday: LocalDate) {
    val age: Int get() = Period.between(birthday, LocalDate.now()).years
}
```

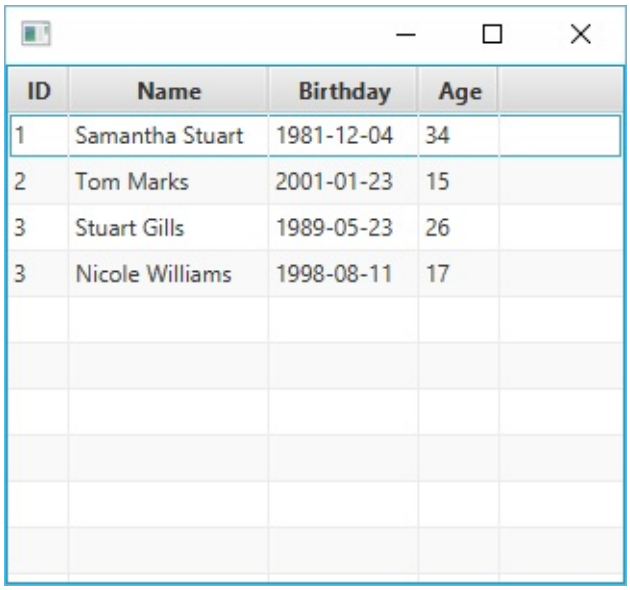
Take several instances of `Person` and put them in an `ObservableList`.

```
private val persons = listOf(
        Person(1,"Samantha Stuart",LocalDate.of(1981,12,4)),
        Person(2,"Tom Marks",LocalDate.of(2001,1,23)),
        Person(3,"Stuart Gills",LocalDate.of(1989,5,23)),
        Person(3,"Nicole Williams",LocalDate.of(1998,8,11))
).observable()
```

You can quickly declare a `TableView` with all of its columns using a functional construct, and specify the `items` property to an `ObservableList<Person>` (Figure 5.3).

```
tableview(persons) {
    column("ID",Person::id)
    column("Name", Person::name)
    column("Birthday", Person::birthday)
    column("Age",Person::age)
}
```

Figure 5.3



The `column()` functions are extension functions for `TableView` accepting a `header` name and a mapped property using reflection syntax. TornadoFX will then take each mapping to render a value for each cell in that given column.

> If you want granular control over `TableView` column resize policies, see Appendix A2 for more information on `SmartResize` policies.

Using "Property" properties

If you follow the JavaFX `Property` conventions to set up your domain class, it will automatically support value editing.

You can create these `Property` objects the conventional way, or you can use TornadoFX's `property` delegates to automatically create these `Property` declarations as shown below.

```
class Person(id: Int, name: String, birthday: LocalDate) {
    var id by property(id)
    fun idProperty() = getProperty(Person::id)

    var name by property(name)
    fun nameProperty() = getProperty(Person::name)

    var birthday by property(birthday)
    fun birthdayProperty() = getProperty(Person::birthday)

    val age: Int get() = Period.between(birthday, LocalDate.now()).years
}
```

You need to create `xxxProperty()` functions for each property to support JavaFX's naming convention when it uses reflection. This can easily be done by relaying their calls to `getProperty()` to retrieve the `Property` for a given field. See Appendix A1 for detailed information on how these property delegates work.

Now on the `TableView`, you can make it editable, map to the properties, and apply the appropriate cell-editing factories to make the values editable.

```
override val root = tableview(persons) {
    isEditable = true
    column("ID",Person::idProperty).makeEditable()
    column("Name", Person::nameProperty).makeEditable()
    column("Birthday", Person::birthdayProperty).makeEditable()
    column("Age",Person::age)
}
```

To allow editing and rendering, TornadoFX provides a few default cell factories you can invoke on a column easily through extension functions.

Extension Function	Description
useTextField()	Uses a standard `TextField` to edit values with a provided `StringConverter`
useComboBox()	Edits a cell value via a `ComboBox` with a specified `ObservableList<T>` of applicable values
useChoiceBox()	Accepts value changes to a cell with a `ChoiceBox`
useCheckBox()	Renders an editable `CheckBox` for a `Boolean` value column
useProgressBar()	Renders the cell as a `ProgressBar` for a `Double` value column

Property Syntax Alternatives

If you do not care about exposing the `Property` in a function (which is common in practial usage) you can express your class like this:

```
class Person(id: Int, name: String, birthday: LocalDate) {
    val idProperty = SimpleIntegerProperty(id)
    var id by idProperty

    val nameProperty = SimpleStringProperty(name)
    var name by nameProperty

    val birthdayProperty = SimpleObjectProperty(birthday)
    var birthday by birthdayProperty

    val age: Int get() = Period.between(birthday, LocalDate.now()).years
}
```

This alternative pattern exposes the `Property` as a field member instead of a function. If you like the above syntax but want to keep the function, you can make the property `private` and add the function like this:

```
private val nameProperty = SimpleStringProperty(name)
fun nameProperty() = nameProperty
var name by nameProperty
```

Choosing from these patterns are all a matter of taste, and you can use whatever version meets your needs or preferences best.

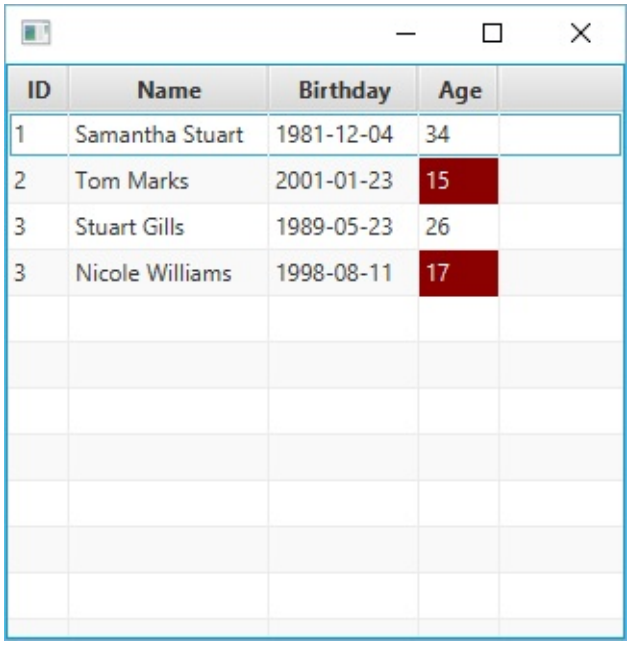
> You can also convert plain properties to JavaFX properties using the TornadoFX Plugin. Refer to Chapter 13 to learn how to do this.

Using cellFormat()

There are other extension functions applied to `TableView` that can assist the flow of declaring a `TableView`. For instance, you can call a `cellFormat()` function on a given column to apply formatting rules, such as highlighting "Age" values less than 18 (Figure 5.4).

```
tableview(persons) {
    column("ID", Person::id)
    column("Name", Person::name)
    column("Birthday", Person::birthday)
    column("Age", Person::age).cellFormat {
        text = it.toString()
        style {
            if (it < 18) {
                backgroundColor += c("#8b0000")
                textFill = Color.WHITE
            } else {
                backgroundColor += Color.WHITE
                textFill = Color.BLACK
            }
        }
     }
}
```

Figure 5.4



Accessing Nested Properties

Let's assume our `Person` object has a `parent` property which is also of of type `Person`. To create a column for the parent name, we have several options. Our first attempt is simply extracting the name property manually:

```
column<Person, String>("Parent name", { it.value.parentProperty.value.nameProperty })
```

Notice how we cannot simply reference the property. We need to access the value provided in the callback to get to the actual instance and nest call to the `nameProperty`. While this works, it has one major drawback. If the parent changes, the `TableView` will not be updated. We can partially remedy this by defining the value for the property as the parent itself, and formatting its name:

```
column("Parent name", Person::parentProperty).cellFormat {
    textProperty().bind(it.parentProperty.value.nameProperty)
}
```

It might still not update right away, even though it would eventually become consistent as the `TableView` refreshes.

To create a binding that would reflect a change to the parent property immediately, consider using a select binding, which we will cover later.

```
column<Person, String>("Parent name", { it.value.parentProperty.select(Person::namePro
perty) })
```

Declaring Column Values Functionally

If you need to map a column's value to a non-property (such as a function), you can use a non-reflection means to extract the values for that column.

Say you have a `WeeklyReport` type that has a `getTotal()` function accepting a `DayOfWeek` argument (an enum of Monday, Tuesday... Sunday).

```
abstract class WeeklyReport(val startDate: LocalDate) {
    abstract fun getTotal(dayOfWeek: DayOfWeek): BigDecimal
}
```

Let's say you wanted to create a column for each `DayOfWeek`. You cannot map to properties, but you can map each `WeeklyReport` item explicitly to extract each value for that `DayOfWeek`.

```
tableview<WeeklyReport> {
    for (dayOfWeek in DayOfWeek.values()) {
        column<WeeklyReport, BigDecimal>(dayOfWeek.toString()) {
            ReadOnlyObjectWrapper(it.value.getTotal(dayOfWeek))
        }
    }
}
```

> This more closely resembles the traditional `setCellValueFactory()` for the JavaFX `TableColumn`.

Row Expanders

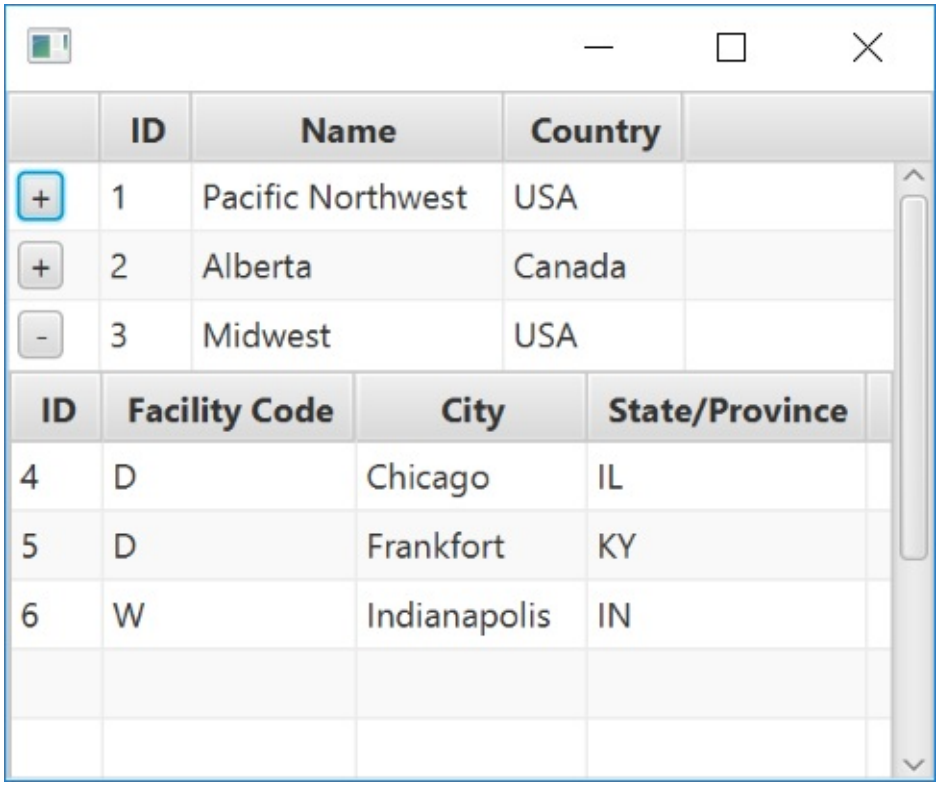
Later we will learn about the `TreeTableView` which has a notion of "parent" and "child" rows, but the constraint with this control is the parent and child must have the same columns. Fortunately, TornadoFX comes with an awesome utility to not only reveal a "child table" for a given row, but any kind of `Node` control.

Say we have two domain types: `Region` and `Branch`. A `Region` is a geographical zone, and it contains one or more `Branch` items which are specific business operation locations (warehouses, distribution centers, etc). Here is a declaration of these types and some given instances.

```
class Region(val id: Int, val name: String, val country: String, val branches: Observa
bleList<Branch>)

class Branch(val id: Int, val facilityCode: String, val city: String, val stateProvince
: String)

val regions = listOf(
        Region(1,"Pacific Northwest", "USA",listOf(
                Branch(1,"D","Seattle","WA"),
                Branch(2,"W","Portland","OR")
        ).observable()),
        Region(2,"Alberta", "Canada",listOf(
                Branch(3,"W","Calgary","AB")
        ).observable()),
        Region(3,"Midwest", "USA", listOf(
                Branch(4,"D","Chicago","IL"),
                Branch(5,"D","Frankfort","KY"),
                Branch(6, "W","Indianapolis", "IN")
        ).observable())
).observable()
```

We can create a `TableView` where each row has a `rowExpander()` function defined, and there we can arbitrarily create any `Node` control built off that particular row's item. In this case, we can nest another `TableView` for a given `Region` to show all the `Branch` items belonging to it. It will have a "+" button column to expand and show this expanded control (Figure 5.5).

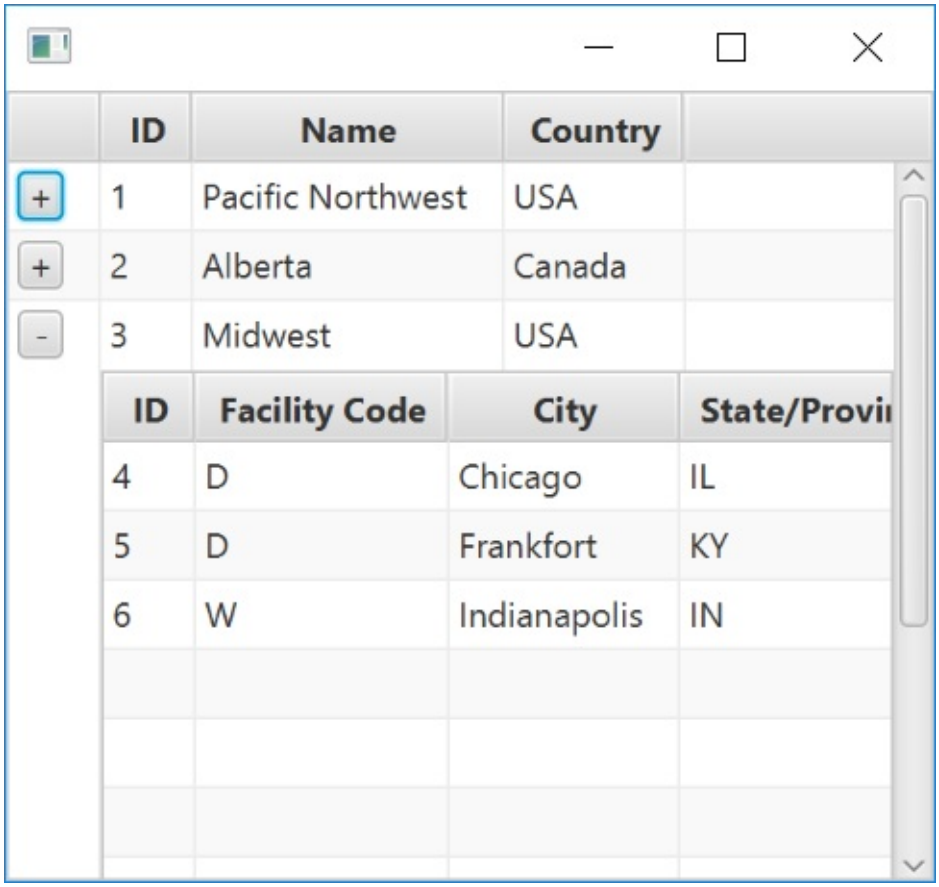
Figure 5.5



There are a few configurability options, like "expand on double-click" behaviors and accessing the `expanderColumn` (the column with the "+" button) to drive a padding (Figure 5.6).

```
override val root = tableview(regions) {
        column("ID",Region::id)
        column("Name", Region::name)
        column("Country", Region::country)
        rowExpander(expandOnDoubleClick = true) {
            paddingLeft = expanderColumn.width
            tableview(it.branches) {
                column("ID",Branch::id)
                column("Facility Code",Branch::facilityCode)
                column("City",Branch::city)
                column("State/Province",Branch::stateProvince)
            }
        }
    }
```

Figure 5.6



The `rowExpander()` function does not have to return a `TableView` but any kind of `Node`, including Forms and other simple or complex controls.

Accessing the Expander Column

You might want to manipulate or call functions on the actual expander column. If you activate expand on double click, you might not want to show the expander column in the table at all. First we need a reference to the expander:

```
val expander = rowExpander(true) { ... }
```

If you want to hide the expander column, just call `expander.isVisible = false`. You can also programmatically toggle the expanded state of any column by calling `expander.toggleExpanded(rowIndex)`.

TreeView

The `TreeView` contains elements where each element may contain child elements. Typically arrows allow you to expand a parent element to see its children. For instance, we can nest employees under department names

Traditionally in JavaFX, populating these elements is rather cumbersome and verbose. Fortunately TornadoFX makes it relatively simple.

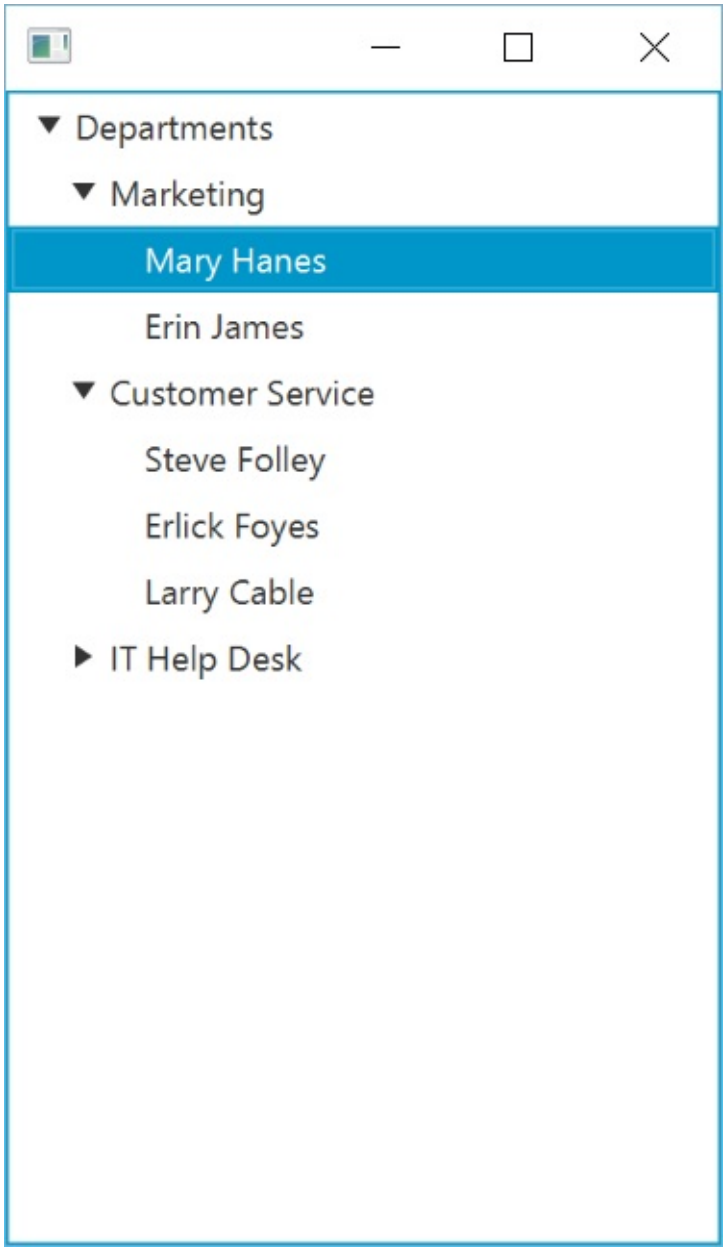
Say you have a simple type `Person` and an `ObservableList` containing several instances.

```
data class Person(val name: String, val department: String)

val persons = listOf(
        Person("Mary Hanes","Marketing"),
        Person("Steve Folley","Customer Service"),
        Person("John Ramsy","IT Help Desk"),
        Person("Erlick Foyes","Customer Service"),
        Person("Erin James","Marketing"),
        Person("Jacob Mays","IT Help Desk"),
        Person("Larry Cable","Customer Service")
        )
```

Creating a `TreeView` with the `treeview()` builder can be done functionally Figure 5.7).

```
// Create Person objects for the departments
// with the department name as Person.name

val departments = persons
    .map { it.department }
    .distinct().map { Person(it, "") }

treeview<Person> {
    // Create root item
    root = TreeItem(Person("Departments", ""))

    // Make sure the text in each TreeItem is the name of the Person
    cellFormat { text = it.name }

    // Generate items. Children of the root item will contain departments
    populate { parent ->
        if (parent == root) departments else persons.filter { it.department == parent.
value.name }
    }
}
```

Figure 5.7



Let's break this down:

```
val departments = persons
    .map { it.department }
    .distinct().map { Person(it, "") }
```

First we gather a distinct list of all the `departments` derived from the `persons` list. But then we put each `department` String in a `Person` object since the `TreeView` only accepts `Person` elements. While this is not very intuitive, this is the constraint and design of `TreeView`. We must make each `department` a `Person` for it to be accepted.

```
treeview<Person> {
    // Create root item
    root = TreeItem(Person("Departments", ""))
```

Next we specify the highest `root` for the `TreeView` that all departments will be nested under, and we give it a placeholder `Person` called "Departments".

```
    cellFormat { text = it.name }
```

Then we specify the `cellFormat()` to render the `name` of each `Person` (including departments) on each cell.

```
   populate { parent ->
        if (parent == root) departments else persons.filter { it.department == parent.
value.name }
    }
```

Finally, we call the `populate()` function and provide a block instructing how to provide children to each `parent`. If the `parent` is indeed the `root`, then we return the `departments`. Otherwise the `parent` is a `department` and we provide a list of `Person` objects belonging to that `department`.

Data driven TreeView

If the child list you return from `populate` is an `ObservableList`, any changes to that list will automatically be reflected in the TreeView. The populate function will be called for any new children that appears, and
removed items will result in removed TreeItems as well.

TreeView with Differing Types

It is not necessarily intuitive to make every entity in the previous example a `Person`. We made each department a `Person` as well as the `root` "Departments". For a more complex `TreeView<T>` where `T` is unknown and can be any number of types, it is better to specify type `T` as `Any`.

Using star projection, you can safely populate multiple types nested into the `TreeView`.

For instance, you can create a `Department` type and leverage `cellFormat()` to utilize type-checking for rendering. Then you can use a `populate()` function that will iterate over each element, and you specify the children for each element (if any).

```
data class Department(val name: String)

// Create Department objects for the departments by getting distinct values from Perso
n.department
val departments = persons.map { it.department }.distinct().map { Department(it) }

// Type safe way of extracting the correct TreeItem text
cellFormat {
    text = when (it) {
        is String -> it
        is Department -> it.name
        is Person -> it.name
        else -> throw IllegalArgumentException("Invalid value type")
    }
}

// Generate items. Children of the root item will contain departments, children of dep
artments are filtered
populate { parent ->
    val value = parent.value
    if (parent == root) departments
    else if (value is Department) persons.filter { it.department == value.name }
    else null
}
```

TreeTableView

The `TreeTableView` operates and functions similarly to a `TreeView`, but it has multiple columns since it is a table. Please note that the columns in a `TreeTableView` are the same for each parent and child element. If you want the columns to be different between parent and child, use a `TableView` with a `rowExpander()` as covered earlier in this chapter.

Say you have a `Person` class that optionally has an `employees` parameter, which defaults to an empty `List<Person>` if nobody reports to that `Person`.

```
class Person(val name: String,
  val department: String,
  val email: String,
  val employees: List<Person> = emptyList())
```

Then you have an `ObservableList<Person>` holding instances of this class.

```
val persons = listOf(
        Person("Mary Hanes", "IT Administration", "mary.hanes@contoso.com", listOf(
            Person("Jacob Mays", "IT Help Desk", "jacob.mays@contoso.com"),
            Person("John Ramsy", "IT Help Desk", "john.ramsy@contoso.com"))),
        Person("Erin James", "Human Resources", "erin.james@contoso.com", listOf(
            Person("Erlick Foyes", "Customer Service", "erlick.foyes@contoso.com"),
            Person("Steve Folley", "Customer Service", "steve.folley@contoso.com"),
            Person("Larry Cable", "Customer Service", "larry.cable@contoso.com")))
).observable()
```

You can create a `TreeTableView` by merging the components needed for a `TableView` and `TreeView` together. You will need to call the `populate()` function as well as set the root `TreeItem`.

```
val treeTableView = TreeTableView<Person>().apply {
    column("Name", Person::nameProperty)
    column("Department", Person::departmentProperty)
    column("Email", Person::emailProperty)

    /// Create the root item that holds all top level employees
    root = TreeItem(Person("Employees by leader", "", "", persons))

    // Always return employees under the current person
    populate { it.value.employees }

    // Expand the two first levels
    root.isExpanded = true
    root.children.forEach { it.isExpanded = true }

    // Resize to display all elements on the first two levels
    resizeColumnsToFitContent()
}
```

It is also possible to work with more of an ad hoc backing store like a `Map`. That would look something like this:

```
val tableData = mapOf(
    "Fruit" to arrayOf("apple", "pear", "Banana"),
    "Veggies" to arrayOf("beans", "cauliflower", "cale"),
    "Meat" to arrayOf("poultry", "pork", "beef")
)

treetableview<String>(TreeItem("Items")) {
    column<String, String>("Type", { it.value.valueProperty() })
    populate {
        if (it.value == "Items") tableData.keys
        else tableData[it.value]?.asList()
    }
}
```

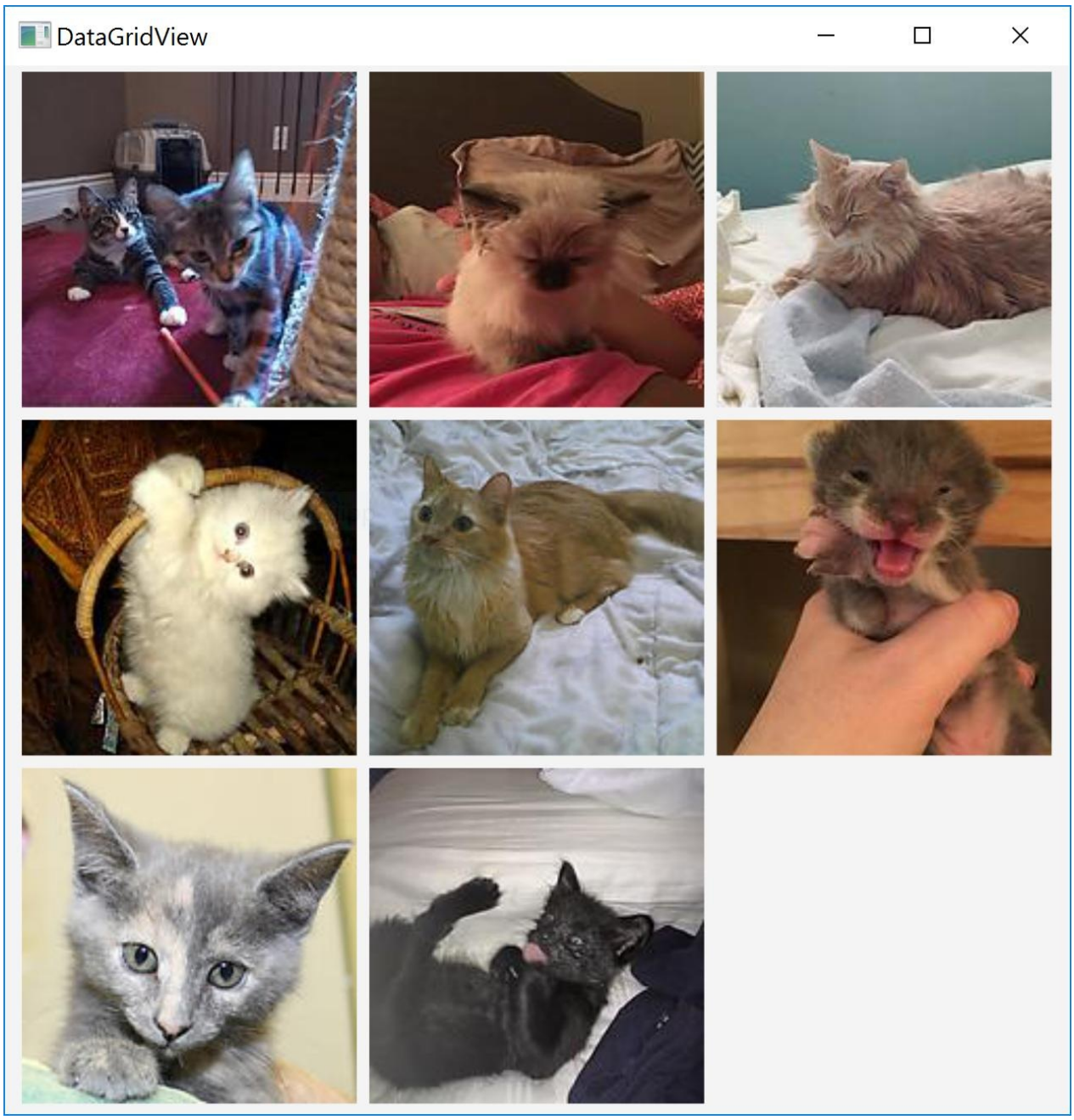
DataGrid

A `DataGrid` is similar to the `GridPane` in that it displays items in a flexible grid of rows and columns, but the similarities ends there. While the `GridPane` requires you to add Nodes to the children list, the `DataGrid` is data driven in the same way as `TableView` and `ListView`. You supply it with a list of items and tell it how to convert those children to a graphical representation.

It supports selection of either a single item or multiple items at a time so it can be used as for example the display of an image viewer or other components where you want a visual representation of the underlying data. Usage wise it is close to a `ListView`, but you can create an arbitrary scene graph inside each cell so it is easy to visualize multiple properties for each item.

```
val kittens = listOf("http://i.imgur.com/DuFZ6PQb.jpg", "http://i.imgur.com/o2QoeNnb.j
pg") // more items here

datagrid(kittens) {
    cellCache {
         imageview(it)
    }
}
```

Figure 5.8

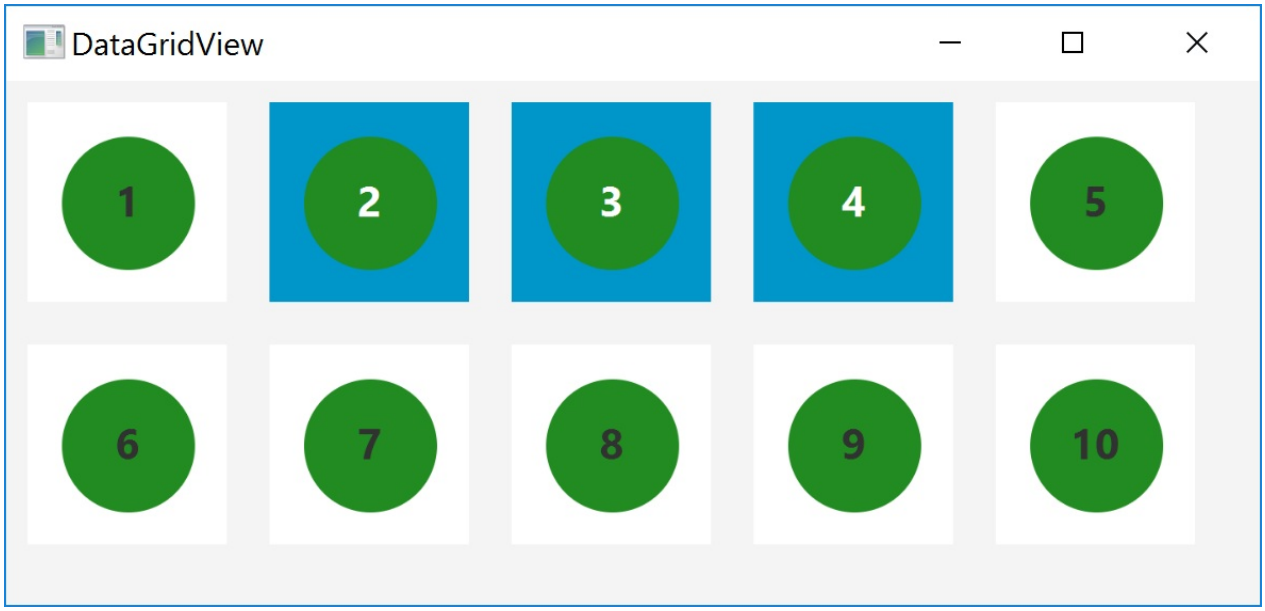


The `cellCache` function receives each item in the list, and since we used a list of Strings in our example, we simply pass that string to the `imageview()` builder to create an `ImageView` inside each table cell. It is important to call the `cellCache` function instead of the `cellFormat` function to avoid recreating the images every time the `DataGrid` redraws. It will reuse the items.

Let's create a scene graph that is a little bit more involved, and also change the default size of each cell:

```
val numbers = (1..10).toList()

datagrid(numbers) {
    cellHeight = 75.0
    cellWidth = 75.0

    multiSelect = true

    cellCache {
        stackpane {
            circle(radius = 25.0) {
                fill = Color.FORESTGREEN
            }
            label(it.toString())
        }
    }
}
```

Figure 5.9



The grid is supplied with a list of numbers this time. We start by specifying a cell height and width of 75 pixels, half of the default size. We also configure multi select to be able to select more than a single element. This is a shortcut of writing `selectionModel.selectionMode = SelectionMode.MULTIPLE` via an extension property. We create a `StackPane` that stacks a `Label` on top of a `Circle` .

> You might wonder why the label got so big and bold by default. This is coming from the default stylesheet. The stylesheet is a good starting point for further customization. All properties of the data grid can be configured in code as well as in CSS, and the stylesheet lists all possible style properties.
>
> The number list showcased multiple selection. When a cell is selected, it receives the CSS pseudo class of `selected`. By default it will behave mostly like a `ListView` row with regards to selection styles. You can access the `selectionModel` of the data grid to listen for selection changes, see what items are selected etc.

Summary

Functional constructs work well with data controls like `TableView`, `TreeView`, and others we have seen in this chapter. Using the builder patterns, you can quickly and functionally declare how data is displayed.

In Chapter 7, we will embed controls in layouts to create more complex UI's easily.

Type-Safe CSS

While you can create plain text CSS style sheets in JavaFX, TornadoFX provides the option to bring type-safety and compiled CSS to JavaFX. You can conveniently choose to create styles in its own class, or do it inline within a control declaration.

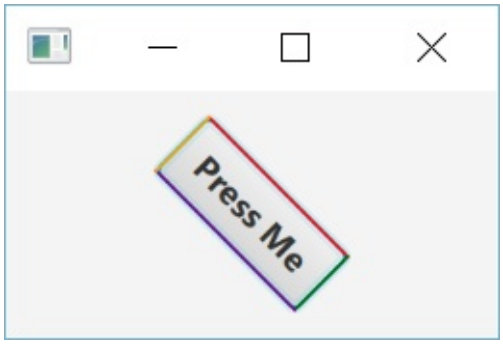
Inline CSS

The quickest and easiest way to style a control on the fly is to call a given `Node`'s inline `style { }` function. All the CSS properties available on a given control are available in a type-safe manner, with compilation checks and auto-completion.

For example, you can style the borders on a `Button` (using the `box()` function), bold its font, and rotate it (Figure 6.1).

```
button("Press Me") {
    style {
        fontWeight = FontWeight.EXTRA_BOLD
        borderColor += box(
                top = Color.RED,
                right = Color.DARKGREEN,
                left = Color.ORANGE,
                bottom = Color.PURPLE
        )
        rotate = 45.deg
    }

    setOnAction { println("You pressed the button") }
}
```

Figure 6.1



This is especially helpful when you want to style a control without breaking the declaration flow of the `Button`. However, keep in mind the `style { }` will replace all styles applied to that control unless you pass `true` for its optional `append` argument.

```
style(append = true) {
      ....
}
```

Some times you want to apply the same styles to many nodes in one go. The `style { }` function can also be applied to any Iterable that contains Nodes:

```
vbox {
    label("First")
    label("Second")
    label("Third")
    children.style {
        fontWeight = FontWeight.BOLD
    }
}
```

The `fontWeight` style is applied to all children of the vbox, in essence all the labels we added.

When your styling complexity passes a certain threshold, you may want to consider using Stylesheets which we will cover next.

Applying Style Classes with Stylesheets

If you want to organize, re-use, combine, and override styles you need to leverage a `Stylesheet`. Traditionally in JavaFX, a stylesheet is defined in a plain CSS text file included in the project. However, TornadoFX allows creating stylesheets with pure Kotlin code. This has the benefits of compilation checks, auto-completion, and other perks that come with statically typed code.

To declare a `Stylesheet`, extend it onto your own class to hold your customized styles.

```
import tornadofx.*

class MyStyle: Stylesheet() {
}
```

Next, you will want to specify its `companion object` to hold class-level properties that can easily be retrieved. Declare a new `cssclass()`-delegated property called `tackyButton`, and define four colors we will use for its borders.

```
import javafx.scene.paint.Color
import tornadofx.*

class MyStyle: Stylesheet() {

    companion object {
        val tackyButton by cssclass()

        private val topColor = Color.RED
        private val rightColor = Color.DARKGREEN
        private val leftColor = Color.ORANGE
        private val bottomColor = Color.PURPLE
    }
}
```

Note also you can use the `c()` function to build colors quickly using RGB values or color Strings.

```
  private val topColor = c("#FF0000")
  private val rightColor = c("#006400")
  private val leftColor = c("#FFA500")
  private val bottomColor = c("#800080")
```

Finally, declare an `init()` block to apply styling to the classes. Define your selection and provide a block that manipulates its various properties. (For compound selections, call the `s()` function, which is an alias for the `select()` function). Set `rotate` to 10 degrees, define the `borderColor` using the four colors and the `box()` function, make the font family "Comic Sans MS", and increase the `fontSize` to 20 pixels. Note that there are extension properties for `Number` types to quickly yield the value in that unit, such as `10.deg` for 10 degrees and `20.px` for 20 pixels.

```
import javafx.scene.paint.Color
import tornadofx.*

class MyStyle: Stylesheet() {

    companion object {
        val tackyButton by cssclass()

        private val topColor = Color.RED
        private val rightColor = Color.DARKGREEN
        private val leftColor = Color.ORANGE
        private val bottomColor = Color.PURPLE
    }

    init {
        tackyButton {
            rotate = 10.deg
            borderColor += box(topColor,rightColor,bottomColor,leftColor)
            fontFamily = "Comic Sans MS"
            fontSize = 20.px
        }
    }
}
```

Now you can apply the `tackyButton` style to buttons, labels, and other controls that support these properties. While this styling can work with other controls like labels, we are going to target buttons in this example.

First, load the `MyStyle` stylesheet into your application by including it as contructor parameter.

```
class MyApp: App(MyView::class, MyStyle::class) {
    init {
        reloadStylesheetsOnFocus()
    }
}
```

> The `reloadStylesheetsOnFocus()` function call will instruct TornadoFX to reload the Stylesheets every time the `Stage` gets focus. You can also pass the `--live-stylesheets` argument to the application to accomplish this.

Important: For the reload to work, you must be running the JVM in debug mode and you must instruct your IDE to recompile before you switch back to your app. Without these steps, nothing will happen. This also applies to `reloadViewsOnFocus()` which is similar, but reloads the whole view instead of just the stylesheet. This way, you can evolve your UI very rapidly in a "code change, compile, refresh" manner.

You can apply styles directly to a control by calling its `addClass()` function. Provide the `MyStyle.tackyButton` style to two buttons (Figure 6.2).

```
class MyView: View() {
    override val root = vbox {
        button("Press Me") {
            addClass(MyStyle.tackyButton)
        }
        button("Press Me Too") {
            addClass(MyStyle.tackyButton)
        }
    }
}
```

Figure 6.2

> Intellij IDEA can perform a quickfix to import member variables, allowing `addClass(MyStyle.tackyButton)` to be shortened to `addClass(tackyButton)` if you prefer.

You can use `removeClass()` to remove the specified style as well.

Targeting Styles to a Type

One of the benefits of using pure Kotlin is you can tightly manipulate UI control behavior and conditions using Kotlin code. For example, you can apply the style to any `Button` by iterating through a control's `children`, filtering for only children that are Buttons, and applying the `addClass()` to them.

```
class MyView: View() {
    override val root = vbox {
        button("Press Me")
        button("Press Me Too")

        children.asSequence()
                .filter { it is Button }
                .forEach { it.addClass(MyStyle.tackyButton) }
    }
}
```

Infact, manipulating classes on several nodes at once is so common that TornadoFX provides a shortcut for it:

```
children.filter { it is Button }.addClass(MyStyle.tackyButton) }
```

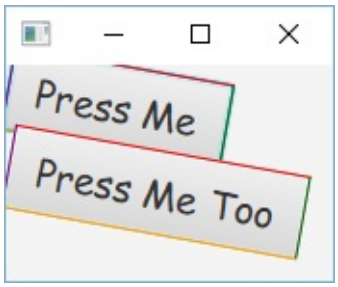
You can also target all `Button` instances in your application by selecting and modifying the `button` in the `Stylesheet`. This will apply the style to all Buttons.

```
import javafx.scene.paint.Color
import tornadofx.*

class MyStyle: Stylesheet() {

    companion object {
        val tackyButton by cssclass()

        private val topColor = Color.RED
        private val rightColor = Color.DARKGREEN
        private val leftColor = Color.ORANGE
        private val bottomColor = Color.PURPLE
    }

    init {
        button {
            rotate = 10.deg
            borderColor += box(topColor,rightColor,leftColor,bottomColor)
            fontFamily = "Comic Sans MS"
            fontSize = 20.px
        }
    }
}
```

```
import javafx.scene.layout.VBox
import tornadofx.*

class MyApp: App(MyView::class, MyStyle::class) {
    init {
        reloadStylesheetsOnFocus()
    }
}
```

```
class MyView: View() {
    override val root = vbox {
        button("Press Me")
        button("Press Me Too")
    }
}
```

Figure 6.3

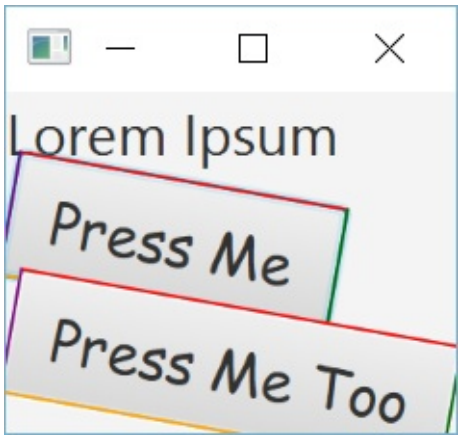


Note also you can select multiple classes and control types to mix-and-match styles. For example, you can set the font size of labels and buttons to 20 pixels, and create tacky borders and fonts only for buttons (Figure 6.4).

```
class MyStyle: Stylesheet() {

    companion object {

        private val topColor = Color.RED
        private val rightColor = Color.DARKGREEN
        private val leftColor = Color.ORANGE
        private val bottomColor = Color.PURPLE
    }

    init {
        s(button, label) {
            fontSize = 20.px
        }
        button {
            rotate = 10.deg
            borderColor += box(topColor,rightColor,leftColor,bottomColor)
            fontFamily = "Comic Sans MS"
        }
    }
}
```

```
class MyApp: App(MyView::class, MyStyle::class) {
    init {
        reloadStylesheetsOnFocus()
    }
}

class MyView: View() {
    override val root = vbox {
        label("Lorem Ipsum")
        button("Press Me")
        button("Press Me Too")
    }
}
```

Figure 6.4

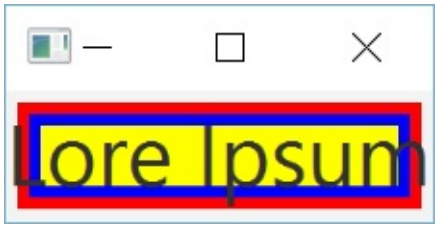


Multi-Value CSS Properties

Some CSS properties accept multiple values, and TornadoFX Stylesheets can streamline this with the `multi()` function. This allows you to specify multiple values via a `varargs` parameter and let TornadoFX take care of the rest. For instance, you can nest multiple background colors and insets into a control (Figure 6.5).

```
label("Lore Ipsum") {
    style {
        fontSize = 30.px
        backgroundColor = multi(Color.RED, Color.BLUE, Color.YELLOW)
        backgroundInsets = multi(box(4.px), box(8.px), box(12.px))
    }
}
```

Figure 6.5



The `multi()` function should work wherever multiple values are accepted. If you want to only assign a single value to a property that accepts multiple values, you will need to use the `plusAssign()` operator to add it (Figure 6.6).

```
label("Lore Ipsum") {
    style {
        fontSize = 30.px
        backgroundColor += Color.RED
        backgroundInsets += box(4.px)
    }
}
```

Figure 6.6

Nesting Styles

Inside a selector block you can apply further styles targeting child controls.

For instance, define a CSS class called `critical`. Make it put an orange border around any control it is applied to, and pad it by 5 pixels.

```
class MyStyle: Stylesheet() {

    companion object {
        val critical by cssclass()
    }

    init {
        critical {
            borderColor += box(Color.ORANGE)
            padding = box(5.px)
        }
    }
}
```

But suppose when we applied `critical` to any control, such as an `HBox`, we want it to add additional stylings to buttons inside that control. Nesting another selection will do the trick.

```
class MyStyle: Stylesheet() {
    companion object {
        val critical by cssclass()
    }
    init {
        critical {
            borderColor += box(Color.ORANGE)
            padding = box(5.px)
            button {
                backgroundColor += Color.RED
                textFill = Color.WHITE
            }
        }
    }
}
```

Now when you apply `critical` to say, an `HBox`, all buttons inside that `HBox` will get that defined style for `button` (Figure 6.7)

```
class MyApp: App(MyView::class, MyStyle::class) {
    init {
        reloadStylesheetsOnFocus()
    }
}

class MyView: View() {
    override val root = hbox {
        addClass(MyStyle.critical)
        button("Warning!")
        button("Danger!")
    }
}
```

Figure 6.7

There is one critical thing to not confuse here. *The orange border is only applied to the HBox* since the `critical` class was applied to it. The buttons do not get an orange border because they are children to the `HBox`. While their style is defined by `critical`, they do not inherit the styles of their parent, only those defined for `button`.

If you want the buttons to get an orange border too, you need to apply the `critical` class directly to them. You will want to use the `and()` to apply specific styles to buttons that are also declared as `critical`.

```
class MyStyle: Stylesheet() {

    companion object {
        val critical by cssclass()
    }

    init {
        critical {

            borderColor += box(Color.ORANGE)
            padding = box(5.px)

            and(button) {
                backgroundColor += Color.RED
                textFill = Color.WHITE
            }
        }
    }
}
```

```
class MyApp: App(MyView::class, MyStyle::class) {
    init {
        reloadStylesheetsOnFocus()
    }
}

class MyView: View() {
    override val root = hbox {
        addClass(MyStyle.critical)

        button("Warning!") {
            addClass(MyStyle.critical)
        }

        button("Danger!") {
            addClass(MyStyle.critical)
        }
    }
}
```

Figure 6.8

Now you have orange borders around the `HBox` as well as the buttons. When nesting styles, keep in mind that wrapping the selection with `and()` will cascade styles to children controls or classes.

Mixins

There are times you may want to reuse a set of stylings and apply them to several controls and selectors. This prevents you from having to redundantly define the same properties and values. For instance, if you want to create a set of styling called `redAllTheThings`, you could define it as a mixin as shown below. Then you can reuse it for a `redStyle` class, as well as a `textInput`, a `label`, and a `passwordField` with additional style modifications (Figure 6.9).

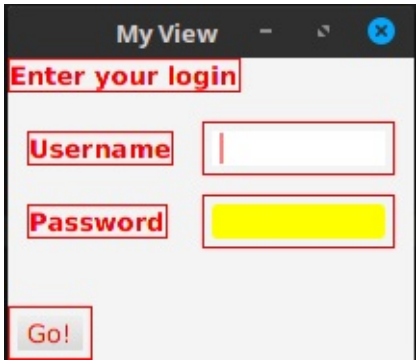
Stylesheet

```
import javafx.scene.paint.Color
import javafx.scene.text.FontWeight
import tornadofx.*

class Styles : Stylesheet() {

    companion object {
        val redStyle by cssclass().
    }

    init {
        val redAllTheThings = mixin {
            backgroundInsets += box(5.px)
            borderColor += box(Color.RED)
            textFill = Color.RED
        }

        redStyle {
            +redAllTheThings
        }

        s(textInput, label) {
            +redAllTheThings
            fontWeight = FontWeight.BOLD
        }

        passwordField {
            +redAllTheThings
            backgroundColor += Color.YELLOW
        }
    }
}
```

App and View

```
class MyApp: App(MyView::class, Styles::class)

class MyView : View("My View") {
    override val root = vbox {
        label("Enter your login")
        form {
            fieldset{
                field("Username") {
                    textfield()
                }
                field("Password") {
                    passwordfield()
                }
            }
        }
        button("Go!") {
            addClass(Styles.redStyle)
        }
    }
}
```

Figure 6.9



The stylesheet is applied to the application by adding it as a constructor parameter to the App class. This is a vararg parameter, so you can send in a comma separated list of multiple stylesheets. If you want to load stylesheets dynamically based on some condition, you can call `importStylesheet(Styles::class` from anywhere. Any UIComponent opened after the call to `importStylesheet` will get the stylesheet applied. You can also load normal text based css stylesheets with this function:

```
importStylesheet("/mystyles.css")
```

> Loading a text based css stylesheet

If you find you are repeating yourself setting the same CSS properties to the same values, you might want to consider using mixins and reusing them wherever they are needed in a `Stylesheet`.

Modifier Selections

TornadoFX also supports modifier selections by leveraging `and()` functions within a selection. The most common case this is handy is styling for "selected" and cursor "hover" contexts for a control.

If you wanted to create a UI that will make any `Button` red when it is hovered over, and any selected `Cell` in data controls such as `ListView` red, you can define a `Stylesheet` like this (Figure 6.10).

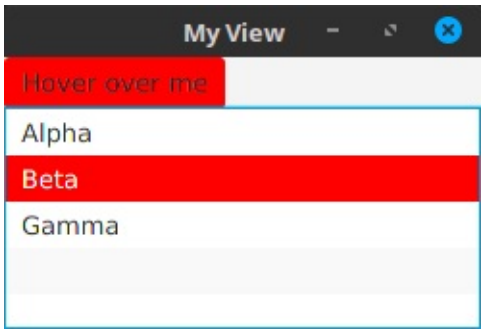
Stylesheet

```
import javafx.scene.paint.Color
import tornadofx.Stylesheet

class Styles : Stylesheet() {

    init {
        button {
            and(hover) {
                backgroundColor += Color.RED
            }
        }
        cell {
            and(selected) {
                backgroundColor += Color.RED
            }
        }
    }
}
```

App and View

```
import tornadofx.*

class MyApp: App(MyView::class, Styles::class)

class MyView : View("My View") {

    val listItems = listOf("Alpha","Beta","Gamma").observable()
and
    override val root = vbox {
        button("Hover over me")
        listview(listItems)
    }
}
```

Figure 6.10 - A cell is selected and the `Button` is being hovered over. Both are now red.



Whenever you need modifiers, use the `select()` function to make those contextual style modifications.

Control-Specific Stylesheets

If you decide to create your own controls (often by extending an existing control, like `Button`), JavaFX allows you to pair a stylesheet with it. In this situation, it is advantageous to load this `Stylesheet` only when this control is loaded. For instance, if you have a `DangerButton` class that extends `Button`, you might consider creating a `Stylesheet` specifically for that `DangerButton`. To allow JavaFX to load it, you need to override the `getUserAgentStyleSheet()` function as shown below. This will convert your type-safe `Stylesheet` into plain text CSS that JavaFX natively understands.

```
class DangerButton : Button("Danger!") {
    init {
        addClass(DangerButtonStyles.dangerButton)
    }
    override fun getUserAgentStylesheet() = DangerButtonStyles().base64URL.toExternalF
orm()
}

class DangerButtonStyles : Stylesheet() {
    companion object {
        val dangerButton by cssclass()
    }

    init {
        dangerButton {
            backgroundInsets += box(0.px)
            fontWeight = FontWeight.BOLD
            fontSize = 20.px
            padding = box(10.px)
        }
    }
}
```

The `DangerButtonStyles().base64URL.toExternalForm()` expression creates an instance of the `DangerButtonStyles`, and turns it into a URL containing the entire stylesheet that JavaFX can consume.

Conclusion

TornadoFX does a great job executing a brilliant concept to make CSS type-safe, and it further demonstrates the power of Kotlin DSL's. Configuration through static text files is slow to express with, but type-safe CSS makes it fluent and quick especially with IDE auto-completion. Even if you are pragmatic about UI's and feel styling is superfluous, there will be times you need to leverage conditional formatting and highlighting so rules "pop out" in a UI. At minimum, get comfortable using the inline `style { }` block so you can quickly access styling properties that cannot be accessed any other way (such as `TextWeight`).

Layouts and Menus

Complex UI's require many controls. It is likely these controls need to be grouped, positioned, and sized with set policies. Fortunately TornadoFX streamlines many layouts that come with JavaFX, as well as features its own proprietary `Form` layout.

TornadoFX also has type-safe builders to create menus in a highly structured, declarative way. Menus can be especially cumbersome to build using conventional JavaFX code, and Kotlin really shines in this department.

Builders for Layouts

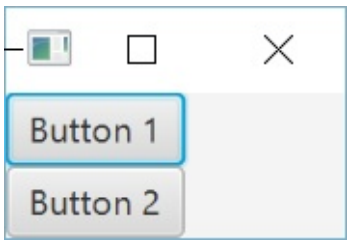
Layouts group controls and set policies about their sizing and positioning behavior. Technically, layouts themselves are controls so therefore you can nest layouts inside layouts. This is critical for building complex UI's, and TornadoFX makes maintenance of UI code easier by visibly showing the nested relationships.

VBox

A `VBox` stacks controls vertically in the order they are declared inside its block (Figure 7.1).

```
vbox {
    button("Button 1").setOnAction {
        println("Button 1 Pressed")
    }
    button("Button 2").setOnAction {
        println("Button 2 Pressed")
    }
}
```

Figure 7.1



You can also call `vboxConstraints()` within a child's block to change the margin and vertical growing behaviors of the `VBox`.

```
vbox {
    button("Button 1") {
         vboxConstraints {
            marginBottom = 20.0
            vGrow = Priority.ALWAYS
          }
    }
    button("Button 2")
}
```

You can use a shorthand extension property for `vGrow` without calling `vboxConstraints()`.

```
vbox {
    button("Button 1") {
           vGrow = Priority.ALWAYS
    }
    button("Button 2")
}
```

HBox

`HBox` behaves almost identically to `VBox`, but it stacks all controls horizontally left-to-right in the order declared in its block.

```
hbox {
    button("Button 1").setOnAction {
        println("Button 1 Pressed")
    }
    button("Button 2").setOnAction {
        println("Button 2 Pressed")
    }
}
```

Figure 7.2

You can also call `hboxconstraints()` within the a child's block to change the margin and horizontal growing behaviors of the `HBox`.

```
hbox {
    button("Button 1") {
        hboxConstraints {
                marginRight = 20.0
          hGrow = Priority.ALWAYS
      }
    }
    button("Button 2")
}
```

You can use a shorthand extension property for `hGrow` without calling `hboxConstraints()`.

```
hbox {
    button("Button 1") {
          hGrow = Priority.ALWAYS
    }
  button("Button 2")
}
```

FlowPane

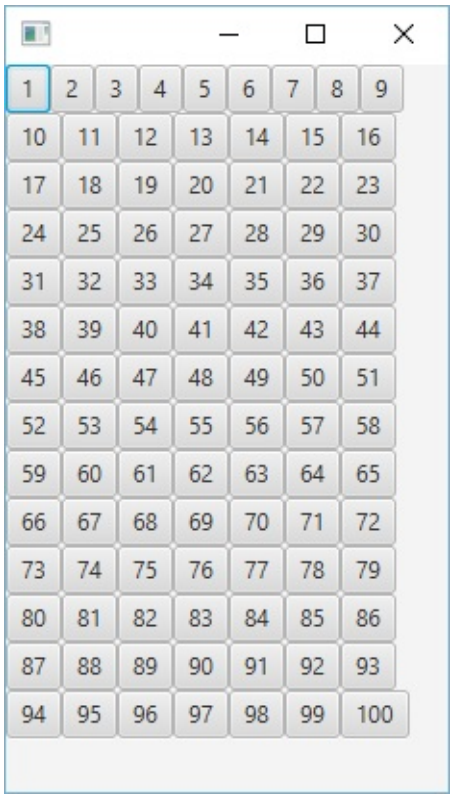
The `FlowPane` lays out controls left-to-right and wraps to the next line on the boundary. For example, say you added 100 buttons to a `FlowPane` (Figure 7.3). You will notice it simply lays out buttons from left-to-right, and when it runs out of room it moves to the "next line".

```
flowpane {
   for (i in 1..100) {
        button(i.toString()) {
            setOnAction { println("You pressed button $i") }
        }
   }
}
```

Figure 7.3

Notice also when you resize the window, the `FlowLayout` will re-wrap the buttons so they all can fit (Figure 7.4)

Figure 7.4



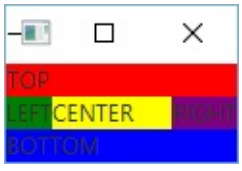
The `FlowLayout` is not used often because it is often simplistic for handling a large number of controls, but it comes in handy for certain situations and can be used inside other layouts.

BorderPane

The `BorderPane` is a highly useful layout that divides controls into 5 regions: `top`, `left`, `bottom`, `right`, and `center`. Many UI's can easily be built using two or more of these regions to hold controls (Figure 7.5).

```
borderpane {
    top = label("TOP") {
        useMaxWidth = true
        style {
            backgroundColor = Color.RED
        }
    }

    bottom = label("BOTTOM") {
        useMaxWidth = true
        style {
            backgroundColor = Color.BLUE
        }
    }

    left = label("LEFT") {
        useMaxWidth = true
        style {
            backgroundColor = Color.GREEN
        }
    }

    right = label("RIGHT") {
        useMaxWidth = true
        style {
            backgroundColor = Color.PURPLE
        }
    }

    center = label("CENTER") {
        useMaxWidth = true
        style {
            backgroundColor = Color.YELLOW
        }
    }
}
```

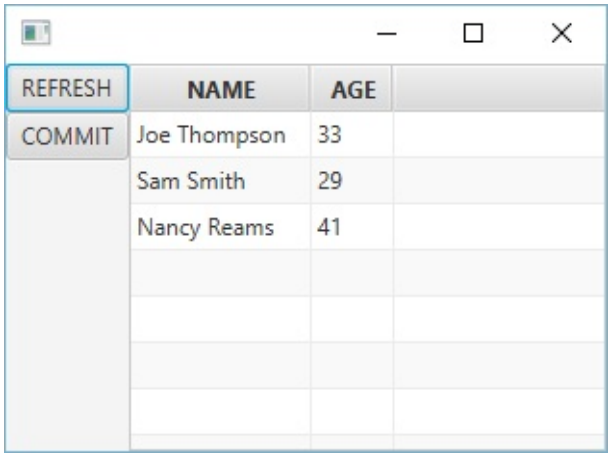
FIGURE 7.5



You will notice that the `top` and `bottom` regions take up the entire horizontal space, while `left`, `center`, and `right` must share the available horizontal space. But `center` is entitled to any extra available space (vertically and horizontally), making it ideal to hold large controls like `TableView`. For instance, you may vertically stack some buttons in the `left` region and put a `TableView` in the `center` region (Figure 7.6).

```
borderpane {
    left = vbox {
        button("REFRESH")
        button("COMMIT")
    }

    center  = tableview<Person> {
        items = listOf(
                Person("Joe Thompson", 33),
                Person("Sam Smith", 29),
                Person("Nancy Reams", 41)
        ).observable()

        column("NAME",Person::name)
        column("AGE",Person::age)
    }
}
```

Figure 7.6



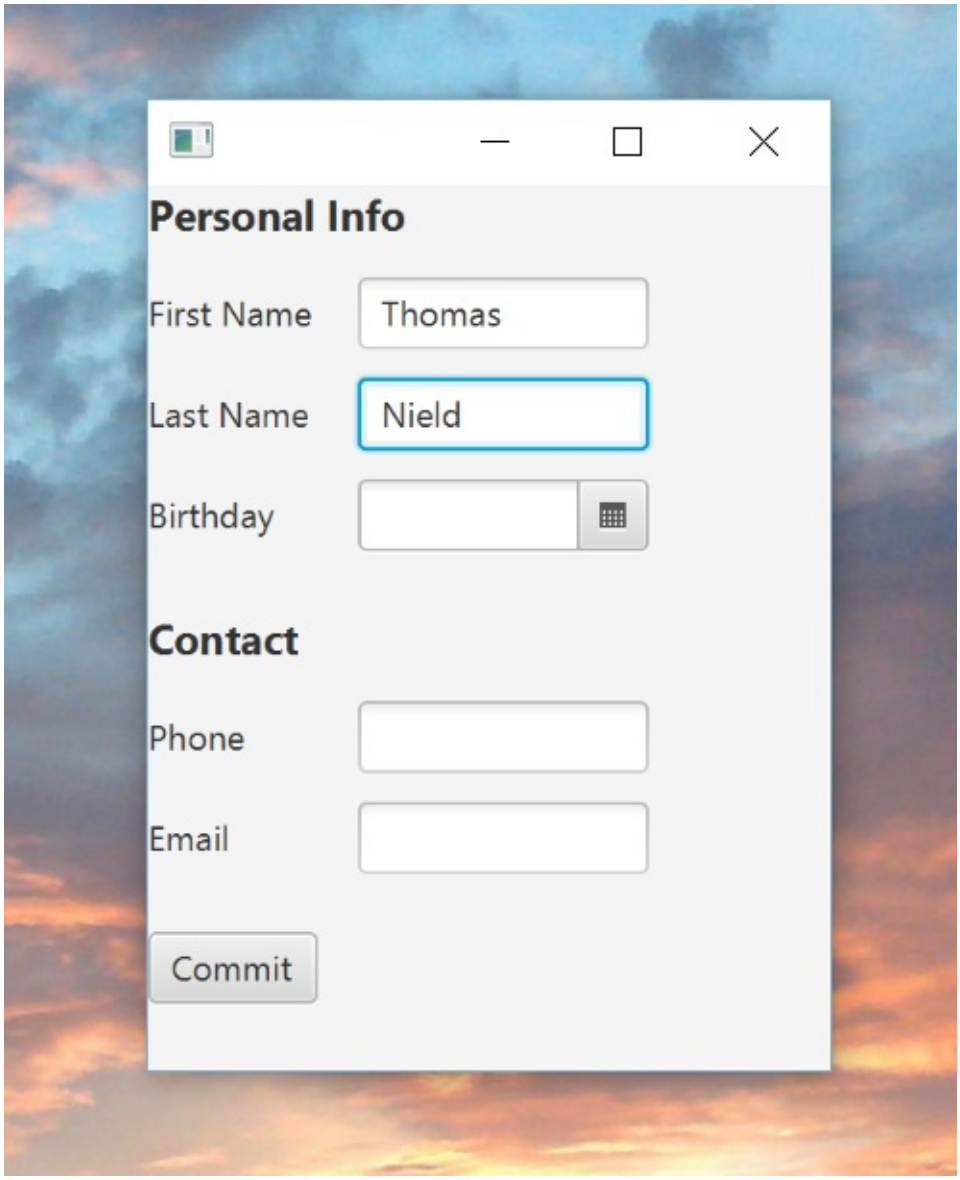
`BorderPane` is a layout you will likely want to use often because it simplifies many complex UI's. The `top` region is commonly used to hold a `MenuBar` and the `bottom` region often holds a status bar of some kind. You have already seen `center` hold the focal control such as a `TableView`, and `left` and `right` hold side panels with any peripheral controls (like Buttons or Toolbars) not appropriate for the `MenuBar`. We will learn about Menus later in this section.

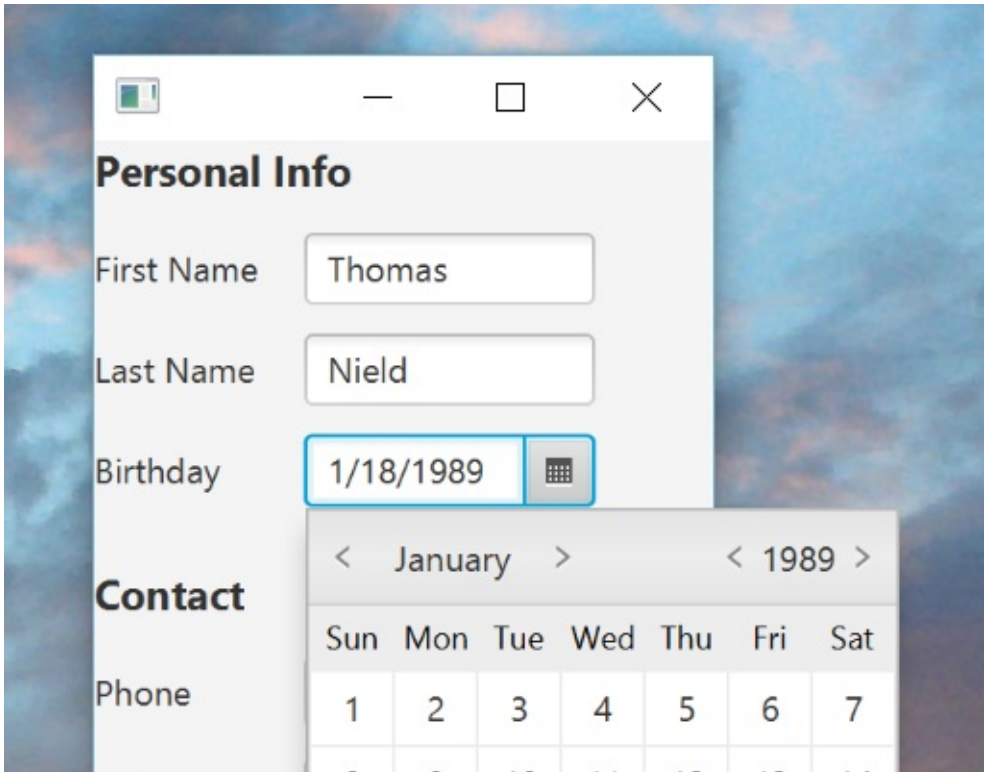
Form Builder

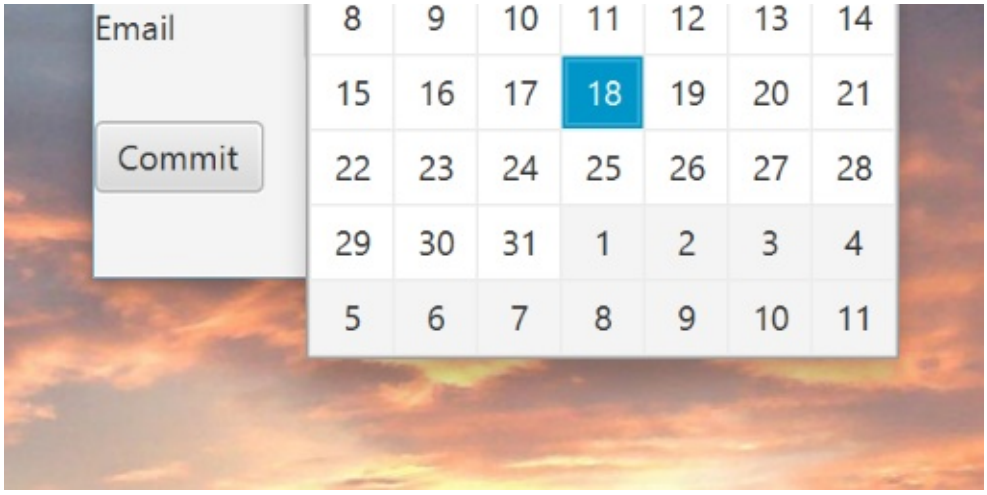
TornadoFX has a helpful `Form` control to handle a large number of user inputs. Having several input fields to take user information is common and JavaFX does not have a built-in solution to streamline this. To remedy this, TornadoFX has a builder to declare a `Form` with any number of fields (Figure 7.7).

```
form {
    fieldset("Personal Info") {
        field("First Name") {
            textfield()
        }
        field("Last Name") {
            textfield()
        }
        field("Birthday") {
            datepicker()
        }
    }
    fieldset("Contact") {
        field("Phone") {
            textfield()
        }
        field("Email") {
            textfield()
        }
    }
    button("Commit") {
        action { println("Wrote to database!")}
    }
}
```

Figure 7.7


Personal Info
First Name
Thomas
Last Name
Nield
Birthday
Contact
Phone
Email
Commit


Personal Info
First Name
Thomas
Last Name
Nield
Birthday
1/18/1989
January
1989
Sun Mon Tue Wed Thu Fri Sat
1 2 3 4 5 6 7
Contact
Phone



Awesome right? You can specify one or more controls for each of the fields, and the `Form` will render the groupings and labels for you.

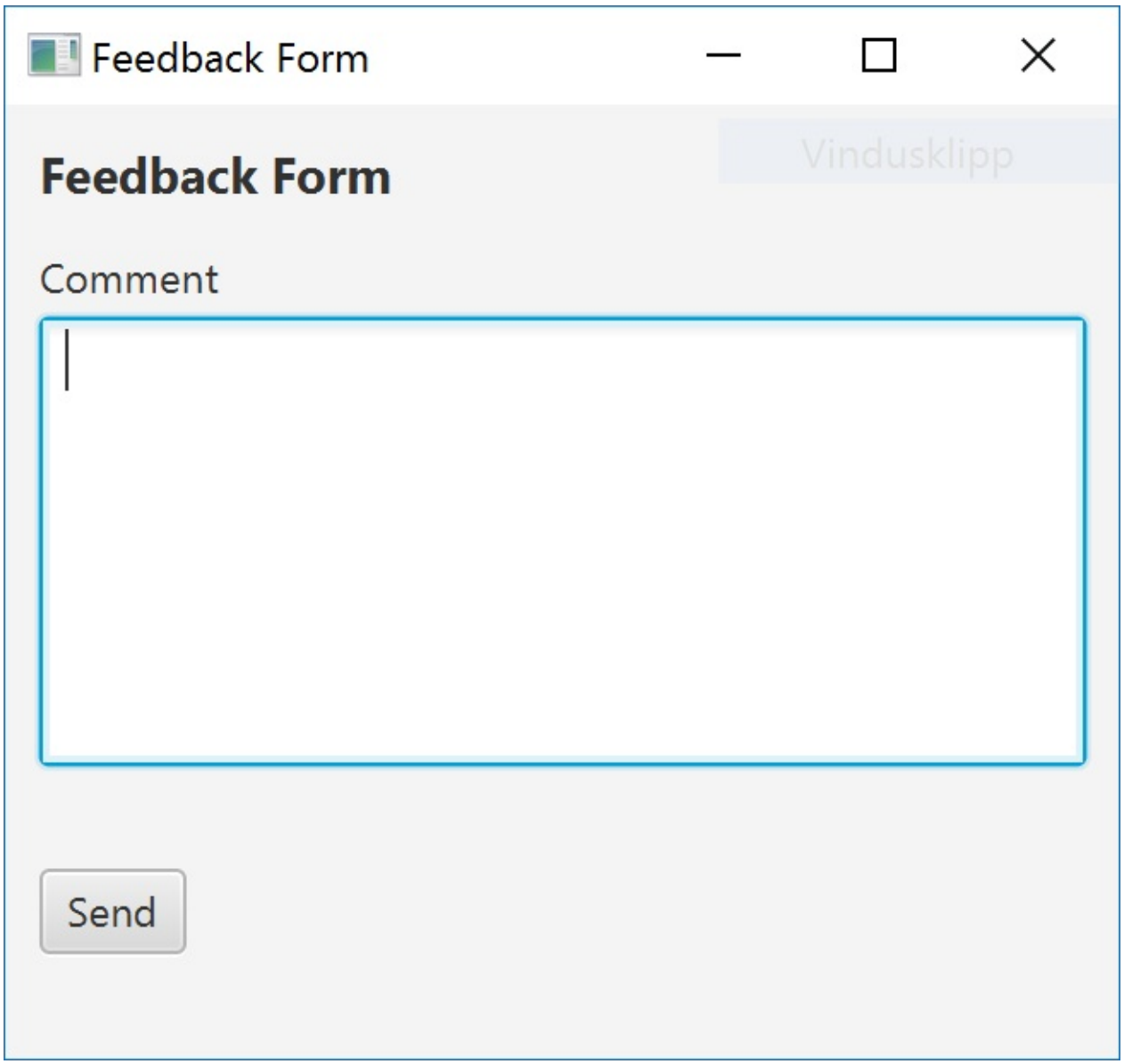
You can choose to lay out the label above the inputs as well:

```
fieldset("FieldSet", labelPosition = VERTICAL)
```

Each `field` represents a container with the label and another container for the input fields you add inside it. The input container is by default an `HBox`, meaning that multiple inputs within a single field will be laid out next to each other. You can specify the `orientation` parameter to a field to make it lay out multiple inputs below each other. Another use case for Vertical orientation is to allow an input to grow as the form expands vertically. This is handy for displaying TextAreas in Forms:

```
form {
    fieldset("Feedback Form", labelPosition = VERTICAL) {
        field("Comment", VERTICAL) {
            textarea {
                prefRowCount = 5
                vgrow = Priority.ALWAYS
            }
        }
        buttonbar {
            button("Send")
        }
    }
}
```

Figure 7.8



The example above also uses the `buttonbar` builder to create a special field with no label while retaining the label indent so the buttons line up under the inputs.

You bind each input to a model, and you can leave the rendering of the control layouts to the `Form`. For this reason you will likely want to use this over the `GridPane` if possible, which we will cover next.

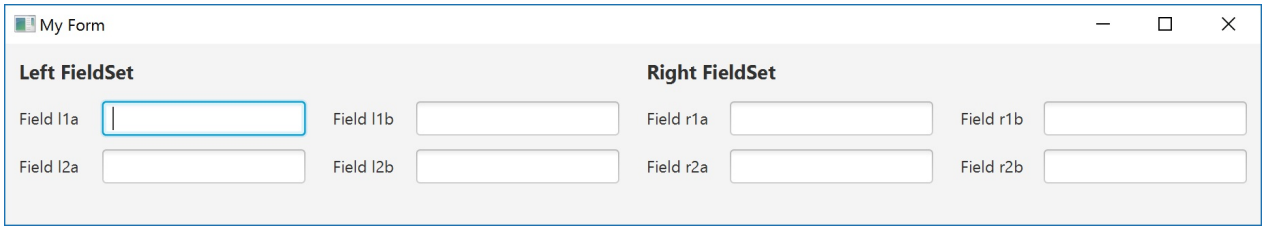
Nesting layouts inside a Form

You can wrap both fieldsets and fields with any layout container of your choosing to create complex form layouts.

```
form {
    hbox(20) {
        fieldset("Left FieldSet") {
            hbox(20) {
                vbox {
                    field("Field l1a") { textfield() }
                    field("Field l2a") { textfield() }
                }
                vbox {
                    field("Field l1b") { textfield() }
                    field("Field l2b") { textfield() }
                }
            }
        }
        fieldset("Right FieldSet") {
            hbox(20) {
                vbox {
                    field("Field r1a") { textfield() }
                    field("Field r2a") { textfield() }
                }
                vbox {
                    field("Field r1b") { textfield() }
                    field("Field r2b") { textfield() }
                }
            }
        }
    }
}
```

The HBoxes are configured with a spacing of 20 pixels, using the parameter for the `hbox` builder. It can also be specified as `hbox(spacing = 20)` for clarity.

Figure 7.9



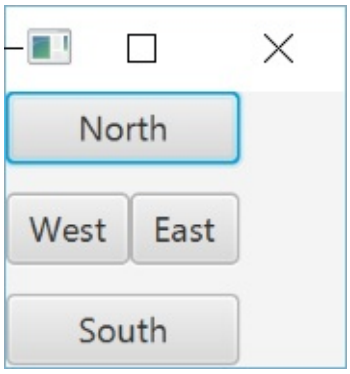
GridPane

If you want to micromanage the layout of your controls, the `GridPane` will give you plenty of that. Of course it requires more configuration and code boilerplate. Before proceeding to use a `GridPane`, you might want to consider using `Form` or other layouts that abstract layout configuration for you.

One way to use `GridPane` is to declare the contents of each `row`. For any given `Node` you can call its `gridpaneConstraints` to configure various `GridPane` behaviors for that `Node`, such as `margin` and `columnSpan` (Figure 7.10)

```
gridpane {
    row {
        button("North") {
            useMaxWidth = true
            gridpaneConstraints {
                marginBottom = 10.0
                columnSpan = 2
            }
        }
    }
    row {
        button("West")
        button("East")
    }
    row {
        button("South") {
            useMaxWidth = true
            gridpaneConstraints {
                marginTop = 10.0
                columnSpan = 2
            }
        }
    }
}
```

Figure 7.11



Notice how there is a margin of `10.0` between each row, which was declared for the `marginBottom` and `marginTop` of the "North" and "South" buttons respectively inside their `gridpaneConstraints`.

Alternatively, you can explicitly specify the column/row index positions for each `Node` rather than declaring each `row` of controls. This will accomplish the exact layout we built previously, but with column/row index specifications instead. It is a bit more verbose, but it

gives you more explicit control over the positions of controls.

```
gridpane {
     button("North") {
          useMaxWidth = true
          gridpaneConstraints {
              columnRowIndex(0,0)
              marginBottom = 10.0
              columnSpan = 2
          }
     }
    button("West").gridpaneConstraints {
        columnRowIndex(0,1)
    }
    button("East").gridpaneConstraints {
        columnRowIndex(1,1)
    }

    button("South") {
        useMaxWidth = true
        gridpaneConstraints {
            columnRowIndex(0,2)
            marginTop = 10.0
            columnSpan = 2
        }
    }
}
```

These are all the `gridpaneConstraints` attributes you can modify on a given `Node`. Some are expressed as simple properties that can be assigned while others are assignable through functions.

Attribute	Description
columnIndex: Int	The column index for the given control
rowIndex: Int	The row index for the given control
columnRowIndex(columnIndex: Int, rowIndex: Int)	Specifes the row and column index
columnSpan: Int	The number of columns the control occupies
rowSpan: Int	The number of rows the control occupies
hGrow: Priority	The horizonal grow priority
vGrow: Priority	The vertical grow priority
vhGrow: Priority	Specifies the same priority for `vGrow` and `hGrow`
fillHeight: Boolean	Sets whether the `Node` fills the height of its area
fillWidth: Boolean	Sets whether the `Node` filles the width of its area
fillHeightWidth: Boolean	Sets whether the `Node` fills its area for both height and width
hAlignment: HPos	The horizonal alignment policy
vAlignment: VPos	The vertical alignment policy
margin: Int	The margin for all four sides of the `Node`
marginBottom: Int	The margin for the bottom side of the `Node`
marginTop: Int	The margin for the top side of the `Node`
marginLeft: Int	The left margin for the left side of the `Node`
marginRight: Int	The right margin for the right side of the `Node`
marginLeftRight: Int	The right and left margins for the `Node`
marginTopBottom: Int	The top and bottom marins for a `Node`

Additionally, if you need to configure `ColumnConstraints`, you can call `gridpaneColumnConstraints` on any child `Node`, or `constraintsForColumn(columnIndex)` on the `GridPane` itself.

```
gridpane {
    row {
        button("Left") {
            gridpaneColumnConstraints {
                percentWidth = 25.0
            }
        }

        button("Middle")
        button("Right")
    }
    constraintsForColumn(1).percentWidth = 50.0
}
```

StackPane

A `StackPane` is a layout you will use less often. For each control you add, it will literally stack them on top of each other not like a `VBox`, but literally overlay them.

For instance, you can create a "BOTTOM" `Button` and put a "TOP" `Button` on top of it. The order you declare controls will add them from bottom-to-top in that same order (Figure 7.10).

```
class MyView: View() {

    override val root =  stackpane {
        button("BOTTOM") {
           useMaxHeight = true
           useMaxWidth = true
           style {
               backgroundColor += Color.AQUAMARINE
               fontSize = 40.0.px
           }
        }

        button("TOP") {
            style {
                backgroundColor += Color.WHITE
            }
        }
    }
}
```

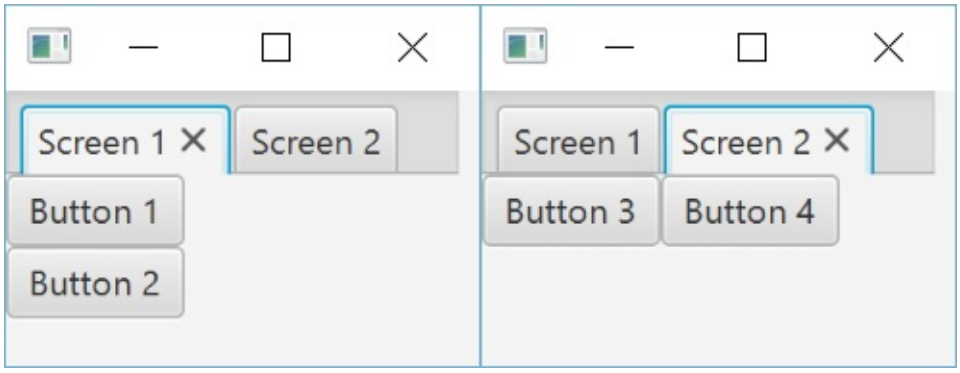
Figure 7.11

TabPane

A `TabPane` creates a UI with different screens separated by "tabs". This allows switching between different screens quickly and easily by clicking on the corresponding tab (Figure 7.11). You can declare a `tabpane()` and then declare as many `tab()` instances as you need. For each `tab()` function pass in the name of the `Tab` and the parent `Node` control to populate it.

```
tabpane {
    tab("Screen 1", VBox()) {
        button("Button 1")
        button("Button 2")
    }
    tab("Screen 2", HBox()) {
        button("Button 3")
        button("Button 4")
    }
}
```

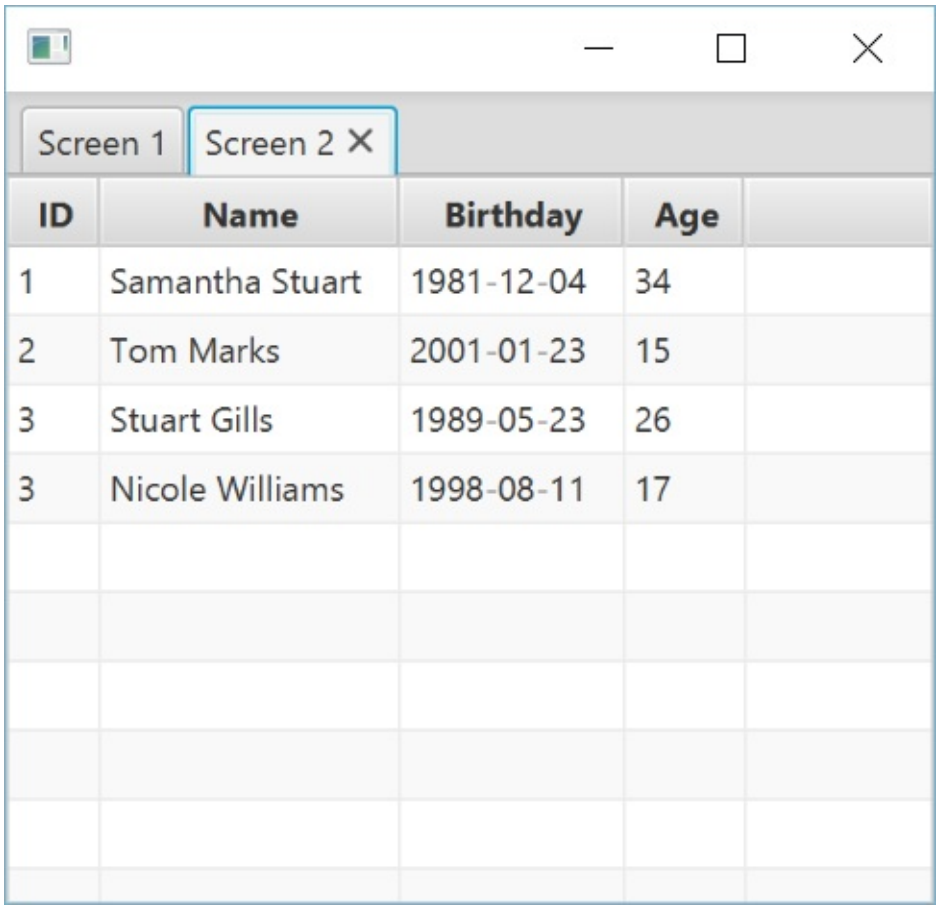
Figure 7.12



`TabPane` is an effective tool to separate screens and organize a high number of controls. The syntax is somewhat succinct enough to declare complex controls like `TableView` right inside the `tab()` block (Figure 7.13).

```
tabpane {
  tab("Screen 1", VBox()) {
      button("Button 1")
      button("Button 2")
  }
  tab("Screen 2", HBox()) {
      tableview<Person> {

          items = listOf(
              Person(1,"Samantha Stuart",LocalDate.of(1981,12,4)),
              Person(2,"Tom Marks",LocalDate.of(2001,1,23)),
              Person(3,"Stuart Gills",LocalDate.of(1989,5,23)),
              Person(3,"Nicole Williams",LocalDate.of(1998,8,11))
          ).observable()

          column("ID",Person::id)
          column("Name", Person::name)
          column("Birthday", Person::birthday)
          column("Age",Person::age)
      }
  }
}
```

Figure 7.13



Like many builders, the `TabPane` has several properties that can adjust the behavior of its tabs. For instance, you can call `tabClosingPolicy` to get rid of the "X" buttons on the tabs so they cannot be closed.

```
class MyView: View() {
    override val root =  tabpane {
        tabClosingPolicy = TabPane.TabClosingPolicy.UNAVAILABLE

        tab("Screen 1", VBox()) {
            button("Button 1")
            button("Button 2")
        }
        tab("Screen 2", HBox()) {
            button("Button 3")
            button("Button 4")
        }
    }
}
```

Builders for Menus

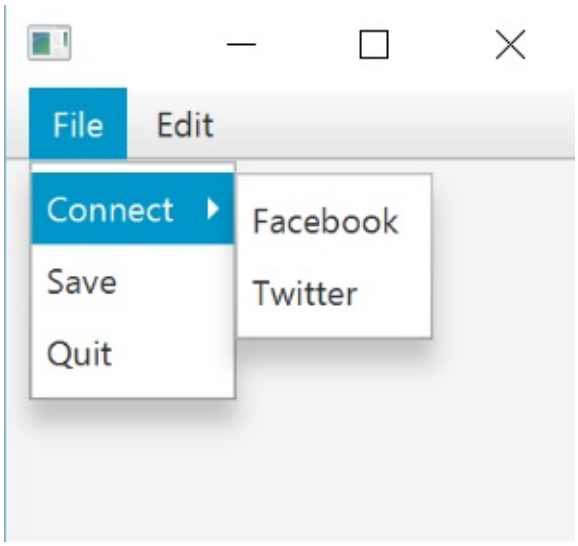
Creating menus can be cumbersome to build in a strictly object-oriented way. But using type-safe builders, Kotlin's functional constructs make it intuitive to declare nested menu hierarchies.

MenuBar, Menu, and MenuItem

It is not uncommon to use navigable menus to keep a large number of commands on a user interface organized. For instance, the `top` region of a `BorderPane` is typically where a `MenuBar` goes. There you can add menus and submenus easily (Figure 7.5).

```
menubar {
   menu("File") {
       menu("Connect") {
           item("Facebook")
           item("Twitter")
       }
       item("Save")
       item("Quit")
   }
   menu("Edit") {
       item("Copy")
       item("Paste")
   }
}
```

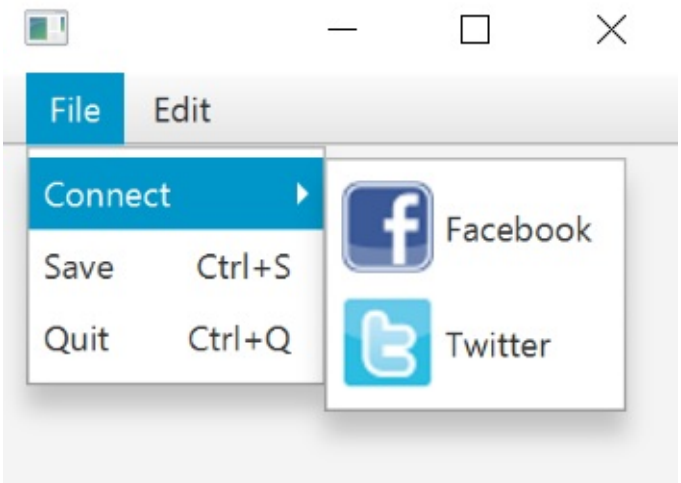
Figure 7.14



You can also optionally provide keyboard shortcuts, graphics, as well as an `action` function parameter for each `item()` to specify the action when it is selected (Figure 7.14).

```
menubar {
     menu("File") {
         menu("Connect") {
             item("Facebook", graphic = fbIcon).action { println("Connecting Facebook!"
) }
             item("Twitter", graphic = twIcon).action { println("Connecting Twitter!")
 }
         }
         item("Save","Shortcut+S").action {
             println("Saving!")
         }
         menu("Quit","Shortcut+Q").action {
             println("Quitting!")
         }
     }
     menu("Edit") {
         item("Copy","Shortcut+C").action {
             println("Copying!")
         }
         item("Paste","Shortcut+V").action {
             println("Pasting!")
         }
     }
 }
```

Figure 7.14

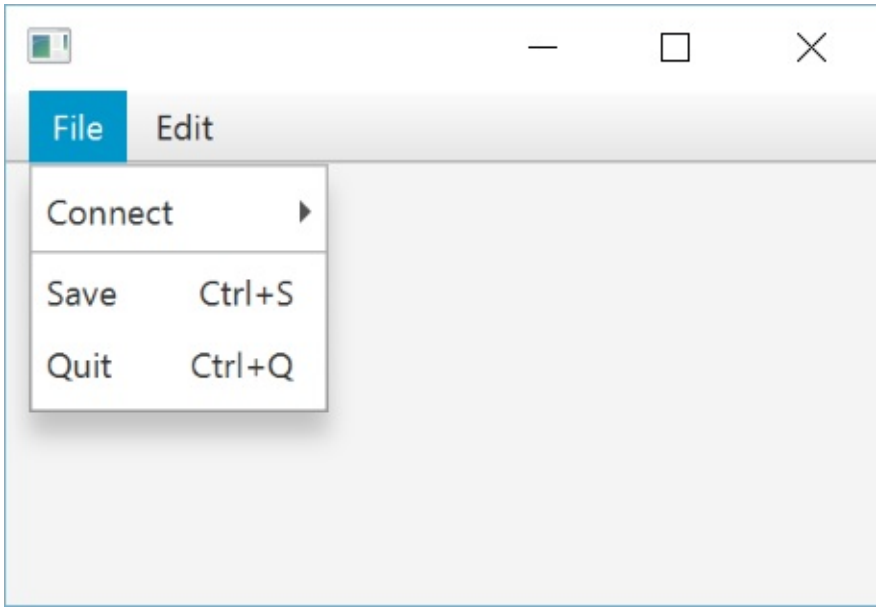


Separators

You can declare a `separator()` between two items in a `Menu` to create a divider line. This is helpful to group commands in a `Menu` and distinctly separate them (Figure 7.15).

```
menu("File") {
    menu("Connect") {
        item("Facebook")
        item("Twitter")
    }
    separator()
    item("Save","Shortcut+S") {
        println("Saving!")
    }
    item("Quit","Shortcut+Q") {
        println("Quitting!")
    }
}
```

Figure 7.15



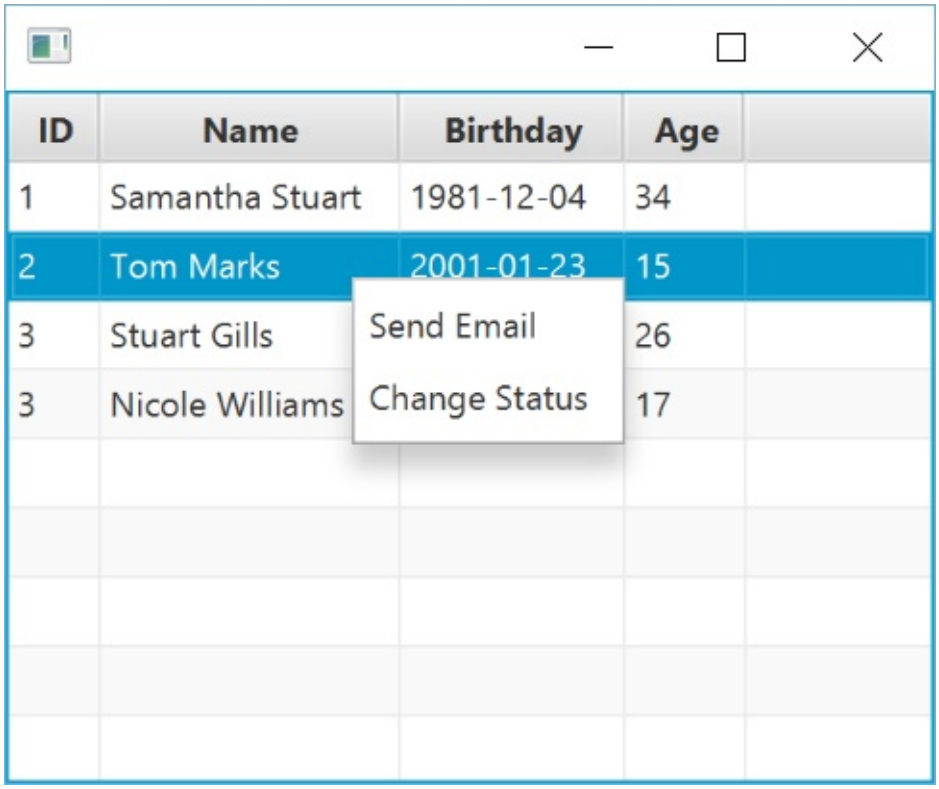
ContextMenu

Most controls in JavaFX have a `contextMenu` property where you can assign a `ContextMenu` instance. This is a `Menu` that pops up when the control is right-clicked.

A `ContextMenu` has functions to add `Menu` and `MenuItem` instances to it just like a `MenuBar`. It can be helpful to add a `ContextMenu` to a `TableView<Person>`, for example, and provide commands to be done on a table record (Figure 7.16). There is a builder called `contextmenu` that will build a `ContextMenu` and assign it to the `contextMenu` property of the control.

```
tableview(persons) {
     column("ID", Person::id)
     column("Name", Person::name)
     column("Birthday", Person::birthday)
     column("Age", Person::age)

     contextmenu {
         item("Send Email").action {
             selectedItem?.apply { println("Sending Email to $name") }
         }
         item("Change Status").action {
             selectedItem?.apply { println("Changing Status for $name") }
         }
     }
 }
```

Figure 7.16

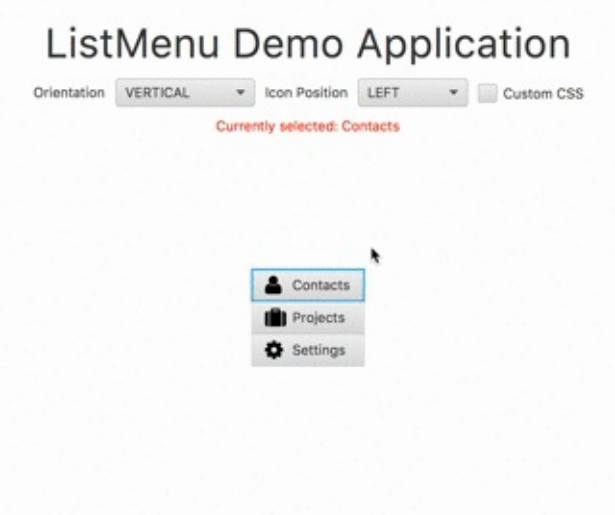


> Note there are also `RadioMenuItem` and `CheckMenuItem` variants of `MenuItem` available.

The `menuitem` builders take the action to perform when the menu is selected as the op block parameter. Unfortunately, this breaks with the other builders, where the op block operates on the element that the builder created. Therefore, the `item` builder was introduced as an alternative, where you operate on the item itself, so that you must call `setOnAction` to assign the action. The `menuitem` builder is not deprecated, as it solves the common case in a more concise way than the `item` builder.

ListMenu

TornadoFX comes with a list menu that behaves and looks more like a typical `ul/li` based HTML5 menu.



The following code example shows how to use the `ListMenu` with the builder pattern:

```
listmenu(theme = "blue") {
    item(text = "Contacts", graphic = Styles.contactsIcon()) {
        // Marks this item as active.
        activeItem = this
        whenSelected { /* Do some action */ }
    }
    item(text = "Projects", graphic = Styles.projectsIcon())
    item(text = "Settings", graphic = Styles.settingsIcon())
}
```

The following Attributes can be used to configure the `ListMenu`:

Attribute	Builder-Attribute	Type	Default	Description
orientation	yes	Orientation	VERTICAL	Configures the orientation of the `ListMenu`. Possible orientations: • VERTICAL • HORIZONTAL
iconPosition	yes	Side	LEFT	Configures the icon position of the `ListMenu`. Possible positions: • TOP • BOTTOM • LEFT • RIGHT
theme	yes	String	null	Currently supported themes `blue`, `null`. If `null` is set the default gray theme is used.
tag	yes	Any?	null	The Tag can be any `object` or `null`, it can be useful to identify the `ListMenu`
activeItem	no	ListMenuItem?	null	Represent's the current active `ListMenuItem` of the `ListMenu`. To select a `ListMenu` on creation, just assign the specific `ListItem` to this property (have a look at the contacts `ListMenuItem` in the code example above.)

Css Properties

Css-Class	Css-Property	Default	Description
`.list-menu`	-fx-graphic-fixed-size	2em	The graphic size.
`.list-menu .list-item`	-fx-cursor	hand	The cursor symbol.
`.list-menu .list-item`	-fx-padding	10	The padding for each `item`
`.list-menu .list-item`	-fx-background-color	-fx-shadow-highlight-color, -fx-outer-border, -fx-inner-border, -fx-body-color	The color of the `item`
`.list-menu .list-item`	-fx-background-insets	0 0 -0.5 0, 0, 0.5, 1.5	The insets of each `item`.
`.list-menu .list-item .label`	-fx-text-fill	-fx-text-base-color	The text color of each `item`.

Pseudo Classes

Pseudo-Class	Css-Property	Default	Description
`.list-menu .list-item:active`	-fx-background-color	-fx-focus-color, -fx-inner-border, -fx-body-color, -fx-faint-focus-color, -fx-body-color	The color will be set if the `item` is active.
`.list-menu .list-item:active`	-fx-background-insets	-0.2, 1, 2, -1.4, 2.6	Insets will be set if the `item` is active.
`.list-menu .list-item:hover`	-fx-color	-fx-hover-base	The hover color.

Have a look at the default Stylesheet for the ListMenu

Item

The `item` builder allows to create `items` for the `ListMenu` in a very convenient way. The following syntax is supported:

```
item("SomeText", graphic = SomeNode, tag = SomeObject) {
    // Marks this item as active.
    activeItem = this

    // Do some action when selected
    whenSelected { /* Action */ }
}
```

Attribute	Builder-Attribute	Type	Default	Description
text	yes	String?	null	The text which should be set for the given `item`.
tag	yes	Any?	null	The Tag can be any `object` or `null` and can be useful to identify the `ListItem`
graphic	yes	Node?	null	The `graphic` can be any `Node` and will be displayed beside the given `text`.

Function	Description
`whenSelected`	A convince function, which will be called anytime the given `ListMenuItem` is selected.

Filling the parent container

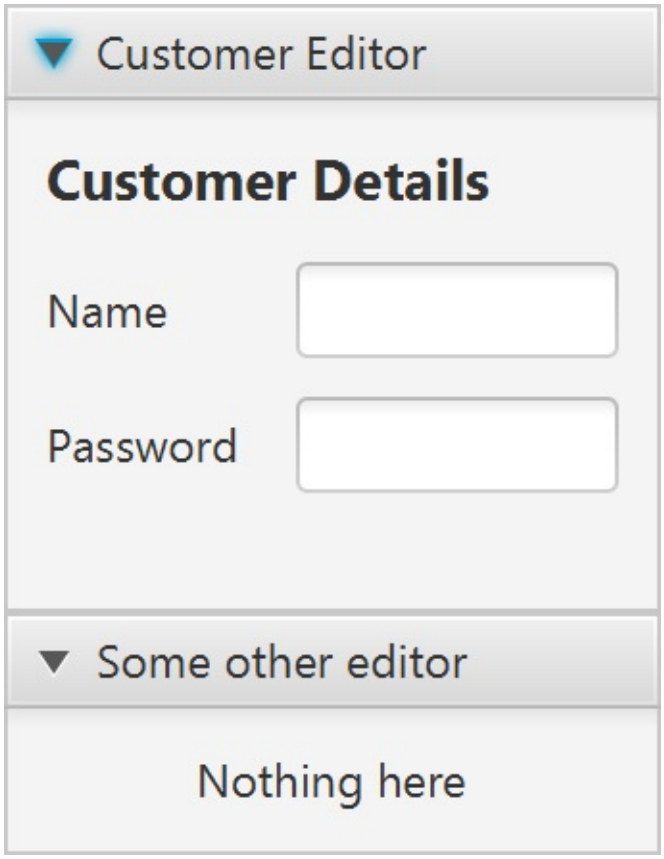
The `useMaxWidth` property can be used to fill the parent container horizontally. The `useMaxHeight` property will fill the parent container vertically. These properties actually applies to all Nodes, but is especially useful for the `ListMenu`.

SqueezeBox

JavaFX has an Accordion control that lets you group a set of `TilePanes` together to form an accordion of controls. The JavaFX Accordion only lets you open a single accordion fold at a time, and it has some other shortcomings. To solve this, TornadoFX comes with the `SqueezeBox` component that behaves and looks very similar to the Accordion, while providing some enhancements.

```
squeezebox {
    fold("Customer Editor", expanded = true) {
        form {
            fieldset("Customer Details") {
                field("Name") { textfield() }
                field("Password") { textfield() }
            }
        }
    }
    fold("Some other editor", expanded = true) {
        stackpane {
            label("Nothing here")
        }
    }
}
```

Figure 7.17



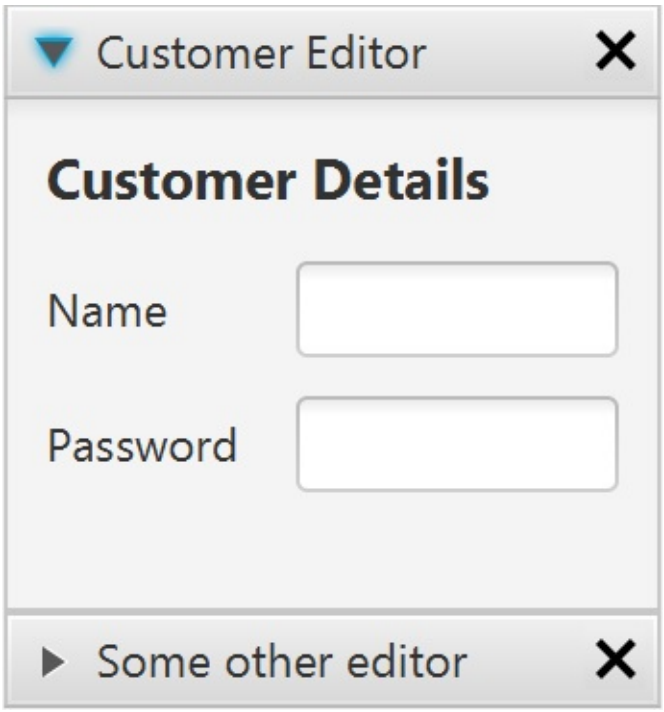
A Squeezebox showing two folds, both expanded by default

You can tell the SqueezeBox to only allow a single fold to be expanded at any given time by passing `multiselect = false` to the builder constructor.

You can optionally allow folds to be closable by clicking a cross in the right corner of the title pane for the fold. You enable the close buttons on a per fold basis by passing `closeable = true` to the `fold` builder.

```
squeezebox {
    fold("Customer Editor", expanded = true, closeable = true) {
        form {
            fieldset("Customer Details") {
                field("Name") { textfield() }
                field("Password") { textfield() }
            }
        }
    }
    fold("Some other editor", closeable = true) {
        stackpane {
            label("Nothing here")
        }
    }
}
```

Figure 7.18



> This SqueezeBox has closeable folds

The `closeable` property can of course be combined with `expanded`.

Another important difference between the SqueezeBox and the Accordion is the way it distributes overflowing space. The Accordion will extend vertically to fill its parent container and push any folds below the currently opened ones all the way to the bottom. This creates an unnatural looking view if the parent container is very large. The squeezebox probably does what you want by default in this regard, but you can add `fillHeight = true` to get a similar look as the Accordion.I

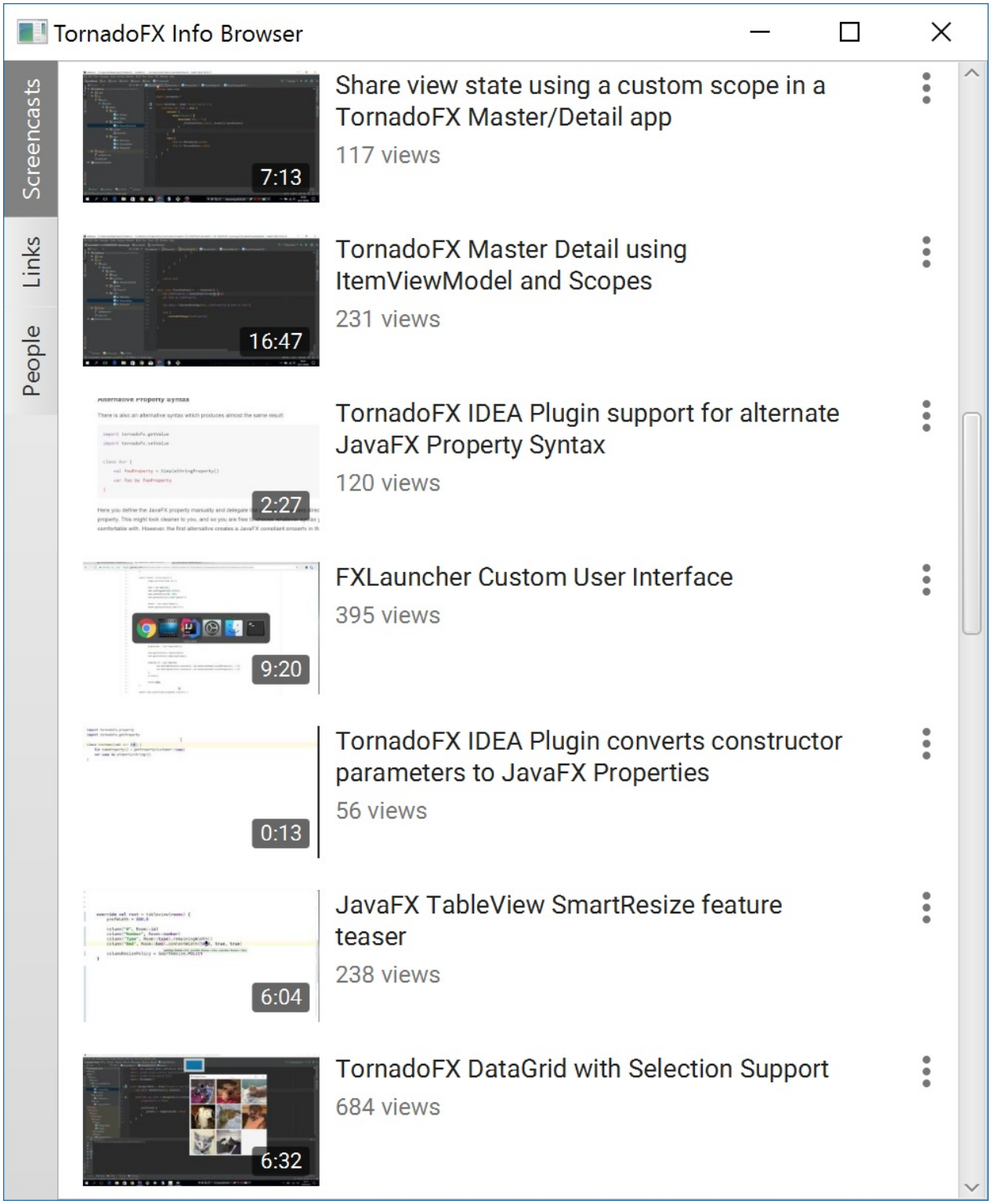
You can style the SqueezeBox like you style a TitlePane. The close button has a css class called `close-button` and the container has a css class called `squeeze-box`.

Drawer

The Drawer is a navigation component much like a TabPane, but it organizes each drawer item in a vertically or horizontally placed button bar on either side of the parent container. It resembles the tool drawers found in many popular business applications and IDEs. When an item is selected, the content for the item is displayed next to or above/below the buttons in a content area spanning the height or width of the control and the preferred width or height of the content, depending on whether it is docked in a vertical or horizontal side of the parent. In `multiselect` mode it will even let you open multiple drawer items simutaneously and have them share the space between them. They will always open in the order of the corresponding buttons.

```
class DrawerView : View("TornadoFX Info Browser") {
    override val root = drawer {
        item("Screencasts", expanded = true) {
            webview {
                prefWidth = 470.0
                engine.userAgent = iPhoneUserAgent
                engine.load(TornadoFXScreencastsURI)
            }
        }
        item("Links") {
            listview(links) {
                cellFormat { link ->
                    graphic = hyperlink(link.name) {
                        setOnAction {
                            hostServices.showDocument(link.uri)
                        }
                    }
                }
            }
        }
        item("People") {
            tableview(people) {
                column("Name", Person::name)
                column("Nick", Person::nick)
            }
        }
    }

    class Link(val name: String, val uri: String)
    class Person(val name: String, val nick: String)

    // Sample data variables left out (iPhoneUserAgent, TornadoFXScreencastsURI, people
 and links)
}
```

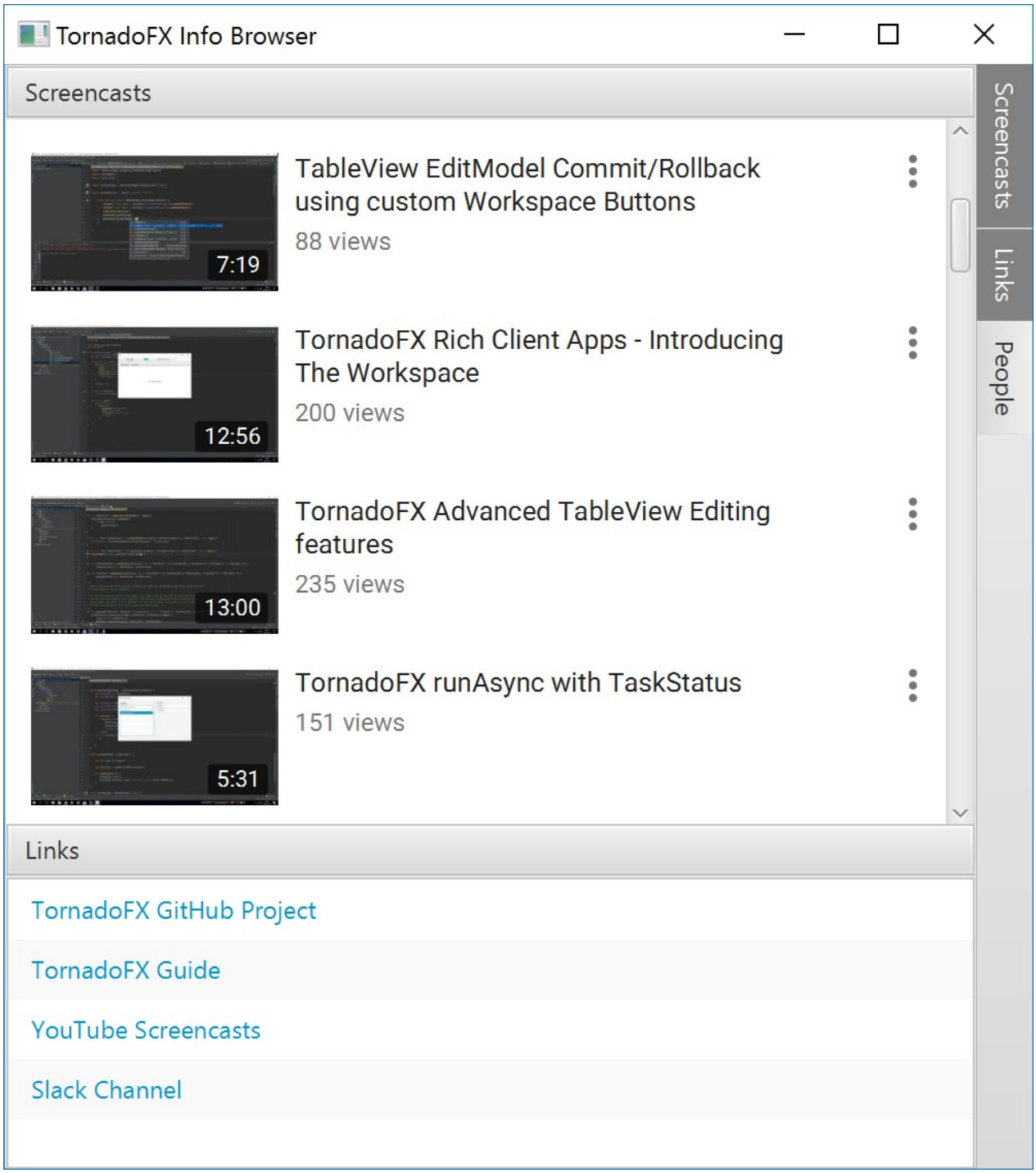
Figure 7.19



The drawer can be configured to show the buttons on the right side, and you can choose to support opening multiple drawer items simultaneously. When running in multiselect mode, a header will appear above the content, which will help to distinguish the items in the content area. You can control the header appearance with the boolean `showHeader` parameter. It will default true when multiselect is enabled and false otherwise.

```
drawer(side = Side.RIGHT, multiselect = true) {
    // Everything else is identical
}
```

Figure 7.20



> Drawer with buttons on the right side, multiselect mode and title panes

When the Drawer is added next to something, you can choose whether the content area of the Drawer should displace the nodes next to it (default) or float over it. The `floatingContent` property is by default false, causing the Drawer to displace the content

next to it.

You can control the size of the content area further using the `maxContentSize` and `fixedContentSize` properties of `Drawer`. Depending on the `dockingSide`, those properties will constrain either the width or the height of the content area.

The `Workspace` features built in support for the Drawer control. The `leftDrawer`, `rightDrawer` and `bottomDrawer` properties of any Workspace will let you dock drawer items into them. Read more about this in the Workspace chapter.

Converting observable list items and binding to layouts

TODO

Summary

By now you should have the tools to quickly create complex UI's with layouts, tabbed panes, as well as other controls to manage controls. Using these in conjunction with the data controls, you should be able to turn around UI's in a fraction of the time.

When it comes to builders, you have reached the top of the peak and have everything you need to be productive. All that is left to cover are charts and shapes, which we will cover in the next two chapters.

Charts

JavaFX comes with a handy set of charts to quickly display data visualizations. While there are more comprehensive charting libraries like JFreeChart and Orson Charts which work fine with TornadoFX, the built-in JavaFX charts satisfy a majority of visualization needs. They also have elegant animations when data is populated or changed.

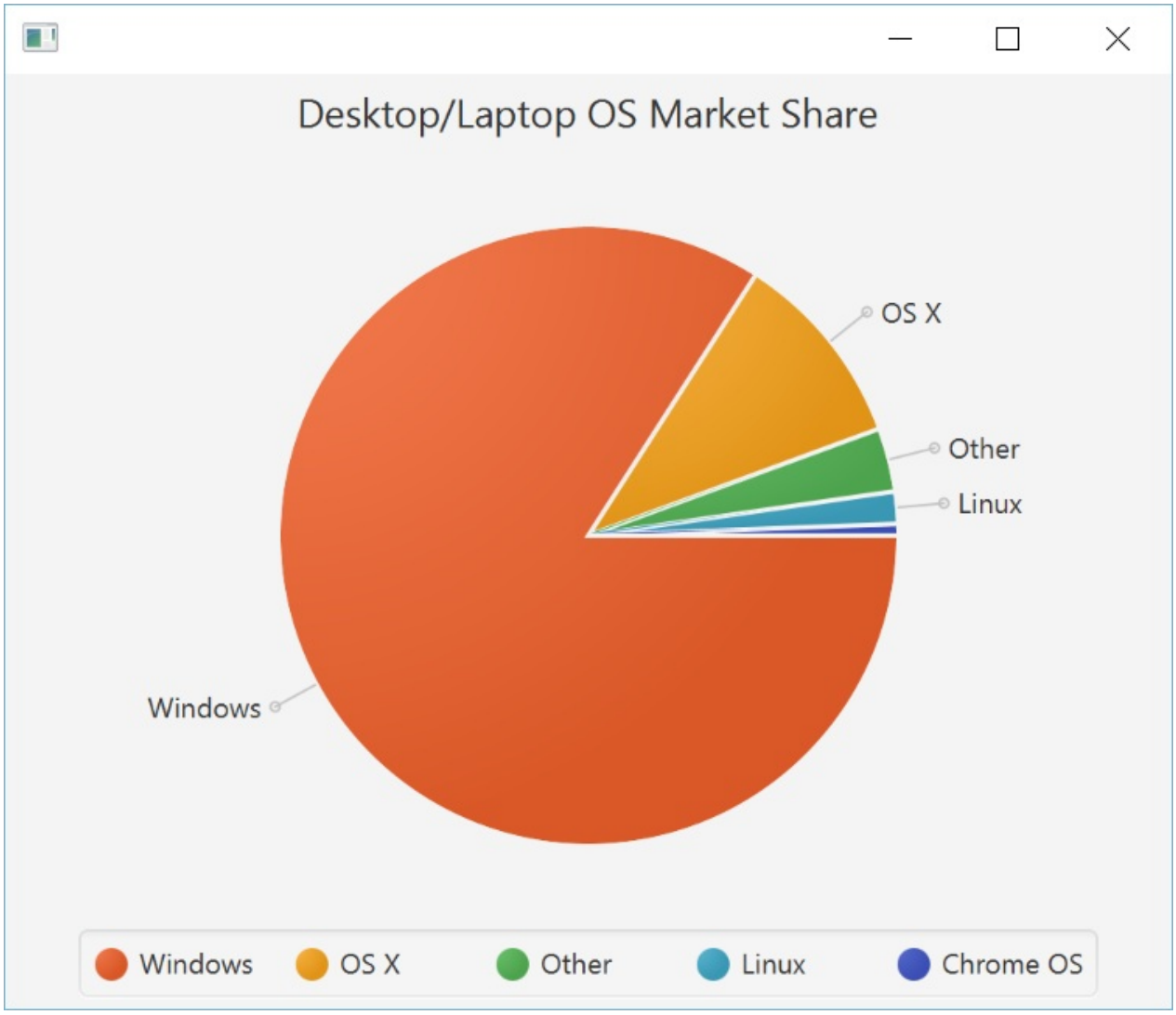
TornadoFX comes with a few builders to streamline the declaration of charts using functional constructs.

PieChart

The `PieChart` is a common visual aid to illustrate proportions of a whole. It is structurally simpler than XY charts which we will learn about later. Inside a `piechart()` builder you can call the `data()` function to pass multiple category-value pairs (Figure 8.1).

```
piechart("Desktop/Laptop OS Market Share") {
    data("Windows", 77.62)
    data("OS X", 9.52)
    data("Other", 3.06)
    data("Linux", 1.55)
    data("Chrome OS", 0.55)
}
```

Figure 8.1



Note you can also provide an explicit `ObservableList<PieChart.Data>` prepared in advance.

```
val items = listOf(
        PieChart.Data("Windows", 77.62),
        PieChart.Data("OS X", 9.52),
        PieChart.Data("Other", 3.06),
        PieChart.Data("Linux", 1.55),
        PieChart.Data("Chrome OS", 0.55)
).observable()

piechart("Desktop/Laptop OS Market Share", items)
```

The block following `piechart` can be used to modify any of the attributes of the `PieChart` just like any other control builder we covered. You can also leverage `for()` loops, Sequences, and other iterative tools within a block to add any number of data items.

```
val items = listOf(
        PieChart.Data("Windows", 77.62),
        PieChart.Data("OS X", 9.52),
        PieChart.Data("Other", 3.06),
        PieChart.Data("Linux", 1.55),
        PieChart.Data("Chrome OS", 0.55)
).observable()

piechart("Desktop/Laptop OS Market Share") {
    for (item in items) {
        data.add(item)
    }
}
```

Map-Based Data Sources

Sometimes you may want to build a chart using a `Map` as a datasource. Using the Kotlin `to` operator, you can construct a `Map` in a Kotlin-esque way and then pass it to the `data` function.

```
 val items = mapOf(
        "Windows" to 77.62,
        "OS X" to 9.52,
        "Other" to 3.06,
        "Linux" to 1.55,
        "Chrome OS" to 0.55
)

piechart("Desktop/Laptop OS Market Share") {
    data(items)
}
```

XY Based Charts

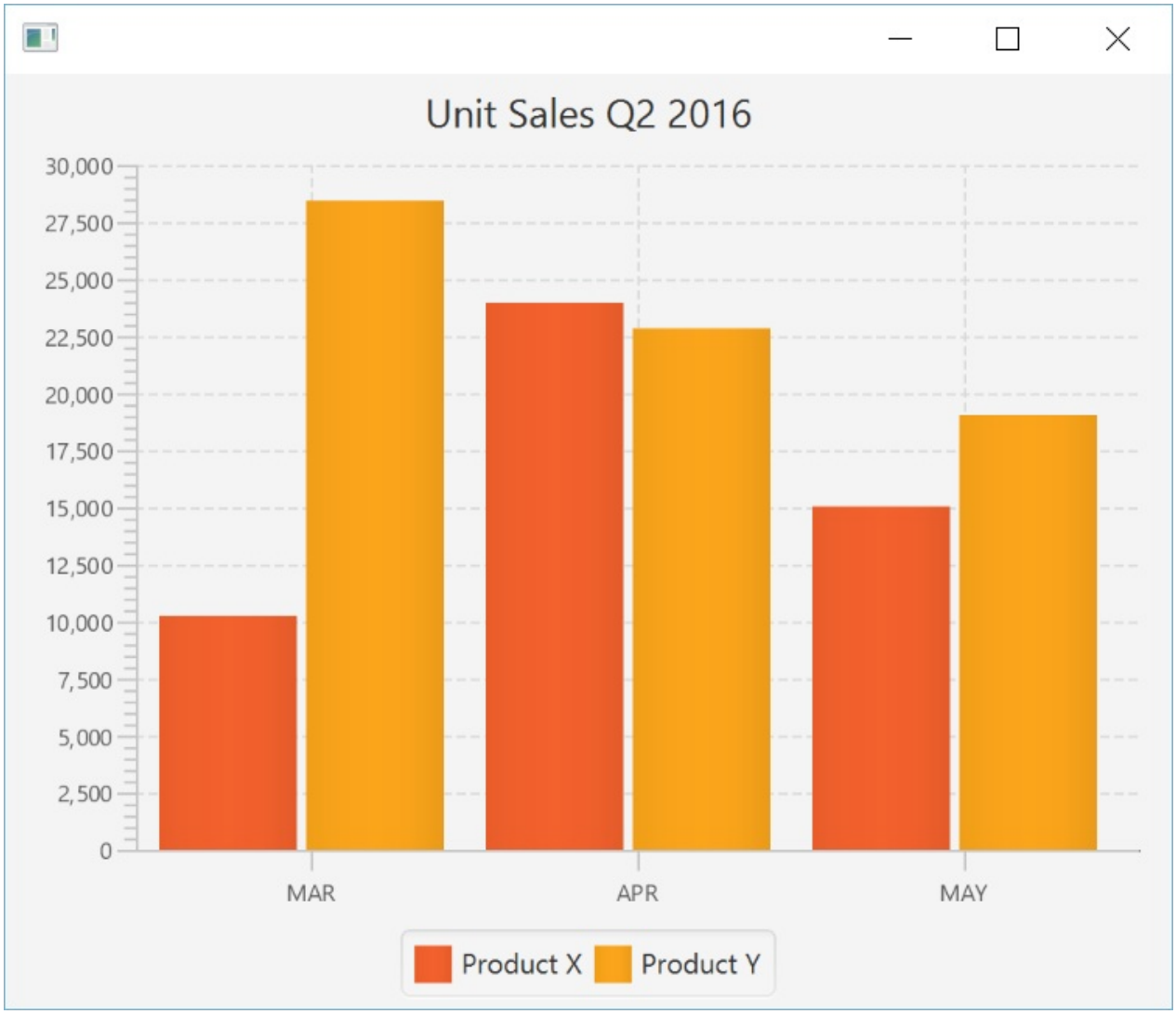
Most charts often deal with one or more series of data points on an XY axis. The most common are bar and line charts.

Bar Charts

You can represent one or more series of data points through a `BarChart`. This chart makes it easy to compare different data points relative to their distance from the X or Y axis (Figure 8.2).

```
barchart("Unit Sales Q2 2016", CategoryAxis(), NumberAxis()) {
   series("Product X") {
       data("MAR", 10245)
       data("APR", 23963)
       data("MAY", 15038)
   }
   series("Product Y") {
       data("MAR", 28443)
       data("APR", 22845)
       data("MAY", 19045)
   }
}
```

Figure 8.2



Above, the `series()` and `data()` functions allow quick construction of data structures backing the charts. On construction, you will need to construct the proper `Axis` type for each X and Y axis. In this example, the months are not necessarily numeric but rather Strings. Therefore they are best represented by a `CategoryAxis`. The units, already being numeric, are fit to use a `NumberAxis`.

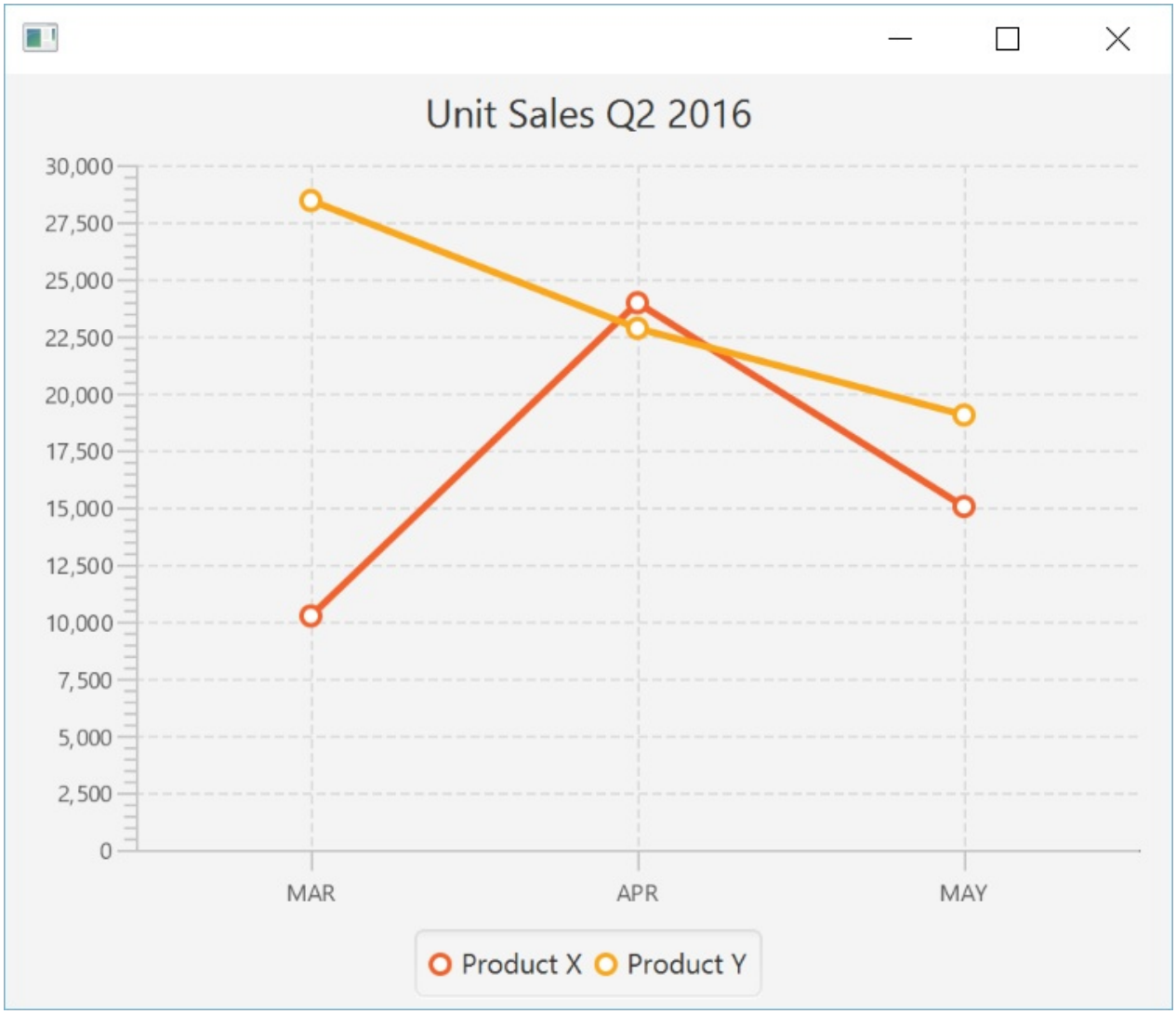
> In the `series()` and `data()` blocks, you can customize further properties like colors. You can even call `style()` to quickly apply type-safe CSS to the chart.

LineChart and AreaChart

A `LineChart` connects data points on an XY axis with lines, quickly visualizing upward and downward trends between them (Figure 8.3)

```
linechart("Unit Sales Q2 2016", CategoryAxis(), NumberAxis()) {
    series("Product X") {
        data("MAR", 10245)
        data("APR", 23963)
        data("MAY", 15038)
    }
    series("Product Y") {
        data("MAR", 28443)
        data("APR", 22845)
        data("MAY", 19045)
    }
}
```

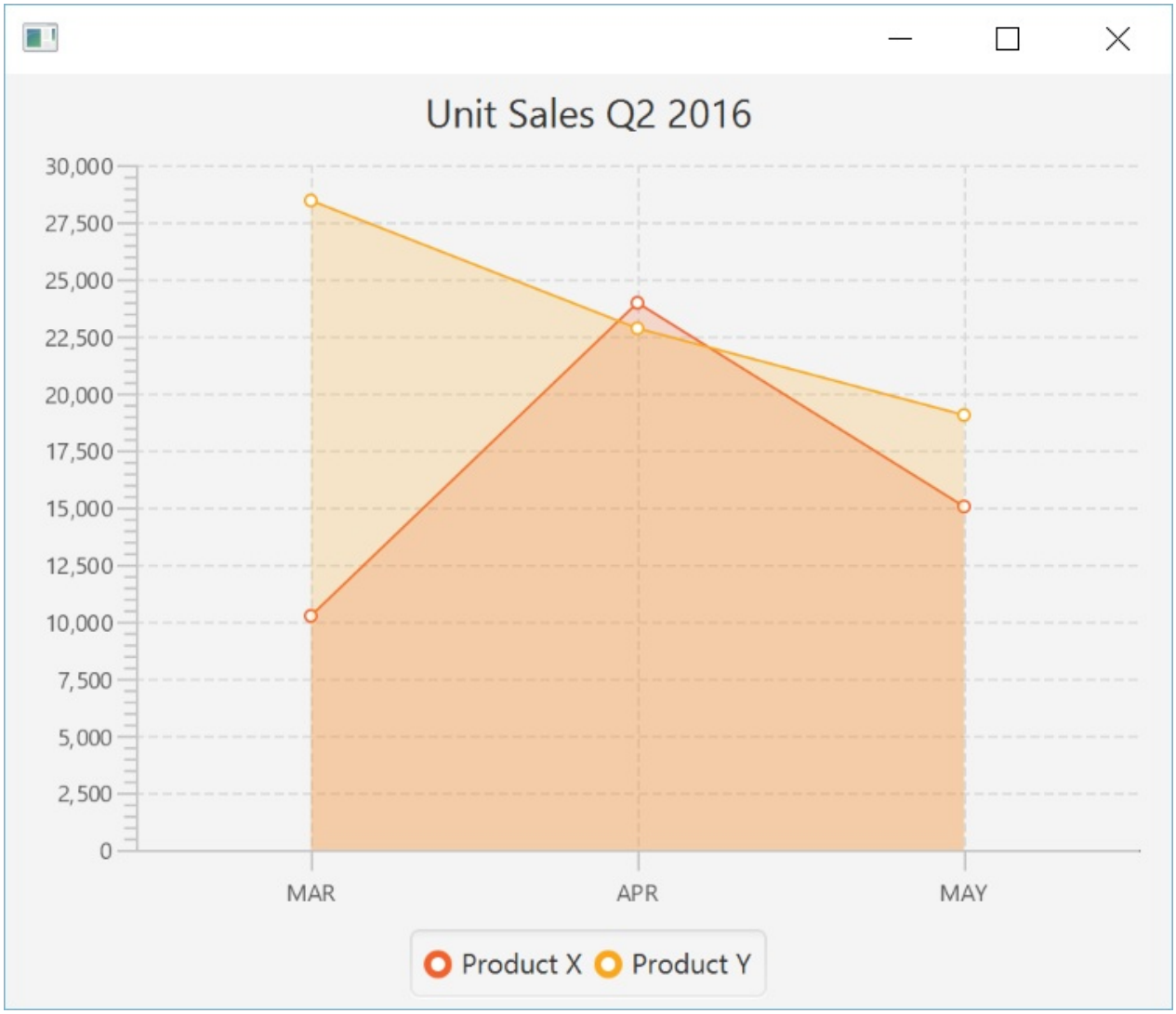
Figure 8.3



The backing data structure is not much different than a `BarChart`, and you use the `series()` and `data()` functions in the same manner.

You can also use a variant of `LineChart` called `AreaChart`, which will shade the area under the lines a distinct color, as well as any overlaps (Figure 8.4).

Figure 8.4



Multiseries

You can streamline the declaration of more than one series using the `multiseries()` function, and call the `data()` functions with `varargs` values. We can consolidate our previous example using this construct:

```
linechart("Unit Sales Q2 2016", CategoryAxis(), NumberAxis()) {

    multiseries("Product X", "Product Y") {
        data("MAR", 10245, 28443)
        data("APR", 23963, 22845)
        data("MAY", 15038, 19045)
    }
}
```

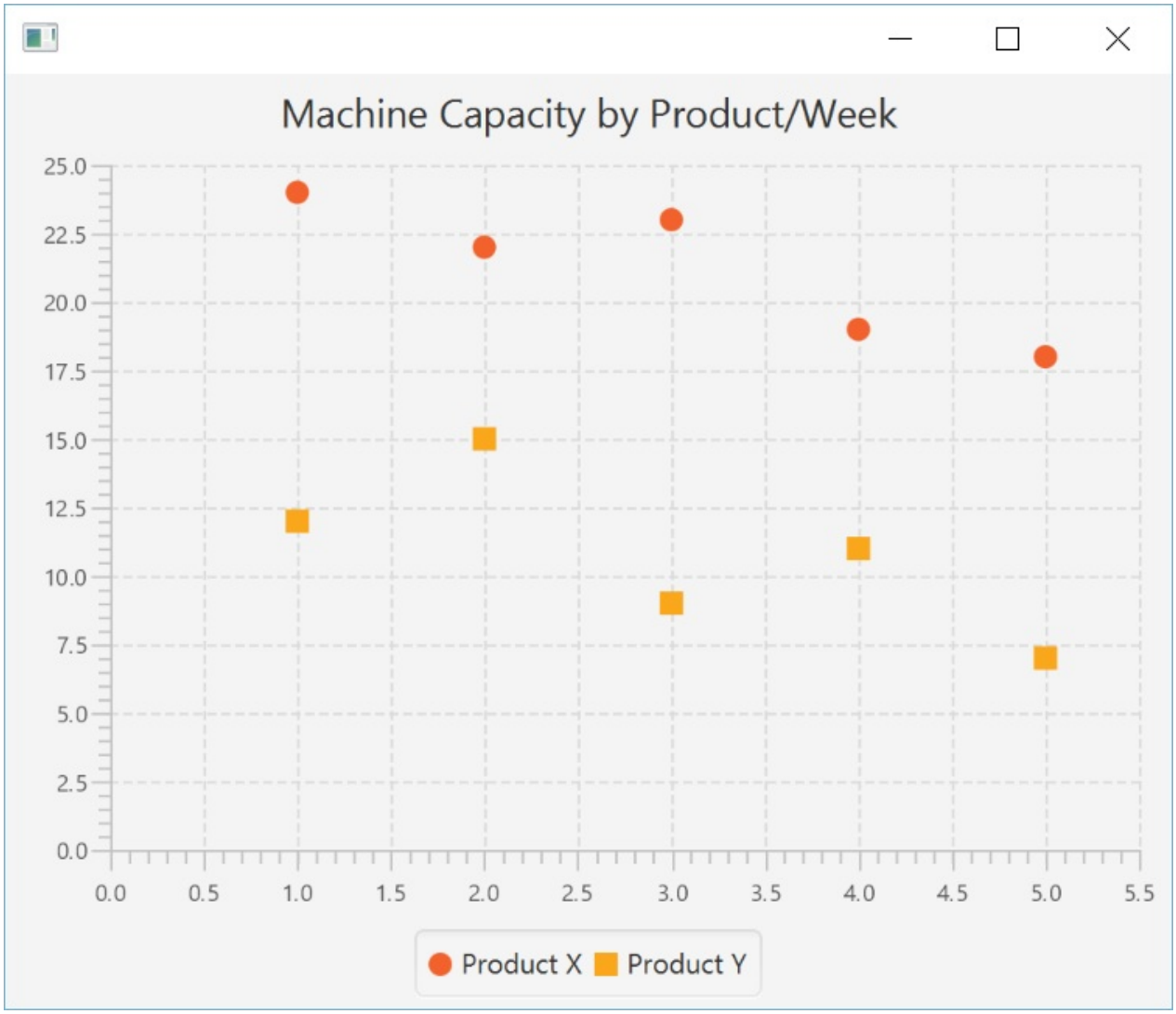
This is just another convenience to reduce boilerplate and quickly declare your data structure for a chart.

ScatterChart

A `ScatterChart` is the simplest representation of an XY data series. It plots the points without bars or lines. It is often used to plot a large volume of data points in order to find clusters. Here is a brief example of a `ScatterChart` plotting machine capacities by week for two different product lines (Figure 8.5).

```
scatterchart("Machine Capacity by Product/Week", NumberAxis(), NumberAxis()) {
    series("Product X") {
        data(1,24)
        data(2,22)
        data(3,23)
        data(4,19)
        data(5,18)
    }
    series("Product Y") {
        data(1,12)
        data(2,15)
        data(3,9)
        data(4,11)
        data(5,7)
    }
}
```

Figure 8.5

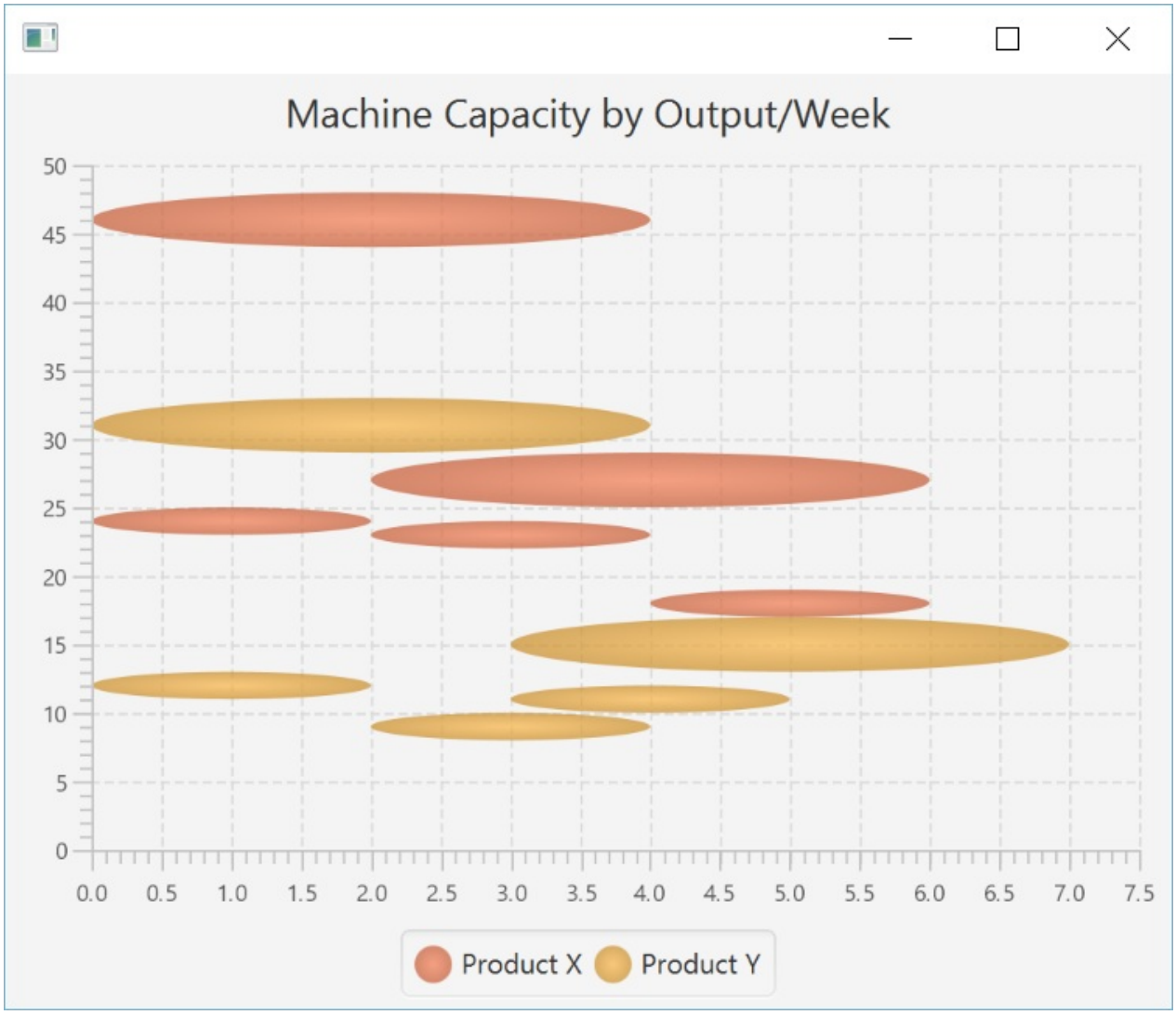


BubbleChart

`BubbleChart` is another XY chart similar to the `ScatterPlot`, but there is a third variable to control the radius of each point. You can leverage this to show, for instance, output by week with the bubble radii reflecting number of machines used (Figure 8.6).

```
bubblechart("Machine Capacity by Output/Week", NumberAxis(), NumberAxis()) {
    series("Product X") {
        data(1,24,1)
        data(2,46,2)
        data(3,23,1)
        data(4,27,2)
        data(5,18,1)
    }
    series("Product Y") {
        data(1,12,1)
        data(2,31,2)
        data(3,9,1)
        data(4,11,1)
        data(5,15,2)
    }
}
```

Figure 8.6



Summary

Charts are a an effective way to visualize data, and the builders in TornadoFX help create them quickly. You can read more about JavaFX charts in Oracle's documentation. If you need more advanced charting functionality, there are libraries like JFreeChart and Orson Charts you can leverage and interop with TornadoFX, but this is beyond the scope of this book.

Shapes and Animation

JavaFX comes with nodes that represent almost any geometric shape as well as a `Path` node that provides facilities required for assembly and management of a geometric path (to create custom shapes). JavaFX also has animation support to gradually change a `Node` property, creating a visual transition between two states. TornadoFX seeks to streamline all these features through builder constructs.

Shape Basics

Every parameter to the shape builders are optional, and in most cases default to a value of `0.0`. This means that you only need to provide the parameters you care about. The builders have positional parameters for most of the properties of each shape, and the rest can be set in the functional block that follows. Therefore these are all valid ways to create a rectangle:

```
rectangle {
    width = 100.0
    height = 100.0
}

rectangle(width = 100.0, height = 100.0)

rectangle(0.0, 0.0, 100.0, 100.0)
```

The form you choose is a matter of preference, but obviously consider the legibility of the code you write. The examples in this chapter specify most of the properties inside the code block for clarity, except when there is no code block support or the parameters are reasonably self-explanatory.

Positioning within the Parent

Most of the shape builders give you the option to specify the location of the shape within the parent. Whether or not this will have any effect depends on the parent node. An `HBox` will not care about the `x` and `y` coordinates you specify unless you call `setManaged(false)` on the shape. However, a `Group` control will. The screenshots in the following examples will be created by wrapping a `StackPane` with padding around a `Group`, and finally the shape was created inside that `Group` as shown below.

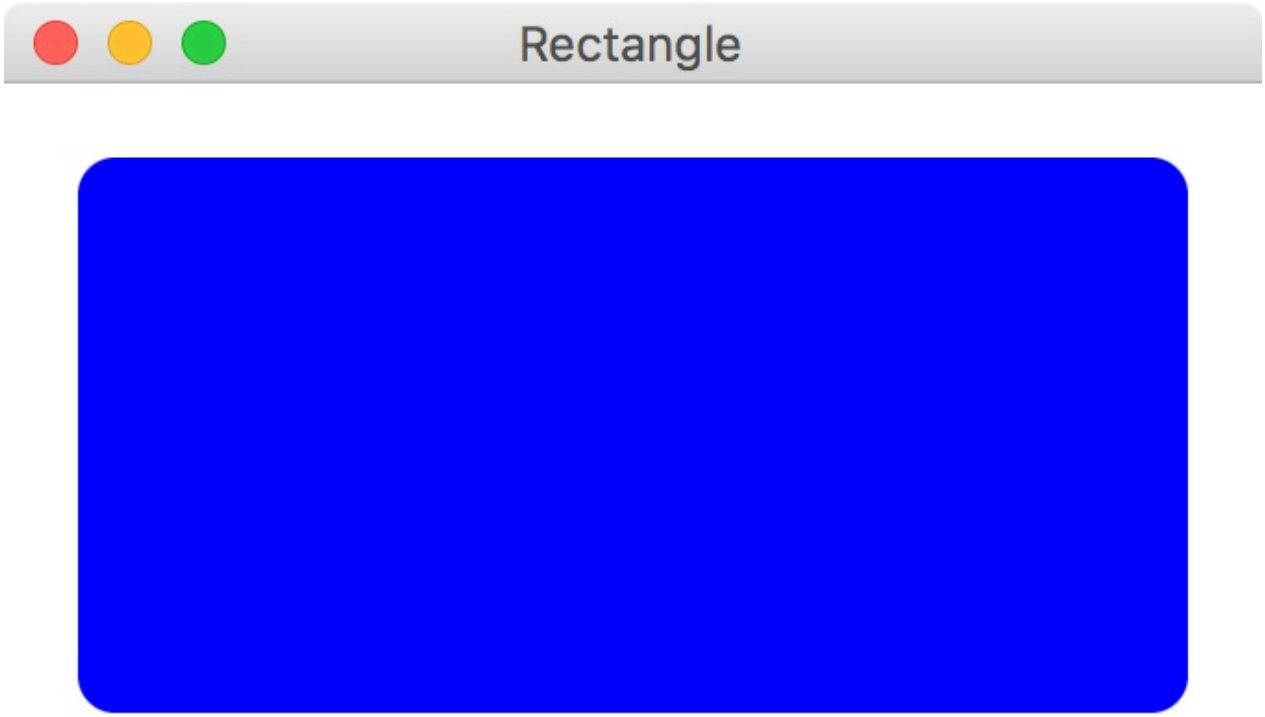
```
class MyView: View() {

    override val root =  stackpane {
       group {
            //shapes will go here
       }
    }
}
```

Rectangle

`Rectangle` defines a rectangle with an optional size and location in the parent. Rounded corners can be specified with the `arcWidth` and `arcHeight` properties (Figure 9.1).

```
rectangle {
    fill = Color.BLUE
    width = 300.0
    height = 150.0
    arcWidth = 20.0
    arcHeight = 20.0
}
```

Figure 9.1

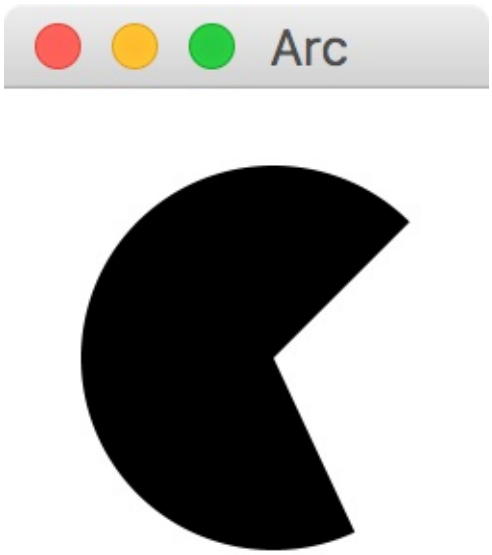


Arc

`Arc` represents an arc object defined by a center, start angle, angular extent (length of the arc in degrees), and an arc type (`OPEN` , `CHORD` , or `ROUND`) (Figure 9.2)

```
arc {
    centerX = 200.0
    centerY = 200.0
    radiusX = 50.0
    radiusY = 50.0
    startAngle = 45.0
    length = 250.0
    type = ArcType.ROUND
}
```

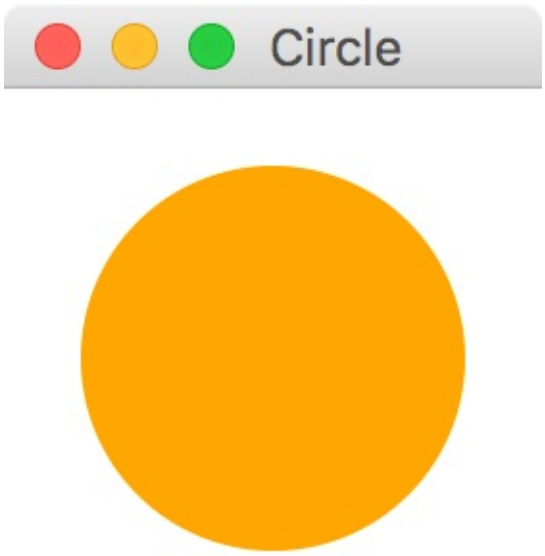
Figure 9.2



Circle

`Circle` represents a circle with the specified `radius` and `center` .

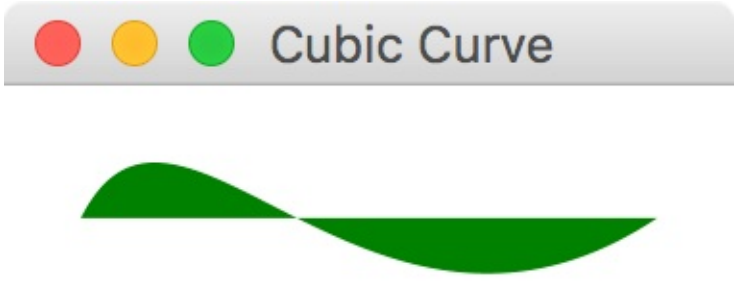
```
circle {
    centerX = 100.0
    centerY = 100.0
    radius = 50.0
}
```



CubicCurve

`CubicCurve` represents a cubic Bézier parametric curve segment in (x,y) coordinate space. Drawing a curve that intersects both the specified coordinates (`startX` , `startY`) and (`endX` , `enfY`), using the specified points (`controlX1` , `controlY1`) and (`controlX2` , `controlY2`) as Bézier control points.

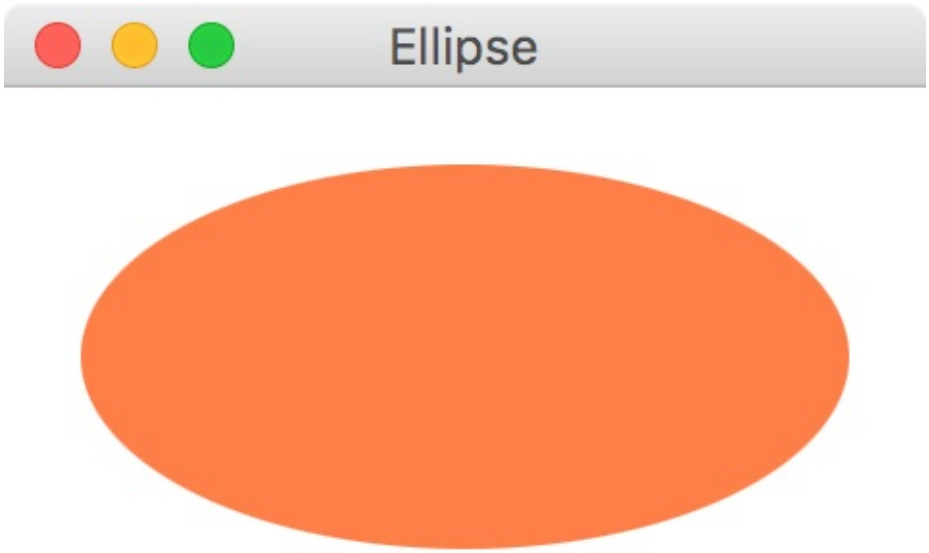
```
cubiccurve {
    startX = 0.0
    startY = 50.0
    controlX1 = 25.0
    controlY1 = 0.0
    controlX2 = 75.0
    controlY2 = 100.0
    endX = 150.0
    endY = 50.0
    fill = Color.GREEN
}
```



Ellipse

`Ellipse` represents an ellipse with parameters specifying size and location.

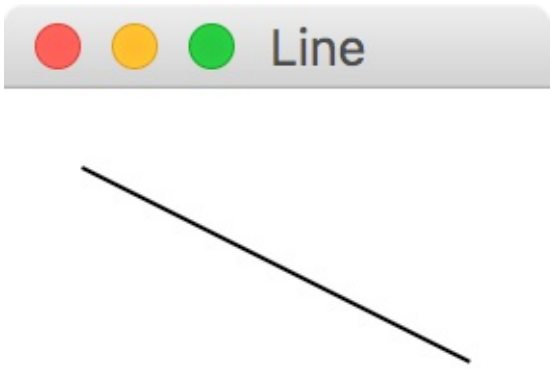
```
ellipse {
    centerX = 50.0
    centerY = 50.0
    radiusX = 100.0
    radiusY = 50.0
    fill = Color.CORAL
}
```



Line

Line is fairly straight forward. Supply start and end coordinates to draw a line between the two points.

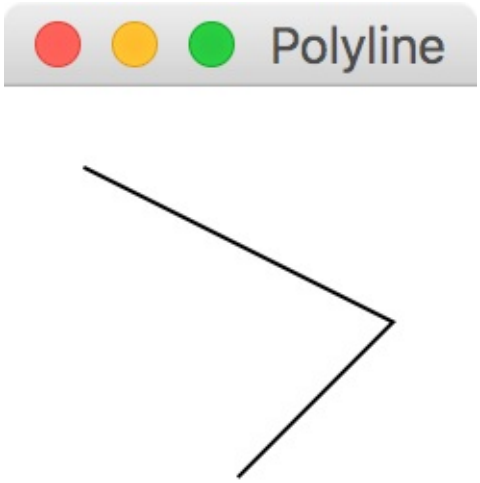
```
line {
    startX = 50.0
    startY = 50.0
    endX = 150.0
    endY = 100.0
}
```



Polyline

A `Polyline` is defined by an array of segment points. `Polyline` is similar to `Polygon`, except it is not automatically closed.

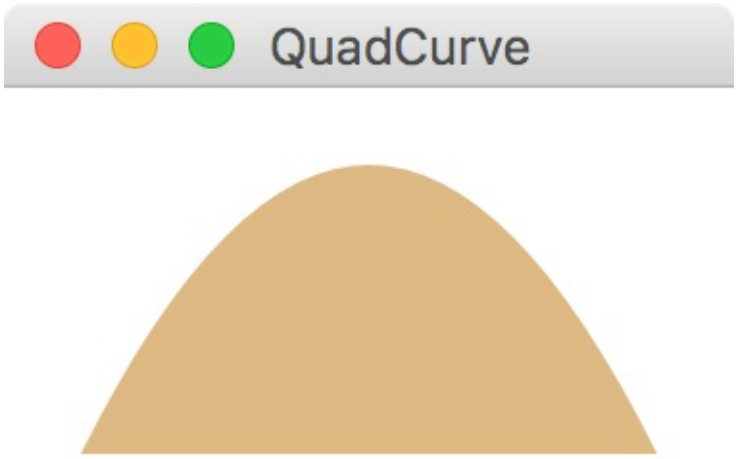
```
polyline(0.0, 0.0, 80.0, 40.0, 40.0, 80.0)
```



QuadCurve

The `Quadcurve` represents a quadratic Bézier parametric curve segment in (x,y) coordinate space. Drawing a curve that intersects both the specified coordinates (`startX` , `startY`) and (`endX` , `endY`), using the specified point (`controlX` , `controlY`) as Bézier control point.

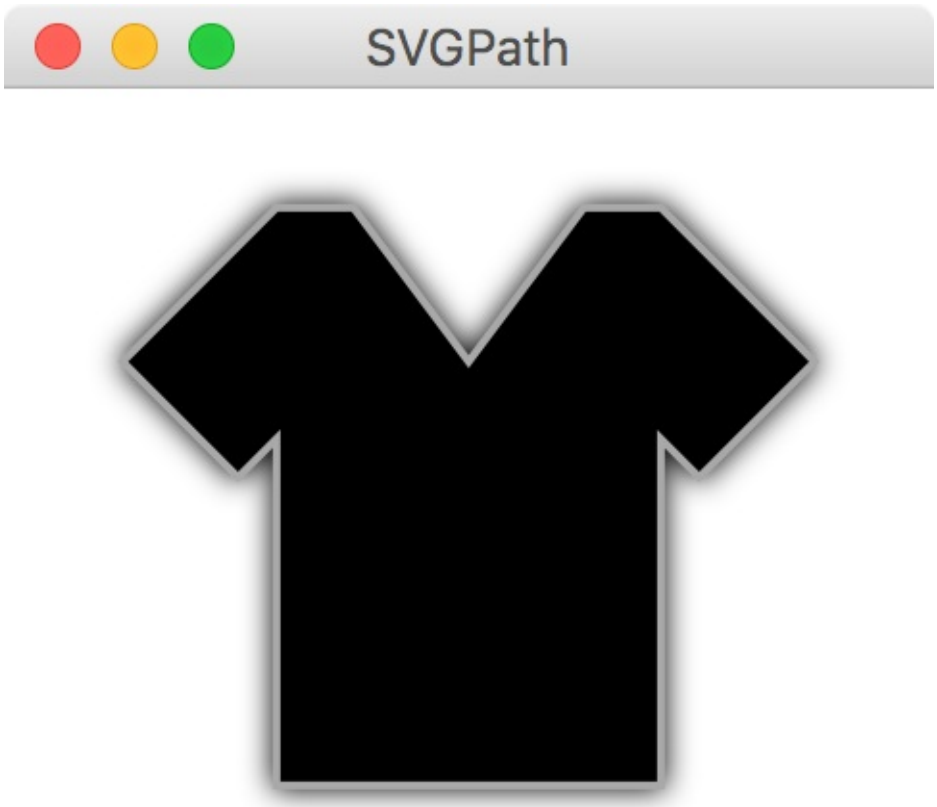
```
quadcurve {
    startX = 0.0
    startY = 150.0
    endX = 150.0
    endY = 150.0
    controlX = 75.0
    controlY = 0.0
    fill = Color.BURLYWOOD
}
```



SVGPath

`SVGPath` represents a shape that is constructed by parsing SVG path data from a String.

```
svgpath("M70,50 L90,50 L120,90 L150,50 L170,50 L210,90 L180,120 L170,110 L170,200 L70,
200 L70,110 L60,120 L30,90 L70,50") {
    stroke = Color.DARKGREY
    strokeWidth = 2.0
    effect = DropShadow()
}
```

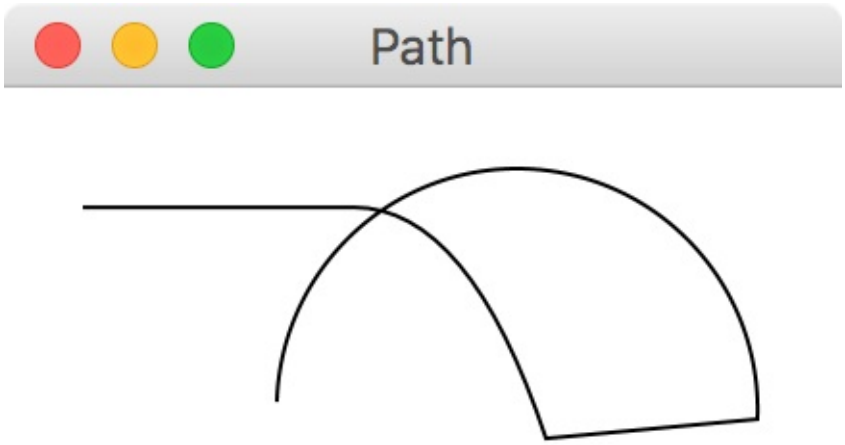


Path

`Path` represents a shape and provides facilities required for basic construction and management of a geometric path. In other words, it helps you create a custom shape. The following helper functions can be used to constuct the path:

- `moveTo(x, y)`
- `hlineTo(x)`
- `vlineTo(y)`
- `quadqurveTo(controlX, controlY, x, y)`
- `lineTo(x, y)`
- `arcTo(radiusX, radiusY, xAxisRotation, x, y, largeArcFlag, sweepFlag)`
- `closepath()`

```
path {
    moveTo(0.0, 0.0)
    hlineTo(70.0)
    quadqurveTo {
        x = 120.0
        y = 60.0
        controlX = 100.0
        controlY = 0.0
    }
    lineTo(175.0, 55.0)
    arcTo {
        x = 50.0
        y = 50.0
        radiusX = 50.0
        radiusY = 50.0
    }
}
```



Animation

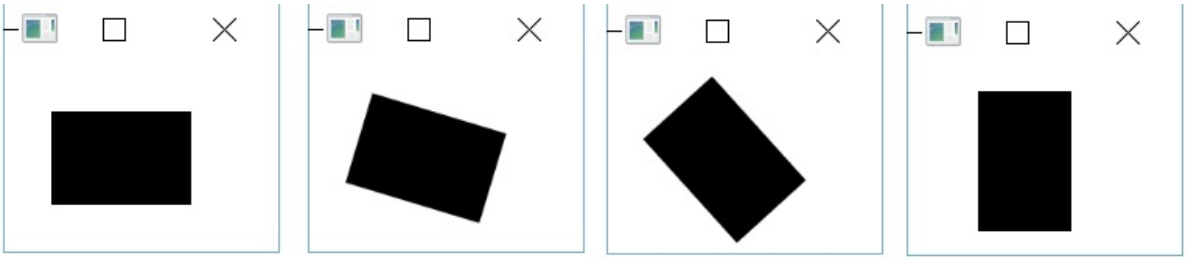
JavaFX has tools to animate any `Node` by gradually changing one or more of its properties. There are three components you work with to create animations in JavaFX.

- `Timeline` - A sequence of `KeyFrame` items executed in a specified order
- `KeyFrame` - A "snapshot" specifying value changes on one or more writable properties (via a `KeyValue`) on one or more Nodes
- `KeyValue` - A pairing of a `Node` property to a value that will be "transitioned" to

A `KeyValue` is the basic building block of JavaFX animation. It specifies a property and the "new value" it will gradually be transitioned to. So if you have a `Rectangle` with a `rotateProperty()` of `0.0`, and you specify a `KeyValue` that changes it to `90.0` degrees, it will incrementally move from `0.0` to `90.0` degrees. Put that `KeyValue` inside a `KeyFrame` which will specify how long the animation between those two values will last. In this case let's make it 5 seconds. Then finally put that `KeyFrame` in a `Timeline`. If you run the code below, you will see a rectange gradually rotate from `0.0` to `90.0` degrees in 5 seconds (Figure 9.1).

```
val rectangle = rectangle(width = 60.0,height = 40.0) {
    padding = Insets(20.0)
}
timeline {
    keyframe(Duration.seconds(5.0)) {
        keyvalue(rectangle.rotateProperty(),90.0)
    }
}
```

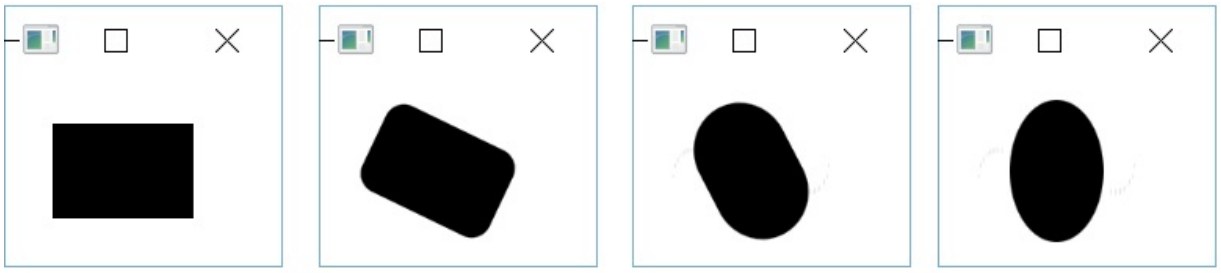
Figure 9.1



In a given `KeyFrame`, you can simultaneously manipulate other properties in that 5-second window too. For instance we can transition the `arcWidthProperty()` and `arcHeightProperty()` while the `Rectangle` is rotating (Figure 9.2)

```
timeline {
        keyframe(Duration.seconds(5.0)) {
            keyvalue(rectangle.rotateProperty(),90.0)
            keyvalue(rectangle.arcWidthProperty(),60.0)
            keyvalue(rectangle.arcHeightProperty(),60.0)
        }
}
```

Figure 9.2



Interpolators

You can also specify an `Interpolator` which can add subtle effects to the animation. For instance, you can specify `Interpolator.EASE_BOTH` to accelerate and decelerate the value change at the beginning and end of the animation gracefully.

```
val rectangle = rectangle(width = 60.0, height = 40.0) {
    padding = Insets(20.0)
}

timeline {
    keyframe(5.seconds) {
        keyvalue(rectangle.rotateProperty(), 180.0, interpolator = Interpolator.EASE_B
OTH)
    }
}
```

Cycles and AutoReverse

You can modify other attributes of the `timeline()` such as `cycleCount` and `autoReverse`. The `cycleCount` will repeat the animation a specified number of times, and setting the `isAutoReverse` to true will cause it to revert back with each cycle.

```
timeline {
    keyframe(5.seconds) {
        keyvalue(rectangle.rotateProperty(), 180.0, interpolator = Interpolator.EASE_B
OTH)
    }
    isAutoReverse = true
    cycleCount = 3
}
```

To repeat the animation indefinitely, set the `cycleCount` to `Timeline.INDEFINITE`.

Shorthand Animation

If you want to animate a single property, you can quickly animate it without declaring a `timeline()`, `keyframe()`, and `keyset()`. Call the `animate()` extension function on that propert and provide the `endValue`, the `duration`, and optionally the `interoplator`. This is much shorter and cleaner if you are animating just one property.

```
rectangle.rotateProperty().animate(endValue = 180.0, duration = 5.seconds)
```

Summary

In this chapter we covered builders for shape and animation. We did not cover JavaFX's `Canvas` as this is beyond the scope of the `TornadoFX` framework. It could easily take up more than several chapters on its own. But the shapes and animation should allow you to do basic custom graphics for a majority of tasks.

This concludes our coverage of TornadoFX builders for now. Next we will cover FXML for those of us that have need to use it.

FXML and Internationalization

TornadoFX's type-safe builders provide a fast, easy, and declarative way to construct UI's. This DSL approach is encouraged because it is more flexible, reliable, and simpler. However, JavaFX also supports an XML-based structure called FXML that can also build a UI layout. TornadoFX has tools to streamline FXML usage for those that need it.

If you are unfamiliar with FXML and are perfectly happy with type-safe builders, please feel free to skip this chapter. If you need to work with FXML or feel you should learn it, please read on. You can also take a look at the official FXML documentation to learn more.

Reasons for Considering FXML

While the developers of TornadoFX strongly encourage using type-safe builders, there are situations and factors that might cause you to consider using FXML.

Separation of Concerns

With FXML it is easy to separate your UI logic code from the UI layout code. This separation is just as achievable with type-safe builders by utilizing MVP or other separation pattern. But some programmers find FXML forces them to maintain this separation and prefer it for that reason.

WYSIWYG Editor

FXML files also can be edited and processed by Scene Builder, a visual layout tool that allows building interfaces via drag-and-drop functionality. Edits in Scene Builder are immediately rendered in a WYSIWYG ("What You See is What You Get") pane next to the editor.

If you prefer making interfaces via drag-and-drop, or have trouble building UI's with pure code, you might consider using FXML simply to leverage Scene Builder.

> The Scene Builder tool was created by Oracle/Sun but is now maintained by Gluon, an innovative company that invests heavily in JavaFX technology, especially for the mobile market.

Compatibility with Existing Codebases

If you are converting an existing JavaFX application to TornadoFX, there is a strong chance your UI was constructed with FXML. If you hesitate to transition legacy FXML to TornadoFX buidlers, or would like to put that off as long as possible, TornadoFX can at least streamline the processing of FXML.

How FXML works

The `root` property of a `View` represents the top level `Node` containing a hierarchy of children Nodes, which makes up the user interface. When you work with FXML, you do not instantiate this root node directly, but instead ask TornadoFX to load it from a corresponding FXML file. By default, TornadoFX will look for a file with the same name as your view with the `.fxml` file ending in the same package as your `View` class. You can also override the FXML location with a parameter if you want to put all your FXML files in a single folder or organize them some other way that does not directly correspond to your `View` location.

A Simple Example

Let's create a basic user interface that presents a `Label` and a `Button`. We will add functionality to this view so when the `Button` is clicked, the `Label` will update its `text` with the number of times the `Button` has been clicked.

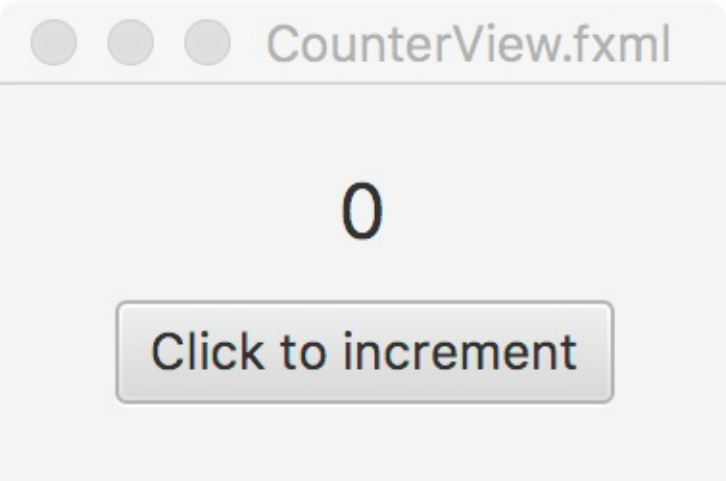
Create a file named `CounterView.fxml` with the following content:

```
<?import javafx.geometry.Insets?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.BorderPane?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.text.Font?>
<BorderPane xmlns="http://javafx.com/javafx/null" xmlns:fx="http://javafx.com/fxml/1">
    <padding>
        <Insets top="20" right="20" bottom="20" left="20"/>
    </padding>

    <center>
        <VBox alignment="CENTER" spacing="10">
            <Label text="0">
                <font>
                    <Font size="20"/>
                </font>
            </Label>
            <Button text="Click to increment" />
        </VBox>
    </center>
</BorderPane>
```

> You may notice above you have to `import` the types you use in FXML just like coding in Java or Kotlin. Intellij IDEA should have a plugin to support using ALT+ENTER to generate the `import` statements.

If you load this file in Scene Builder you will see the following result (Figure 9.1).

Figure 9.1



Next let's load this FXML into TornadoFX.

Loading FXML into TornadoFX

We have created an FXML file containing our UI structure, but now we need to load it into a TornadoFX `View` for it to be usable. Logically, we can load this `Node` hierarchy into the `root` node of our `View`. Define the following `View` class:

```
class CounterView : View() {
    override val root : BorderPane by fxml()
}
```

Note that the `root` property is defined by the `fxml()` delegate. The `fxml()` delegate takes care of loading the corresponding `CounterView.fxml` into the `root` property. If we placed `CounterView.fxml` in a different location (such as `/views/`) that is different than where the `CounterView` file resides, we would add a parameter.

```
class CounterView : View() {
    override val root : BorderPane by fxml("/views/CounterView.fxml")
}
```

We have laid out the UI, but it has no functionality yet. We need to define a variable that holds the number of times the `Button` has been clicked. Add a variable called `counter` and define a function that will increment its value:

```
class CounterView : View() {
    override val root : BorderPane by fxml()
    val counter = SimpleIntegerProperty()

    fun increment() {
        counter.value += 1
    }
}
```

We want the `increment()` function to be called whenever the `Button` is clicked. Back in the FXML file, add the `onAction` attribute to the button:

```
<Button text="Click to increment" onAction="#increment"/>
```

Since the FXML file automatically gets bound to our `View`, we can reference functions via the `#functionName` syntax. Note that we do not add parenthesis to the function call, and you cannot pass parameters directly. You can however add a parameter of type `javafx.event.ActionEvent` to the `increment` function if you want inspect the source `Node` of the action or check what kind of action triggered the button. For this example we do not need it, so we leave the `increment` function without parameters.

FXML file locations

By default, build tools like Maven and Gradle will ignore any extra resources you put into your source root folders, so if you put your FXML files there they won't be available at runtime unless you specifically tell your build tool to include them. This could still be problematic because IDEA might not pick up your custom resource location from the build file, once again resulting in failure at runtime. For that resource, we recommend that you place your FXML files in `src/main/resources` and either follow the same folder structure as your packages, or put them all in a `views` folder or similar. The latter requires you to add the FXML location parameter to the `fxml` delegate, and might be messy if you have a large number of Views, so going with the default is a good idea.

Accessing Nodes with the `fxid` delegate

Using just FXML, we have wired the `Button` to call `increment()` every time it is called. We still need to bind the `counter` value to the `text` property of the `Label`. To do this, we need an identifier for the `Label`, so in our FXML file we add the `fx:id` attribute to it.

```
<Label fx:id="counterLabel">
```

Now we can inject this `Label` into our `View` class:

```
val counterLabel : Label by fxid()
```

This tells TornadoFX to look for a `Node` in our structure with the `fx:id` property with the same name as the property we defined (which is "counterLabel"). It is also possible to use another property name in the `View` and add a name parameter to the `fxid` delegate:

```
val myLabel : Label by fxid("counterLabel")
```

Now that we have a hold of the `Label`, we can use the binding shortcuts of TornadoFX to bind the `counter` value to the `text` property of the `counterLabel`. Our whole `View` should now look like this:

```
class CounterView : View() {
    override val root : BorderPane by fxml()
    val counter = SimpleIntegerProperty()
    val counterLabel: Label by fxid()

    init {
        counterLabel.bind(counter)
    }

    fun increment() {
        counter.value += 1
    }
}
```

Our app is now complete. Every time the button is clicked, the `label` will increment its count.

Internationalization

JavaFX has strong support for multi-language UI's. To support internationalization in FXML, you normally have to register a resource bundle with the `FXMLLoader` and it will in return replace instances of resource names with their locale-specific value. A resource name is the key in the resource bundle prepended with `%`.

TornadoFX makes this easier by supporting a convention for resource bundles: Create a resource bundle with the same base name as your `View`, and it will be automatically loaded, both for use programatically within the `View` and from the FXML file.

Let's internationalize the button text in our UI. Create a file called `CounterView.properties` and add the following content:

```
clickToIncrement=Click to increment
```

If you want to support multiple languages, create a file with the same base name followed by an underscore, and then the language code. For instance, to support French create the file `CounterView_fr.properties`. The closest geographical match to the current locale will be used.

```
clickToIncrement=Cliquez sur incrément
```

Now we swap the button text with the resource key in the FXML file.

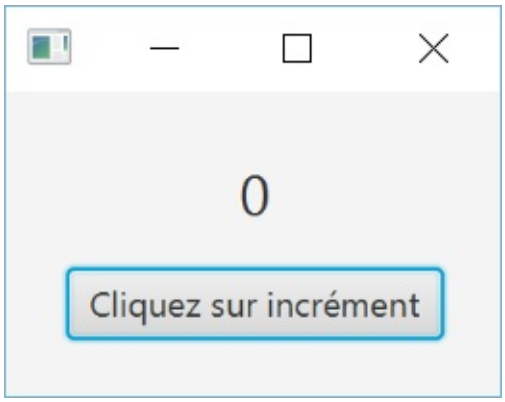
```
<Button text="%clickToIncrement" onAction="#increment"/>
```

If you want to test this functionality and force a different `Locale`, regardless which one you are currently in, override it by assigning `FX.local` when your `App` class is initialized.

```
class MyApp: App() {
    override val primaryView = MyView::class

    init {
        FX.locale = Locale.FRENCH
    }
}
```

You should then see your `Button` use the French text (Figure 9.2).

Figure 9.2



Internationalization with Type-Safe Builders

Internationalization is not limited for use with FXML. You can also use it with type-safe builders. Set up your `.properties` files as specified before. But instead of using an embedded `%clickToIncrement` text in an FXML file, use the `messages[]` accessor to look up the value in the `ResourceBundle`. Pass this value as the `text` for the `Button`.

```
button(messages["clickToIncrement"]) {
    setOnAction { increment() }
}
```

Summary

FXML is helpful to know as a JavaFX developer, but it is definitely not required if you are content with TornadoFX type-safe builders and do not have any existing JavaFX applications to maintain. Type-safe builders have the benefit of using pure Kotlin, allowing you to code anything you want right within the structure declarations. FXML's benefits are primarily separation of concerns between UI and functionality, but even that can be accomplished with type-safe builders. It also can be built via drag-and-drop through the Scene Builder tool, which may be preferable for those who struggle to build UI's any other way.

Editing Models and Validation

TornadoFX doesn't force any particular architectural pattern on you as a developer, and it works equally well with both MVC, MVP, and their derivatives.

To help with implementing these patterns TornadoFX provides a tool called `ViewModel` that helps cleanly separate your UI and business logic, giving you features like *rollback*/*commit* and *dirty state checking*. These patterns are hard or cumbersome to implement manually, so it is advised to leverage the `ViewModel` and `ViewModelItem` when it is needed.

Typically you will use the `ItemViewModel` when you are creating a facade in front of a single object, and a `ViewModel` for more complex situations.

A Typical Use Case

Say you have a given domain type `Person`. We allow its two properties to be nullable so they can be inputted later by the user.

```
import tornadofx.*

class Person(name: String? = null, title: String? = null) {
    val nameProperty = SimpleStringProperty(this, "name", name)
    var name by nameProperty

    val titleProperty = SimpleStringProperty(this, "title", title)
    var title by titleProperty
}
```

(Notice the import, you need to import at least `tornadofx.getValue` and `tornadofx.setValue` for the *by* delegate to work)*[]:

Consider a Master/Detail view where you have a `TableView` displaying a list of people, and a `Form` where the currently selected person's information can be edited. Before we get into the `ViewModel`, we will create a version of this `View` without using the `ViewModel`.

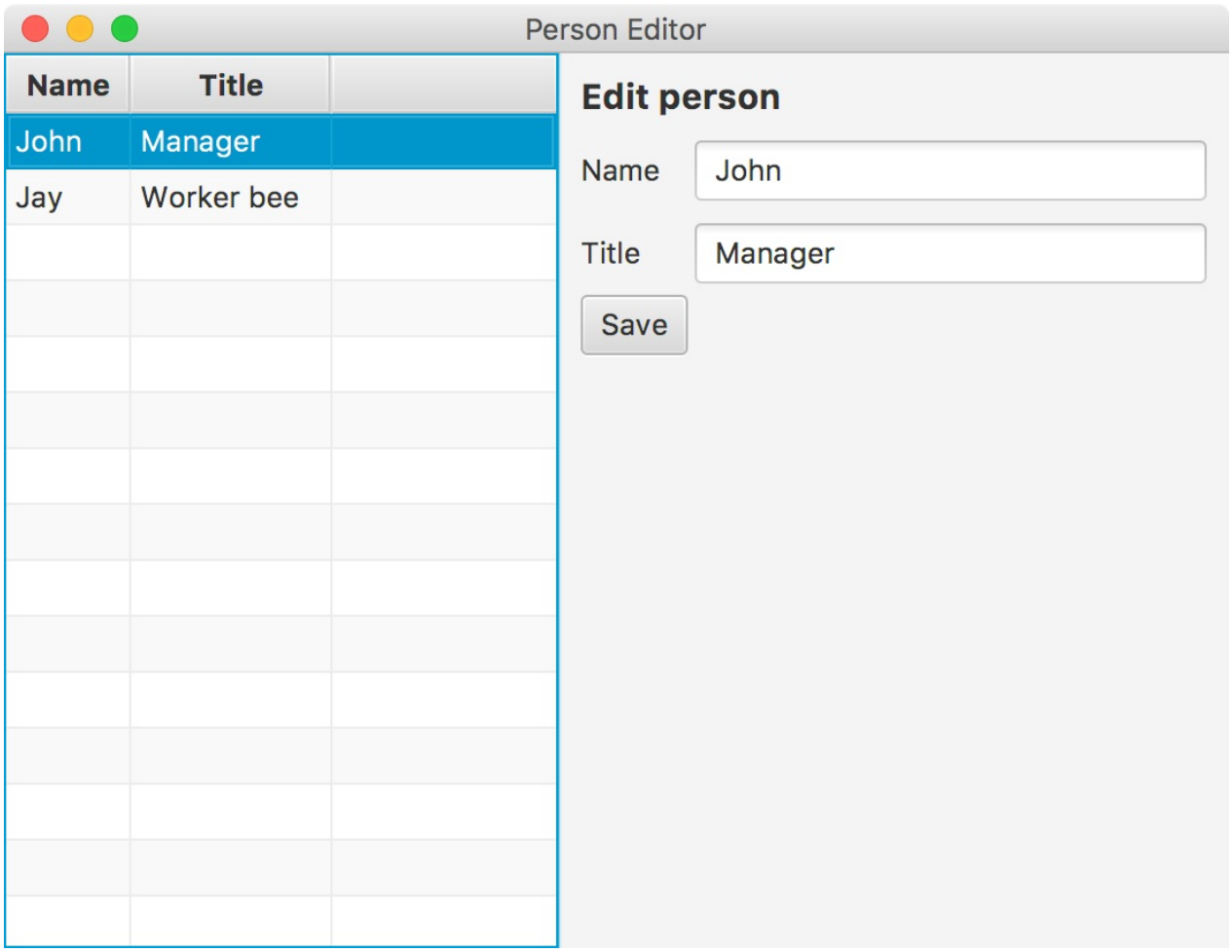


Figure 11.1

Below is code for our first attempt in building this, and it has a number of problems we will address.

```
import javafx.scene.control.TableView
import javafx.scene.control.TextField
import javafx.scene.layout.BorderPane
import tornadofx.*

class Person(name: String? = null, title: String? = null) {
    val nameProperty = SimpleStringProperty(this, "name", name)
    var name by nameProperty

    val titleProperty = SimpleStringProperty(this, "title", title)
    var title by titleProperty
}

class PersonEditor : View("Person Editor") {
    override val root = BorderPane()
    var nameField : TextField by singleAssign()
    var titleField : TextField by singleAssign()
    var personTable : TableView<Person> by singleAssign()
    // Some fake data for our table
    val persons = listOf(Person("John", "Manager"), Person("Jay", "Worker bee")).obser
```

```
vable()

    var prevSelection: Person? = null

    init {
        with(root) {
            // TableView showing a list of people
            center {
                tableview(persons) {
                    personTable = this
                    column("Name", Person::nameProperty)
                    column("Title", Person::titleProperty)

                    // Edit the currently selected person
                    selectionModel.selectedItemProperty().onChange {
                        editPerson(it)
                        prevSelection = it
                    }
                }
            }

            right {
                form {
                    fieldset("Edit person") {
                        field("Name") {
                            textfield() {
                                nameField = this
                            }
                        }
                        field("Title") {
                            textfield() {
                                titleField = this
                            }
                        }
                        button("Save").action {
                            save()
                        }
                    }
                }
            }
        }
    }

    private fun editPerson(person: Person?) {
        if (person != null) {
            prevSelection?.apply {
                nameProperty.unbindBidirectional(nameField.textProperty)
                titleProperty.unbindBidirectional(titleField.textProperty)
            }
            nameField.bind(person.nameProperty)
            titleField.bind(person.titleProperty)
            prevSelection = person
        }
```

```
    }

    private fun save() {
        // Extract the selected person from the tableView
        val person = personTable.selectedItem!!

        // A real application would persist the person here
        println("Saving ${person.name} / ${person.title}")
    }
}
```

We define a `View` consisting of a `TableView` in the center of a `BorderPane` and a `Form` on the right side. We define some properties for the form fields and the table itself so we can reference them later.

While we build the table, we attach a listener to the selected item so we can call the `editPerson` function when the table selection changes. The `editPerson` function binds the properties of the selected person to the text fields in the form.

Problems with our initial attempt

At first glance it might look OK, but when we dig deeper there are several issues.

Manual binding

Every time the selection in the table changes, we have to unbind/rebind the data for the form fields manually. Apart from the added code and logic, there is another huge problem with this: the data is updated for every change in the text fields, and the changes will even be reflected in the table. While this might look cool and is technically correct, it presents one big problem: what if the user does not want to save the changes? We have no way of rolling back. So to prevent this, we would have to skip the binding altogether and manually extract the values from the text fields, then create a new `Person` object on save. In fact, this is a pattern found in many applications and expected by most users. Implementing a "Reset" button for this form would mean managing variables with the initial values and again assigning those values manually to the text fields.

Tight Coupling

Another issue is when it is time to save the edited person, the save function has to extract the selected item from the table again. For that to happen the save function has to know about the `TableView`. Alternatively it would have to know about the text fields like the `editPerson` function does, and manually extract the values to reconstruct a `Person` object.

Introducing ViewModel

The `ViewModel` is a mediator between the `TableView` and the `Form`. It acts as a middleman between the data in the text fields and the data in the actual `Person` object. As you will see, the code is much shorter and easier to reason about. The implementation code of the `PersonModel` will be shown shortly. For now just focus on its usage.

```
class PersonEditor : View("Person Editor") {
    override val root = BorderPane()
    val persons = listOf(Person("John", "Manager"), Person("Jay", "Worker bee")).obser
vable()
    val model = PersonModel(Person())

    init {
        with(root) {
            center {
                tableview(persons) {
                    column("Name", Person::nameProperty)
                    column("Title", Person::titleProperty)

                    // Update the person inside the view model on selection change
                    model.rebindOnChange(this) { selectedPerson ->
                        person = selectedPerson ?: Person()
                    }
                }
            }

            right {
                form {
                    fieldset("Edit person") {
                        field("Name") {
                            textfield(model.name)
                        }
                        field("Title") {
                            textfield(model.title)
                        }
                        button("Save") {
                            enableWhen(model.dirty)
                            action {
                                save()
                            }
                        }
                        button("Reset").action {
                            model.rollback()
                        }
                    }
                }
            }
        }
    }
```

```
    private fun save() {
        // Flush changes from the text fields into the model
        model.commit()

        // The edited person is contained in the model
        val person = model.person

        // A real application would persist the person here
        println("Saving ${person.name} / ${person.title}")
    }

}
class PersonModel(var person: Person) : ViewModel() {
    val name = bind { person.nameProperty }
    val title = bind { person.titleProperty }
}
```

This looks a lot better, but what exactly is going on here? We have introduced a subclass of `ViewModel` called `PersonModel`. The model holds a `Person` object and has properties for the `name` and `title` fields. We will discuss the model further after we have looked at the rest of the client code.

Note that we hold no reference to the `TableView` or the text fields. Apart from a lot less code, the first big change is the way we update the `Person` inside the model:

```
model.rebindOnChange(this) { selectedPerson ->
    person = selectedPerson ?: Person()
}
```

The `rebindOnChange()` function takes the `TableView` as an argument and a function that will be called when the selection changes. This works with `ListView`, `TreeView`, `TreeTableView`, and any other `ObservableValue` as well. This function is called on the model and has the `selectedPerson` as its single argument. We assign the selected person to the `person` property of the model, or a new `Person` if the selection was empty/null. That way we ensure that there is always data for the model to present.

When we create the TextFields, we bind the model properties directly to it since most `Node` builders accept an `ObservableValue` to bind to.

```
field("Name") {
    textfield(model.name)
}
```

Even when the selection changes, the model properties persist but the values for the properties are updated. We totally avoid the manual binding from our previous attempt.

Another big change in this version is that the data in the table does not update when we type into the text fields. This is because the model has exposed a copy of the properties from the person object and does not write back into the actual person object before we call `model.commit()`. This is exactly what we do in the `save` function. Once `commit` has been called, the data in the facade is flushed back into our person object and the table will now reflect our changes.

Rollback

Since the model holds a reference to the actual `Person` object, we can can reset the text fields to reflect the actual data in our `Person` object. We could add a reset button like this:

```
button("Reset").action {
    model.rollback()
}
```

When the button is pressed, any changes are discarded and the text fields show the actual `Person` object values again.

The PersonModel

We never explained how the `PersonModel` works yet, and you probably have been wondering about how the `PersonModel` is implemented. Here it is:

```
class PersonModel(var person: Person) : ViewModel() {
    val name = bind { person.nameProperty }
    val title = bind { person.titleProperty }
}
```

It can hold a `Person` object, and it has defined two strange-looking properties called `name` and `title` via the `bind` delegate. Yeah it looks weird, but there is a very good reason for it. The `{ person.nameProperty }` parameter for the `bind` function is a lambda that returns a property. This returned property is examined by the `ViewModel`, and a new property of the same type is created. It is then put into the `name` property of the `ViewModel`.

When we bind a text field to the `name` property of the model, only the copy is updated when you type into the text field. The `ViewModel` keeps track of which actual property belongs to which facade, and when you call `commit` the values from the facade are flushed into the actual backing property. On the flip side, when you call `rollback` the exact opposite happens: The actual property value is flushed into the facade.

The reason the actual property is wrapped in a function is that this makes it possible to change the `person` variable and then extract the property from that new person. You can read more about this below (rebinding).

Dirty Checking

The model has a `Property` called `dirty`. This is a `BooleanBinding` which you can observe to enable or disable certain features. For example, we could easily disable the save button until there are actual changes. The updated save button would look like this:

```
button("Save") {
    enableWhen(model.dirty)
    action {
        save()
    }
}
```

There is also a plain `val` called `isDirty` which returns a `Boolean` representing the dirty state for the entire model.

One thing to note is that if the backing object is being modified while the `ViewModel` is also modified via the UI, all uncommitted changes in the `ViewModel` are being overridden by the changes in the backing object. That means the data in the `ViewModel` might get lost if external modification of the backing object takes place.

```
val person = Person("John", "Manager")
val model = PersonModel(person)

model.name.value = "Johnny"   //modify the ViewModel
person.name = "Johan"         //modify the underlying object

println("  Person = ${person.name}, ${person.title}")            //output:   Person =
 Johan, Manager
println("Is dirty = ${model.isDirty}")                           //output: Is dirty =
 false
println("   Model = ${model.name.value}, ${model.title.value}")   //output:    Model =
 Johan, Manager
```

As can be seen above the changes in the `ViewModel` got overridden when the underlying object was modified. And the `ViewModel` was not flagged as `dirty`.

Dirty Properties

You can check if a specific property is dirty, meaning that it has been changed compared to the backing source object value.

```
val nameWasChanged = model.isDirty(model.name)
```

There is also an extension property version that accomplishes the same task:

```
val nameWasChanged = model.name.isDirty
```

The shorthand version is an extension `val` on `Property<T>` but it will only work for properties that are bound inside a `ViewModel`. You will find `model.isNotDirty` properties as well.

If you need to dynamically react based on the dirty state of a specific property in the `ViewModel`, you can get a hold of a `BooleanBinding` representing the dirty state of that field like this:

```
val nameDirtyProperty = model.dirtyStateFor(PersonModel::name)
```

Extracting the Source Object Value

To retrieve the backing object value for a property you can call `model.backingValue(property)`.

```
val person = model.backingValue(property)
```

Supporting Objects that Do Not Expose JavaFX Properties

You probably wondered how to deal with domain objects that do not use JavaFX properties. Maybe you have a simple POJO with getters and setters, or normal kotlin `var` type properties. Since `ViewModel` requires JavaFX properties, TornadoFX comes with powerful wrappers that can turn any type of property into an observable JavaFX property. Here are some examples:

```
// Java POJO getter/setter property
class JavaPersonViewModel(person: JavaPerson) : ViewModel() {
    val name = bind { person.observable(JavaPerson::getName, JavaPerson::setName) }
}

// Kotlin var property
class PersonVarViewModel(person: Person) : ViewModel() {
    val name = bind { person.observable(Person::name) }
}
```

As you can see, it is easy to convert any property type to an observable property.

Specific Property Subtypes (IntegerProperty, BooleanProperty)

If you bind, for example, an `IntegerProperty`, the type of the facade property will look like `Property<Int>` but it is infact an `IntegerProperty` under the hood. If you need to access the special functions provided by `IntegerProperty`, you will have to cast the bind result:

```
val age = bind(Person::ageProperty) as IntegerProperty
```

Similarily, you can expose a read only property by specifying a read only type:

```
val age = bind(Person::ageProperty) as ReadOnlyIntegerProperty
```

The reason for this is an unfortunate shortcoming on the type system that prevents the compiler from differentiating between overloaded `bind` functions for these specific types, so the single `bind` function inside `ViewModel` inspects the property type and returns the best match, but unfortunately the return type signature has to be `Property<T>` for now.

Rebinding

As you saw in the `TableView` example above, it is possible to change the domain object that is wrapped by the `ViewModel`. This test case sheds some more light on that:

```
@Test fun swap_source_object() {
    val person1 = Person("Person 1")
    val person2 = Person("Person 2")

    val model = PersonModel(person1)
    assertEquals(model.name, "Person 1")

    model.rebind { person = person2 }
    assertEquals(model.name, "Person 2")
}
```

The test creates two `Person` objects and a `ViewModel`. The model is initialised with the first person object. It then checks that `model.name` corresponds to the name in `person1`. Now something weird happens:

```
model.rebind { person = person2 }
```

The code inside the `rebind()` block above will be executed and all the properties of the model are updated with values from the new source object. This is actually analogous to writing:

```
model.person = person2
model.rebind()
```

The form you choose is up to you, but the first form makes sure you do not forget to call rebind. After `rebind` is called, the model is not dirty and all values will reflect the ones form the new source object or source objects. It's important to note that you can pass multiple source objects to a view model and update all or some of them as you see fit.

Rebind Listener

Our `TableView` example called the `rebindOnChange()` function and passed in a `TableView` as the first argument. This made sure that rebind was called whenever the selection of the `TableView` changed. This is actually just a shortcut to another function with the same name that takes an observable and calls rebind whenever that observable changes. If you call this function, you do not need to call rebind manually as long as you have an observable that represent the state change that should cause the model to rebind.

As you saw, `TableView` has a shorthand support for the `selectionModel.selectedItemProperty`. If not for this shorthand function call, you would have to write it like this:

```
model.rebindOnChange(table.selectionModel.selectedItemProperty()) {
    person = it ?: Person()
}
```

The above example is included to clarify how the `rebindOnChange()` function works under the hood. For real use cases involving a `TableView`, you should opt for the shorter version or use the `ItemViewModel`.

ItemViewModel

When working with the `ViewModel` you will notice some repetitive and somewhat verbose tasks. They include calling `rebind` or configuring `rebindOnChange` to change the source object. The `ItemViewModel` is an extension to the `ViewModel` and in almost all use cases you would want to inherit from this instead of the `ViewModel` class.

The `ItemViewModel` has a property called `itemProperty` of the specified type, so our `PersonModel` would now look like:

```
class PersonModel : ItemViewModel<Person>() {
    val name = bind(Person::nameProperty)
    val title = bind(Person::titleProperty)
}
```

You will notice we no longer need to pass in the `var person: Person` in the constructor. The `ItemViewModel` now has an observable property called `itemProperty` and getters/setters via the `item` property. Whenever you assign something to `item` or via `itemProperty.value`, the model is automatically rebound for you. There is also an observable `empty` boolean value you can use to check if the `ItemViewModel` is currently holding a `Person`.

The binding expressions need to take into account that it might not represent any item at the time of binding. That is why the binding expressions above now use the null safe operator.

We just got rid of some boiler plate, but the `ItemViewModel` gives us a lot more functionality. Remember how we bound the selected person from the `TableView` to our model earlier?

```
// Update the person inside the view model on selection change
model.rebindOnChange(this) { selectedPerson ->
    person = selectedPerson ?: Person()
}
```

Using the `ItemViewModel` this can be rewritten:

```
// Update the person inside the view model on selection change
bindSelected(model)
```

This will effectively attach the listener we had to write manually before and make sure that the `TableView` selection is visible in the model.

The `save()` function will now also be slightly different, since there is no `person` property in our model:

```
private fun save() {
    model.commit()
    val person = model.item
    println("Saving ${person.name} / ${person.title}")
}
```

The person is extracted from the `itemProperty` using the `item` getter.

When working with the ItemViewModel() and POJO's starting at 1.7.1 you can create the bindings as follows

```
data class Person(val firstName: String, val lastName: String)

class PersonModel : ItemViewModel<Person>() {
    val firstname = bind { item?.firstName?.toProperty() }
    val lastName = bind { item?.lastName?.toProperty() }
}
```

OnCommit callback

Sometimes it's desirable to do a specific action after the model was successfully committed. The `ViewModel` offers two callbacks, `onCommit` and `onCommit(commits: List<Commit>)`, for that.

The first function `onCommit`, has no parameters and will be called after a successful commit, right before the optional `successFn` is invoked (see: `commit`).
The second function will be called in the same order and with the addition of passing a list of committed properties along.
Each `Commit` in the list, consists of the original `ObservableValue`, the `oldValue` and the `newValue`
and a property `changed`, to signal if the `oldValue` is different then the `newValue`.

Let's look at an example how we can retrieve only the changed objects and print them to `stdout`.

To find out which object changed we defined a little extension function, which will find the given property and
if it was changed will return the old and new value or null if there was no change.

```
class PersonModel : ItemViewModel<Person>() {

    val firstname = bind(Person::firstName)
    val lastName = bind(Person::lastName)

    override val onCommit(commits: List<Commit>) {
       // The println will only be called if findChanged is not null
       commits.findChanged(firstName)?.let { println("First-Name changed from ${it.fir
st} to ${it.second}")}
       commits.findChanged(lastName)?.let { println("Last-Name changed from ${it.first
} to ${it.second}")}
    }

    private fun <T> List<Commit>.findChanged(ref: Property<T>): Pair<T, T>? {
        val commit = find { it.property == ref && it.changed}
        return commit?.let { (it.newValue as T) to (it.oldValue as T) }
    }
}
```

Injectable Models

Most commonly you will not have both the `TableView` and the editor in the same `View`. We would then need to access the `ViewModel` from at least two different views, one for the `TableView` and one for the form. Luckily, the `ViewModel` is injectable, so we can rewrite our editor example and split the two views:

```
class PersonList : View("Person List") {
    val persons = listOf(Person("John", "Manager"), Person("Jay", "Worker bee")).obser
vable()
    val model : PersonModel by inject()

    override val root = tableview(persons) {
        title = "Person"
        column("Name", Person::nameProperty)
        column("Title", Person::titleProperty)
        bindSelected(model)
    }
}
```

The person `TableView` now becomes a lot cleaner and easier to reason with. In a real application the list of persons would probably come from a controller or a remoting call though. The model is simply injected into the `View`, and we will do the same for the editor:

```
class PersonEditor : View("Person Editor") {
    val model : PersonModel by inject()

    override val root = form {
        fieldset("Edit person") {
            field("Name") {
                textfield(model.name)
            }
            field("Title") {
                textfield(model.title)
            }
           button("Save") {
                enableWhen(model.dirty)
                action {
                    save()
                }
            }
            button("Reset").action {
                model.rollback()
            }
        }
    }

    private fun save() {
        model.commit()
        println("Saving ${model.item.name} / ${model.item.title}")
    }
}
```

The injected instance of the model will be the exact same one in both views. Again, in a real application the save call would probably be offloaded
to a controller asynchronously.

When to Use `ViewModel` vs `ItemViewModel`

This chapter has progressed from the low-level implementation `ViewModel` into a streamlined `ItemViewModel`. You might wonder if there are any use cases for inheriting from `ViewModel` instead of `ItemViewModel` at all. The answer is that while you would typically extend `ItemViewModel` more than 90% of the time, there are some use cases where it does not make sense. Since ViewModels can be injected and used to keep navigational state and overall UI state, you might use it for situations where you do not have a single domain object

- you could have multiple domain objects or just a collection of loose properties. In this use case the `ItemViewModel` does not make any sense, and you might implement the `ViewModel` directly. For common cases though, `ItemViewModel` is your best friend.

> There is one potential issue with this approach. If we want to display multiple "pairs" of lists and forms, perhaps in different windows, we need a way to separate and bind the model belonging to a spesific pair of list and form. There are many ways to deal with that, but one tool very well suited for this is the scopes. Check out the scope documentation for more information about this approach.

Validation

Almost every application needs to check that the input supplied by the user conforms to a set of rules or are otherwise acceptable. TornadoFX sports an extensible validation and decoration framework.

We will first look at validation as a standalone feature before we integrate it with the `ViewModel`.

Under the Hood

The following explanation is a bit verbose and does not reflect the way you would write validation code in your application. This section will provide you with a solid understanding of how validation works and how the individual pieces fit together.

Validator

A `Validator` knows how to inspect user input of a specified type and will return a `ValidationMessage` with a `ValidationSeverity` describing how the input compares to the expected input for a specific control. If a `Validator` deems that there is nothing to report for an input value, it returns `null`. A text message can optionally accompany the `ValidationMessage`, and would normally be displayed by the `Decorator` configured in the `ValidationContext`. We will cover more on decorators later.

The following severity levels are supported:

- `Error` - Input was not accepted
- `Warning` - Input is not ideal, but accepted
- `Success` - Input is accepted
- `Info` - Input is accepted

There are multiple severity levels representing successful input to easier provide the contextually correct feedback in most cases. For example, you might want to give an informational message for a field no matter the input value, or specifically mark fields with a green checkbox when they are entered. The only severity that will result in an invalid status is the `Error` level.

ValidationTrigger

By default validation will happen when the input value changes. The input value is always an `ObservableValue<T>`, and the default trigger simply listens for changes. You can however choose
to validate when the input field looses focus, or when a save button is clicked for instance. The following ValidationTriggers can be configured for each validator:

- `OnChange` - Validate when input value changes, optionally after a given delay in milliseconds
- `OnBlur` - Validate when the input field looses focus
- `Never` - Only validate when `ValidationContext.validate()` is called

ValidationContext

Normally you would validate user input from multiple controls or input fields at once. You can gather these validators in a `ValidationContext` so you can check if all validators are valid, or ask the validation context to perform validation for all fields at any given time. The context also controls what kind of decorator will be used to convey the validation message for each field. See the Ad Hoc validation example below.

Decorator

The `decorationProvider` of a `ValidationContext` is in charge of providing feedback when a `ValidationMessage` is associated with an input. By default this is an instance of `SimpleMessageDecorator` which will mark the input field with a colored triangle in the topper left corner and display a popup with the message while the input has focus.

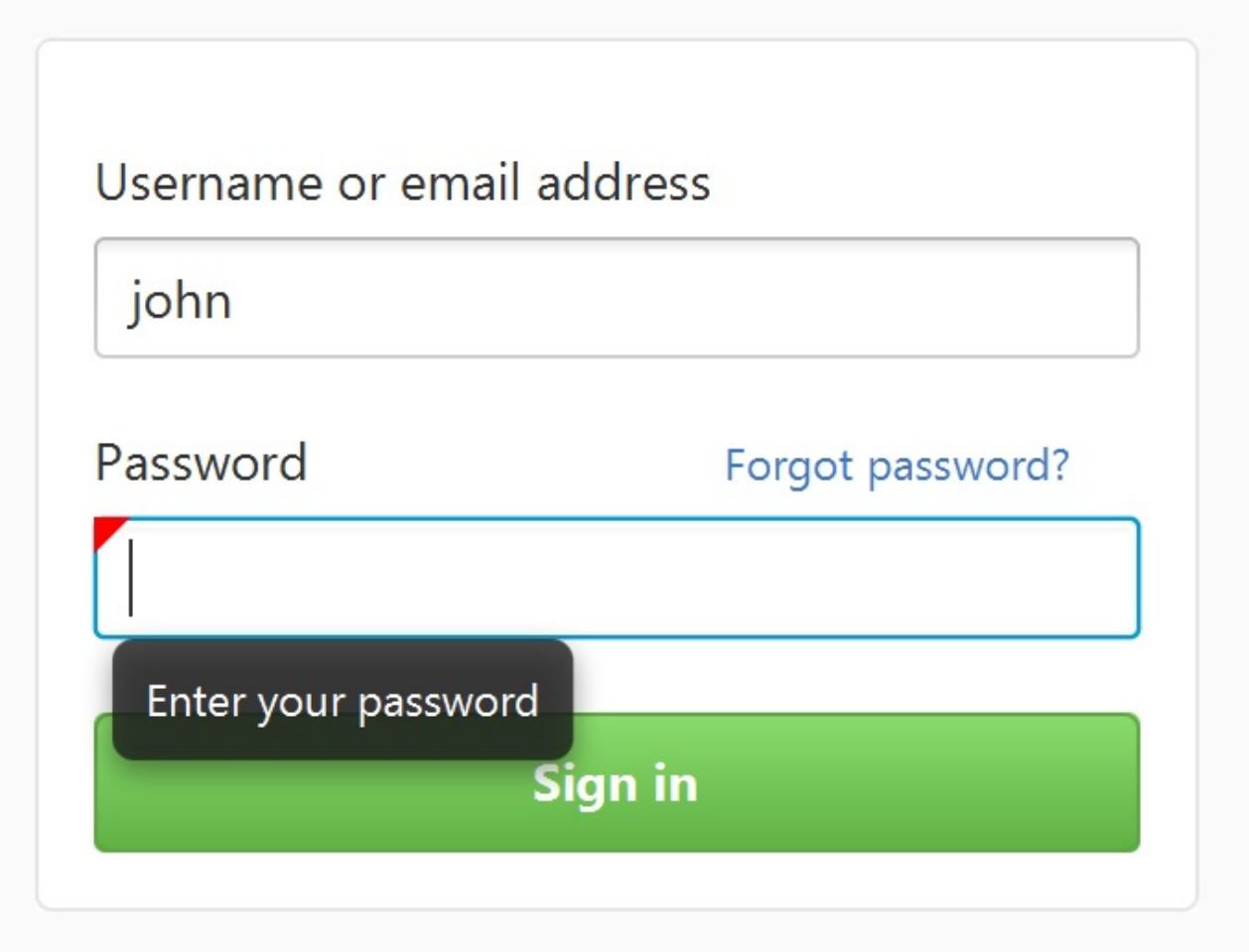


Figure 11.2 The default decorator showing a required field validation message

If you don't like the default decorator look you can easily create your own by implementing the `Decorator` interface:

```
interface Decorator {
    fun decorate(node: Node)
    fun undecorate(node: Node)
}
```

You can assign your decorator to a given `ValidationContext` like this:

```
context.decorationProvider = MyDecorator()
```

Tip: You can create a decorator that applies CSS style classes to your inputs instead of overlaying other nodes to provide feedback.

Ad Hoc Validation

While you will probably never do this in a real application, it is possible to set up a `ValidationContext` and apply validators to it manually. The following example is actually taken from the internal tests of the framework. It illustrates the concept, but is not a practical pattern in an application.

```
// Create a validation context
val context = ValidationContext()

// Create a TextField we can attach validation to
val input = TextField()

// Define a validator that accepts input longer than 5 chars
val validator = context.addValidator(input, input.textProperty()) {
    if (it!!.length < 5) error("Too short") else null
}

// Simulate user input
input.text = "abc"

// Validation should fail
assertFalse(validator.validate())

// Extract the validation result
val result = validator.result

// The severity should be error
assertTrue(result is ValidationMessage && result.severity == ValidationSeverity.Error)

// Confirm valid input passes validation
input.text = "longvalue"
assertTrue(validator.validate())
assertNull(validator.result)
```

Take special note of the last parameter to the `addValidator` call. This is the actual validation logic. The function is passed the current input for the property it validates and must return null if there are no messages, or an instance of `ValidationMessage` if something is noteworthy about the input. A message with severity `Error` will cause the validation to fail. As you can see, you don't need to instantiate a ValidationMessage yourself, simply use one of the functions `error`, `warning`, `success` or `info` instead.

Validation with ViewModel

Every ViewModel contains a `ValidationContext`, so you don't need to instantiate one yourself. The Validation framework integrates with the type safe builders as well, and even provides some built in validators, like the `required` validator. Going back to our person editor, we can make the input fields required with this simple change:

```
field("Name") {
    textfield(model.name).required()
}
```

That's all there is to it. The required validator optionally takes a message that will be presented to the user if the validation fails. The default text is "This field is required".

Instead of using the built in `required` validator we can express the same thing manually:

```
field("Name") {
    textfield(model.name).validator {
        if (it.isNullOrBlank()) error("The name field is required") else null
    }
}
```

If you want to further customize the textfield, you might want to add another set of curly braces:

```
field("Name") {
    textfield(model.name) {
        // Manipulate the text field here
        validator {
            if (it.isNullOrBlank()) error("The name field is required") else null
        }
    }
}
```

Binding buttons to validation state

You might want to only enable certain buttons in your forms when the input is valid. The `model.valid` property can be used for this purpose. Since the default validation trigger is `OnChange`, the valid state would only be accurate when you first try to commit the model. However, if you want

to bind a button to the `valid` state of the model you can call `model.validate(decorateErrors = false)` to force all validators to report their results without
actually showing any validation errors to the user.

```
field("username") {
    textfield(username).required()
}
field("password") {
    passwordfield(password).required()
}
buttonbar {
    button("Login", ButtonBar.ButtonData.OK_DONE) {
        enableWhen(model.valid)
        action {
            model.commit {
                doLogin()
            }
        }
    }
}
// Force validators to update the `model.valid` property
model.validate(decorateErrors = false)
```

Notice how the login button's enabled state is bound to the enabled state of the model via `enableWhen { model.valid }` call. After all
the fields and validators are configured, the `model.validate(decorateErrors = false)` make sure the valid state of the model is updated
without triggering error decorations on the fields that fail validation. The decorators will kick in on value change by default, unless you
override the `trigger` parameter to `validator`. The `required()` build in validator also accepts this parameter. For example, to run the validator
only when the input field looses focus you can call
`textfield(username).required(ValidationTrigger.OnBlur)`.

Validation in dialogs

The `dialog` builder creates a window with a form and a fieldset and let's you start adding fields to it. Some times you don't have a ViewModel for such cases, but you might still want to
use the features it provides. For such situations you can instantiate a ViewModel inline and hook up one or more properties to it. Here is an example dialog that requires the user to enter some input in a textarea:

```
dialog("Add note") {
    val model = ViewModel()
    val note = model.bind { SimpleStringProperty() }

    field("Note") {
        textarea(note) {
            required()
            whenDocked { requestFocus() }
        }
    }
    buttonbar {
        button("Save note").action {
            model.commit { doSave() }
        }
    }
}
```

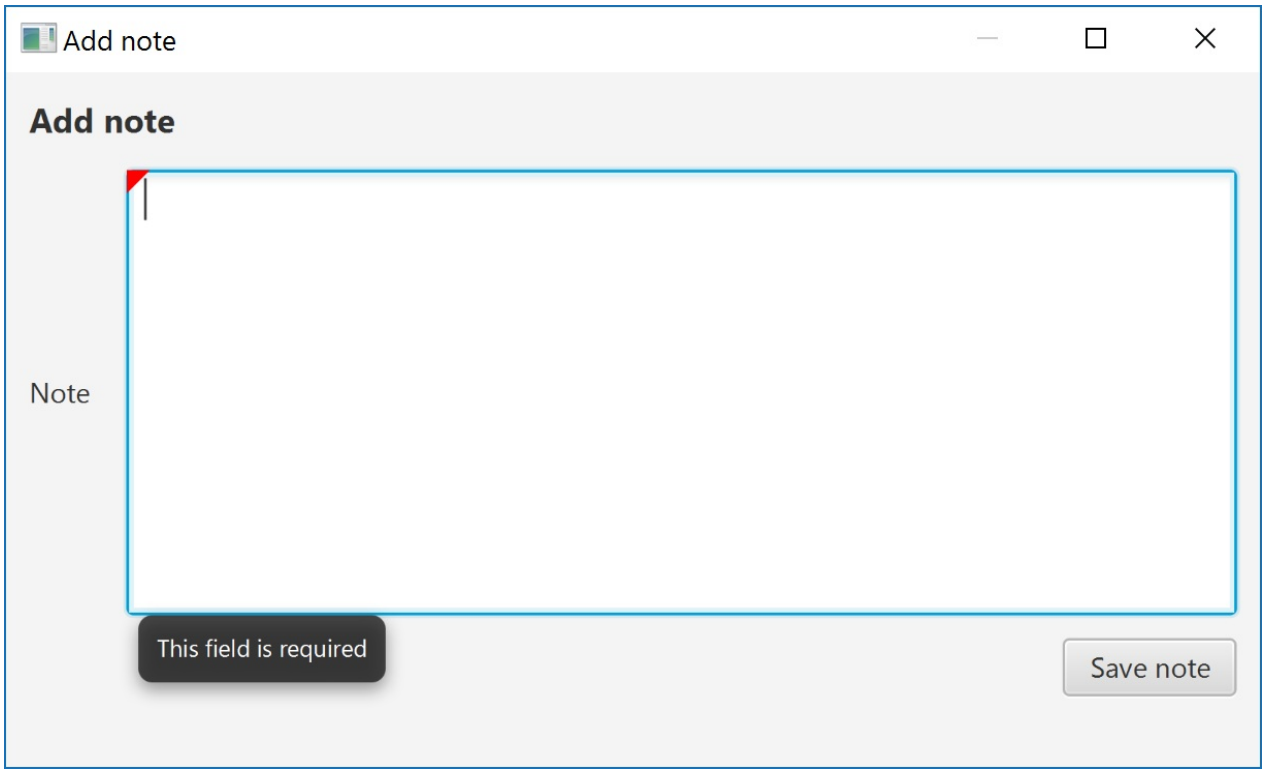


Figure 11.3 A dialog with a inline ViewModel context

Notice how the `note` property is connected to the context by specifying it's bean parameter. This is crucial for making the field validation available.

Partial commit

It's also possible to do a partial commit by suppling a list of fields you want to commit to avoid committing everything. This can be convenient in situations where you edit the same ViewModel instance from different Views, for example in a Wizard. See the Wizard chapter

for more information about partial commit, and the corresponding partial validation features.

TableViewEditModel

If you are pressed for screen real estate and do not have space for a master/detail setup with a `TableView`, an effective option is to edit the `TableView` directly. By enabling a few streamlined features in TornadoFX, you can not only enable easy cell editing but also enable dirty state tracking, committing, and rollback. By calling `enableCellEditing()` and `enableDirtyTracking()`, as well as accessing the `tableViewEditModel` property of a `TableView`, you can easily enable this functionality.

When you edit a cell, a blue flag will indicate its dirty state. Calling `rollback()` will revert dirty cells to their original values, whereas `commit()` will set the current values as the new baseline (and remove all dirty state history).

```
import tornadofx.*

class MyApp: App(MyView::class)
class MyView : View("My View") {

    val controller: CustomerController by inject()
    var tableViewEditModel: TableViewEditModel<Customer> by singleAssign()

    override val root =  borderpane {
        top = buttonbar {
            button("COMMIT").setOnAction {
                tableViewEditModel.commit()
            }
            button("ROLLBACK").setOnAction {
                tableViewEditModel.rollback()
            }
        }
        center = tableview<Customer> {

            items = controller.customers
            isEditable = true

            column("ID",Customer::idProperty)
            column("FIRST NAME", Customer::firstNameProperty).makeEditable()
            column("LAST NAME", Customer::lastNameProperty).makeEditable()

            enableCellEditing() //enables easier cell navigation/editing
            enableDirtyTracking() //flags cells that are dirty

            tableViewEditModel = editModel
        }
    }
}

class CustomerController : Controller() {
    val customers = listOf(
            Customer(1, "Marley", "John"),
            Customer(2, "Schmidt", "Ally"),
            Customer(3, "Johnson", "Eric")
    ).observable()
}

class Customer(id: Int, lastName: String, firstName: String) {
    val lastNameProperty = SimpleStringProperty(this, "lastName", lastName)
    var lastName by lastNameProperty
    val firstNameProperty = SimpleStringPorperty(this, "firstName", firstName)
    var firstName by firstNameProperty
    val idProperty = SimpleIntegerProperty(this, "id", id)
    var id by idProperty
}
```

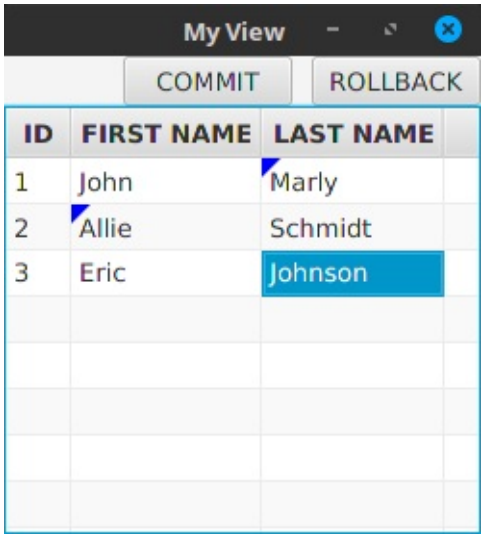


Figure 11.4 A `TableView` with dirty state tracking, with `rollback()` and `commit()` functionality.

Note also there are many other helpful properties and functions on the `TableViewEditModel`. The `items` property is an `ObservableMap<S, TableColumnDirtyState<S>>` mapping the dirty state of each record item `S`. If you want to filter out and commit only dirty records so you can persist them somewhere, you can have your "Commit" `Button` perform this action instead.

```
button("COMMIT").action {
    tableViewEditModel.items.asSequence()
            .filter { it.value.isDirty }
            .forEach {
                println("Committing ${it.key}")
                it.value.commit()
            }
}
```

There are also `commitSelected()` and `rollbackSelected()` to only commit or rollback the selected records in the `TableView`.

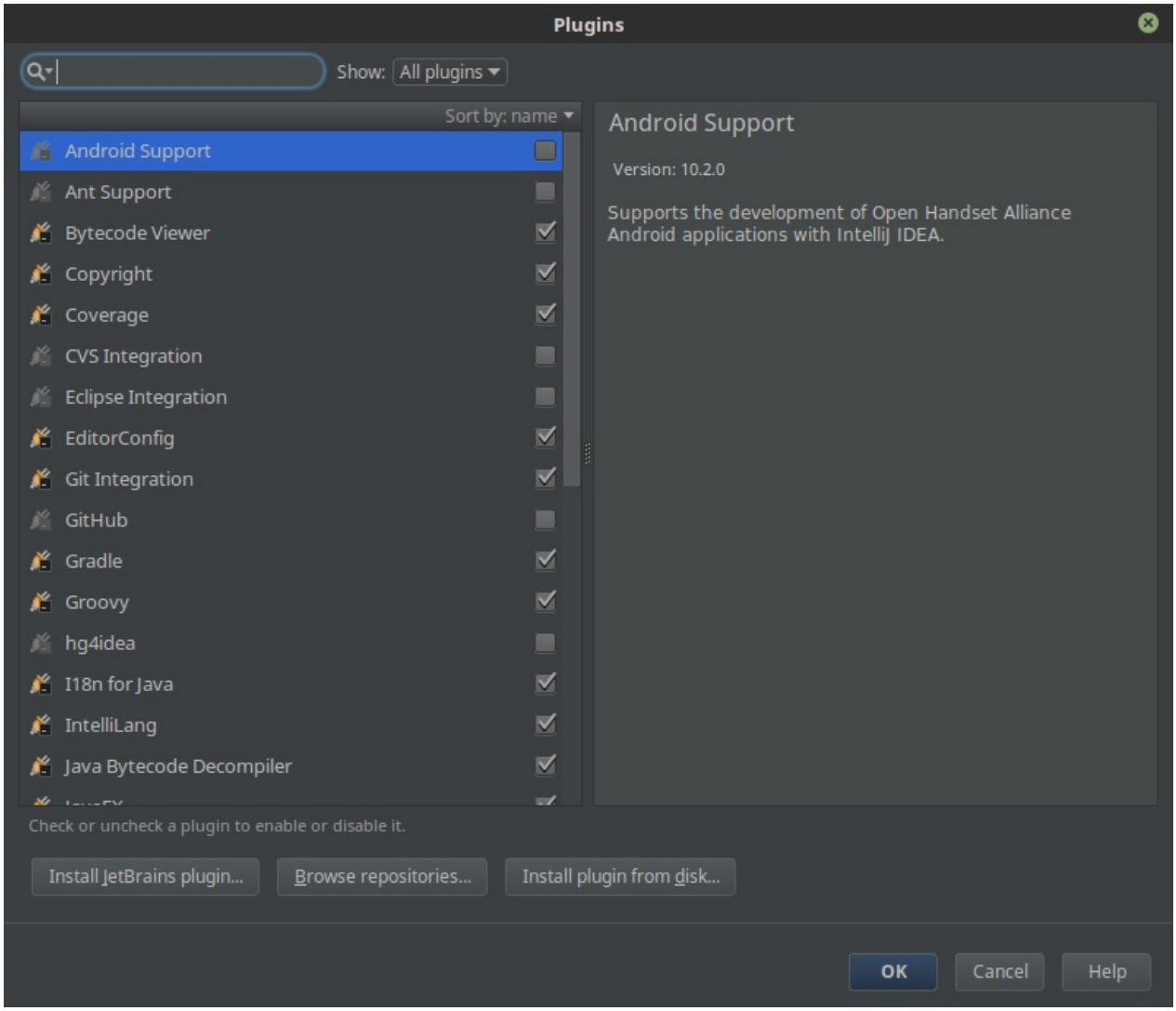
13. TornadoFX IDEA Plugin

To save time in using TornadoFX, you can install a convenient Intellij IDEA plugin to automatically generate project templates, Views, injections, and other TornadoFX features. Of course, you do not have to use this plugin which was done throughout this book. But it adds some convenience to build TornadoFX applications a little more quickly.

Installing the Plugin

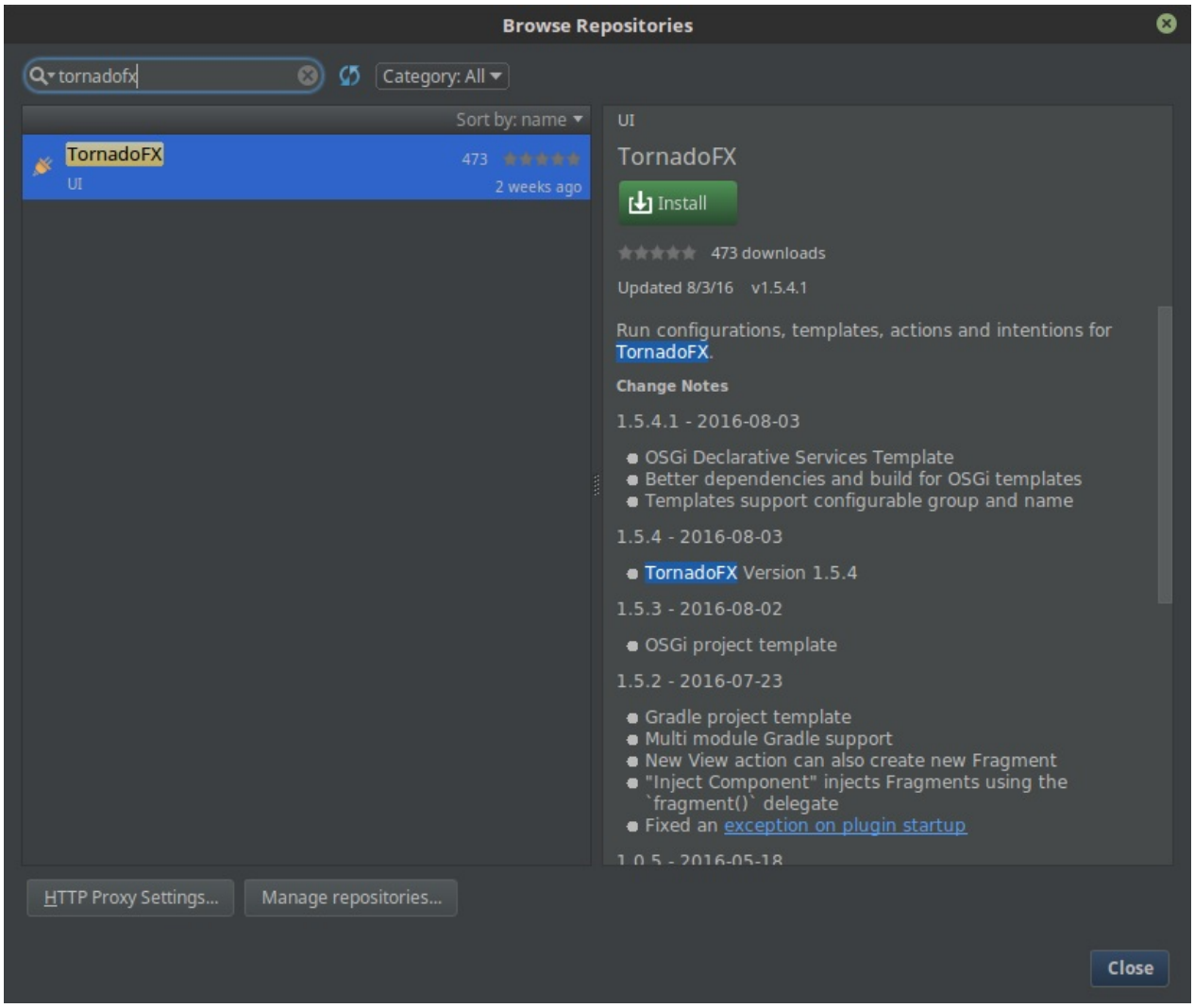
In the Intellij IDEA workspace, press `CONTROL` + `SHIFT` + `A` and type "Plugins", then press `ENTER`. You will see a dialog to search and install plugins. Click the *Browse Repositories* button (Figure 13.1).

Figure 13.1 After bringing up the *Plugins* dialog, click *Browse Repositories*.



You will then see a list of 3rd party plugins available to install. Search for "TornadoFX", select it, and click the green *Install* button (Figure 13.2).

Figure 13.2 Search for "TornadoFX" and click *Install*



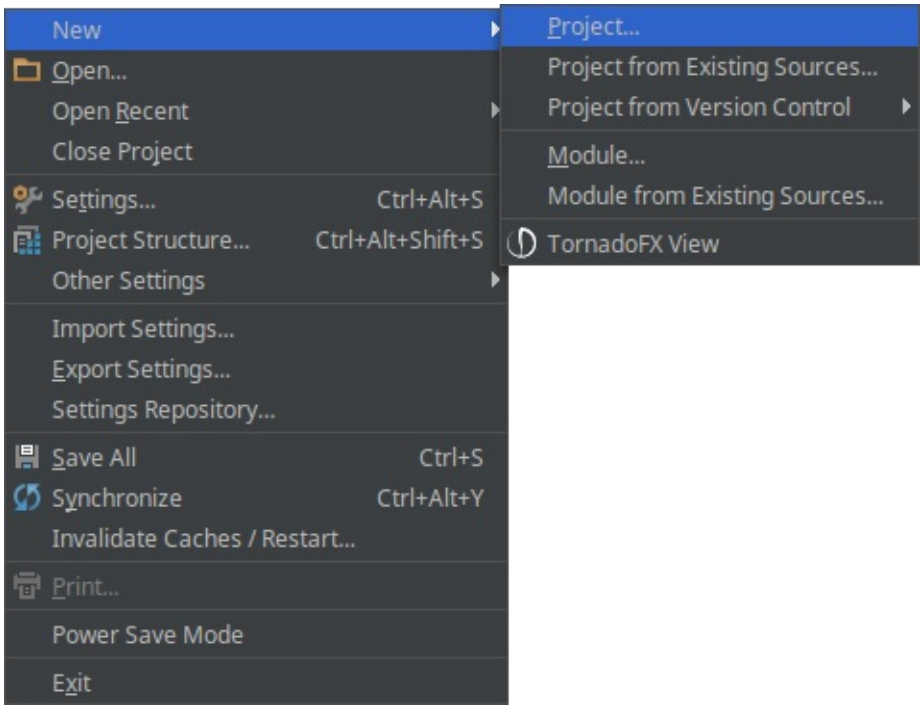
Wait for it to finish installing and the restart Intellij IDEA.

TornadoFX Project Templates

The TornadoFX plugins has some Maven and Gradle project templates to quickly create a configured TornadoFX application.

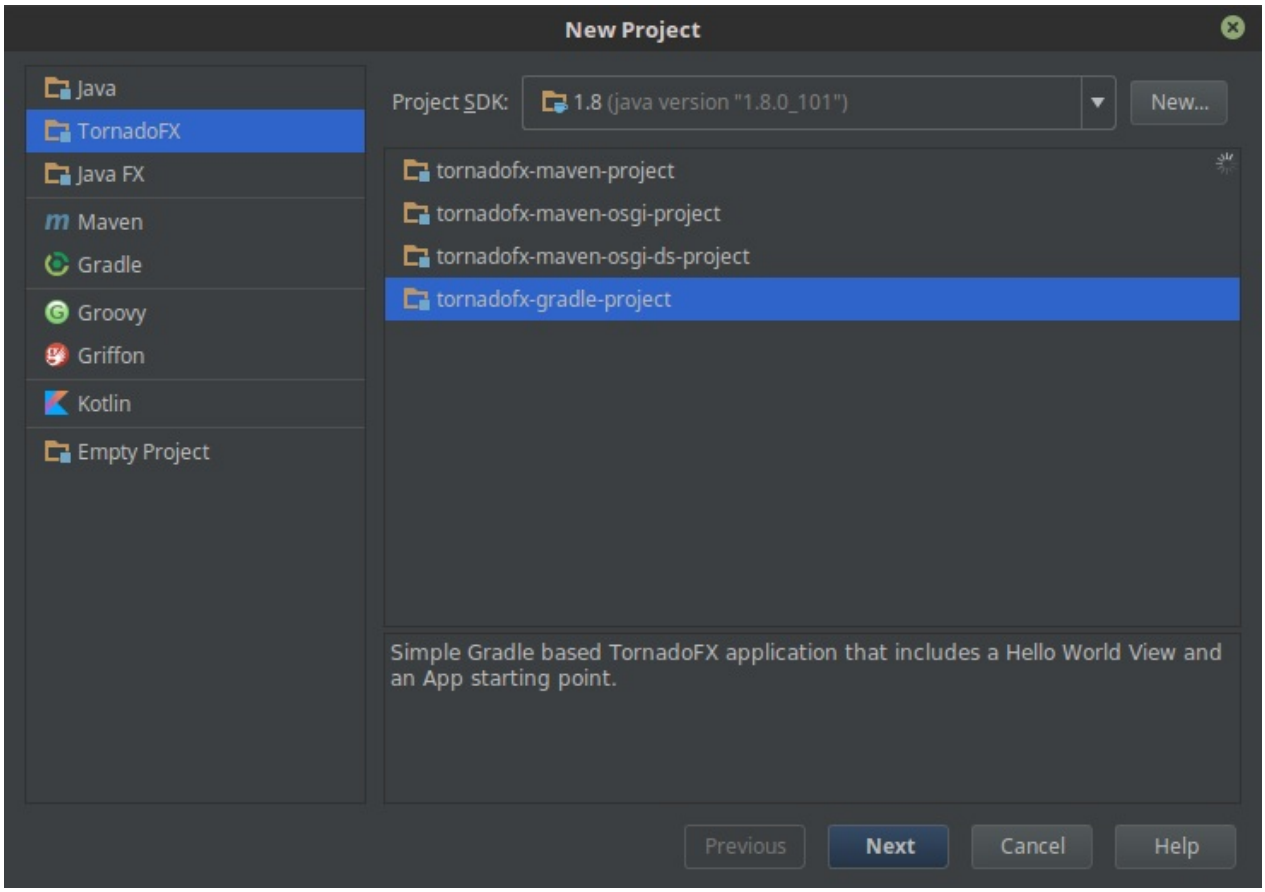
In Intellij IDEA, navigate to *File -> New -> Project...* (Figure 13.3).

Figure 13.3



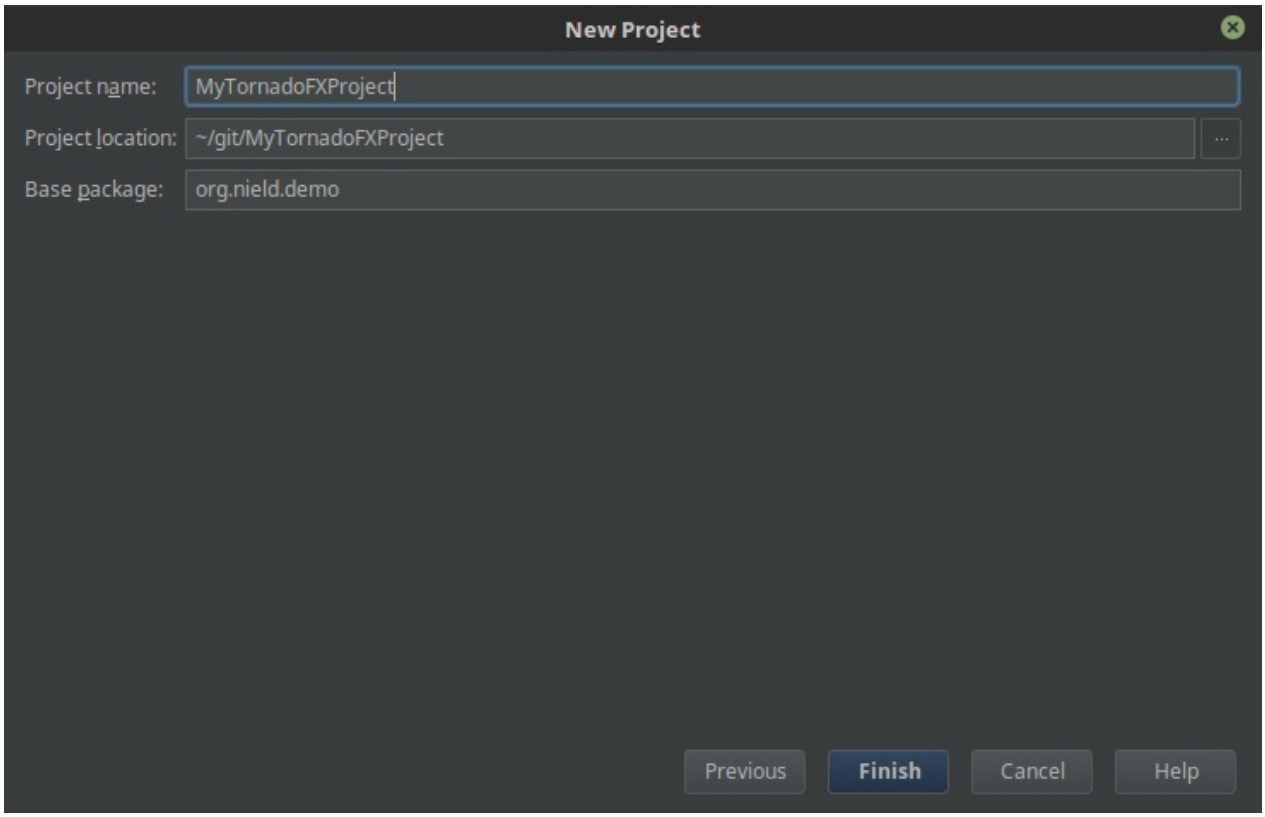
You will then see a dialog to create a new TornadoFX project. You can create Gradle and Maven flavors, with or without OSGi support. Let's create a Gradle one for demonstration (Figure 13.4).

Figure 13.4



In the next dialog, give your project a name, a location folder, and a base package with your domain (Figure 13.5). Then click *Finish*.

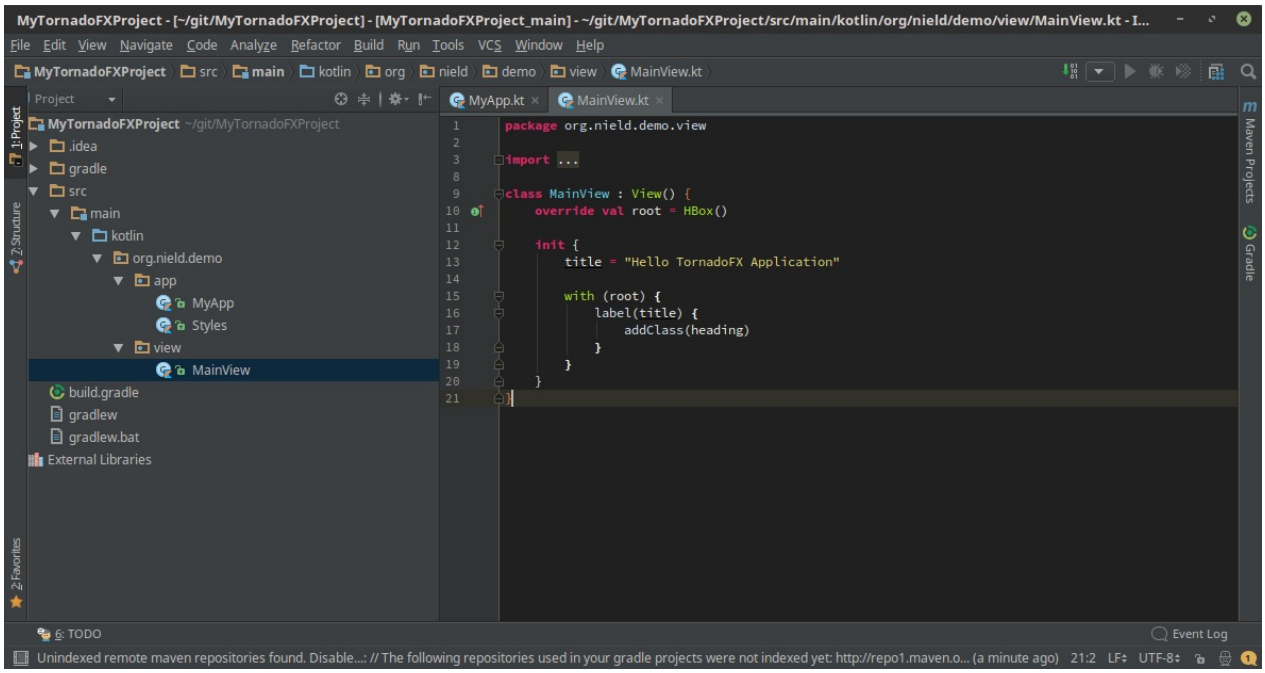
Figure 13.5



You may be prompted to import the project as a Gradle project, and click on that prompt if you encounter it. You will then have a TornadoFX application configured and set up, including `App`, `View`, and `Styles` entities set up (Figure 13.6).

Figure 13.6

A generated TornadoFX project with a Gradle configuration.

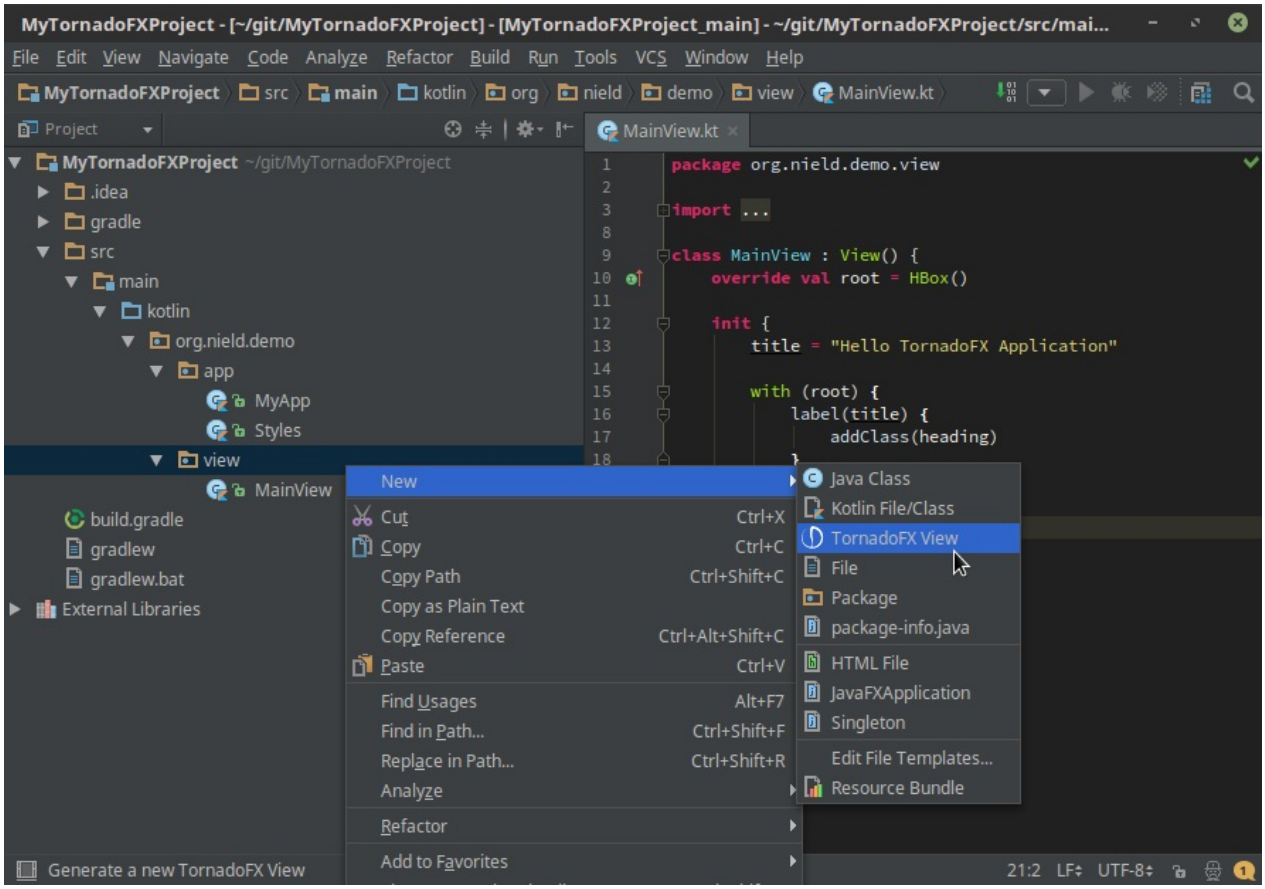


These steps apply to the Maven and OSGi wizards as well, and do not forget to put your project on a version tracking system like GIT!.

Creating Views

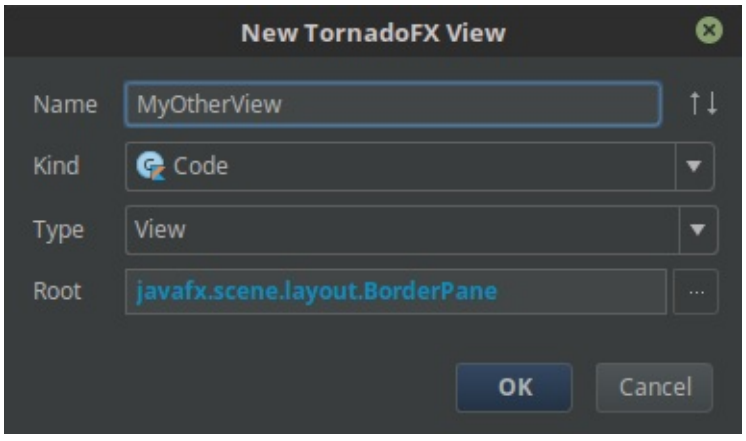
You can create Views, Fragments, and FXML files quickly with the plugin. You can right click a folder in the Project, then navigate the popup menu to *New -> TornadoFX View* (Figure 13.7).

Figure 13.7



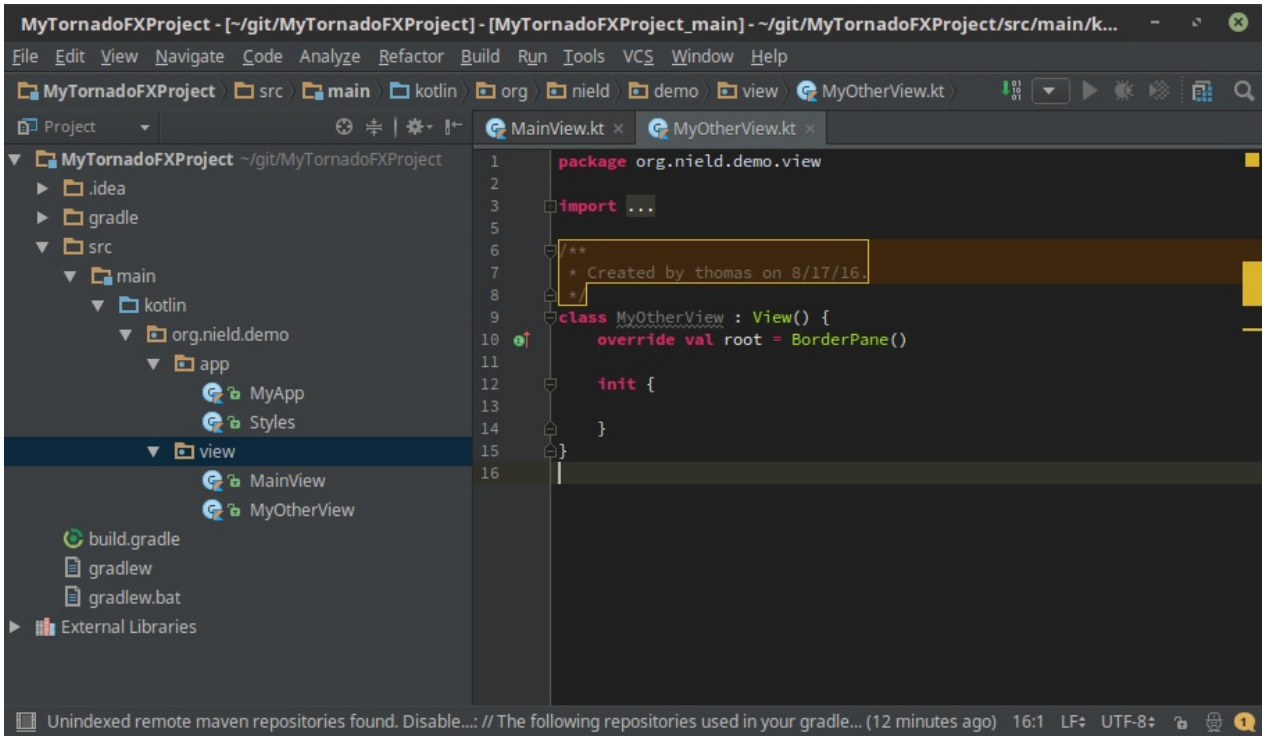
You will then come to a dialog to dictate how the `View` is constructed. You even have the option of specifying it as a `Fragment` instead through the *Type* parameter, as well as an FXML via *Kind*. Finally, you can specify the `Node` type for the *Root*, which should default to a `BorderPane`.

Figure 13.8



Click *OK* and a new `View` will generated and added to your project (Figure 13.9).

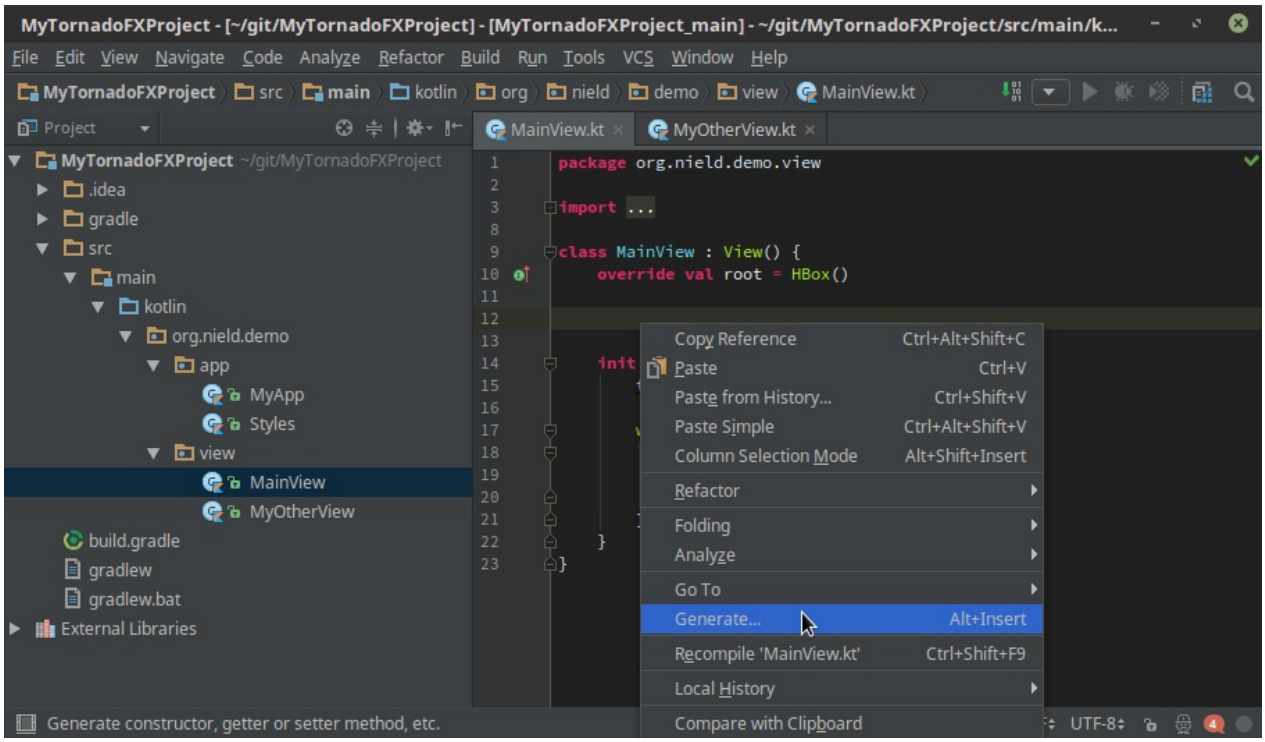
Figure 13.9 A new `View` generated with the TornadoFX plugin

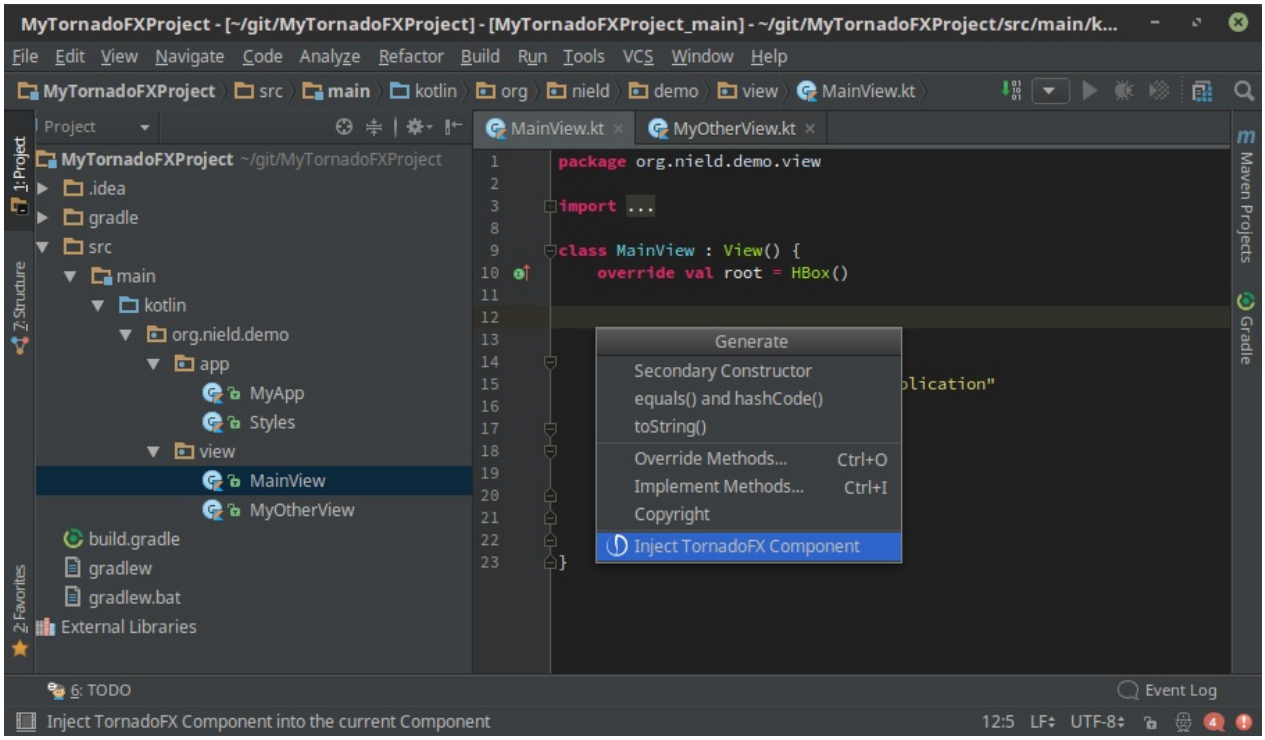


Injecting Components

One last minor convenience. You can generate TornadoFX `Component` injections quickly with the plugin. For instance, if you right click the class body of the `MainView`, you can generate the `MyOtherView` as an injected property (Figure 13.10).

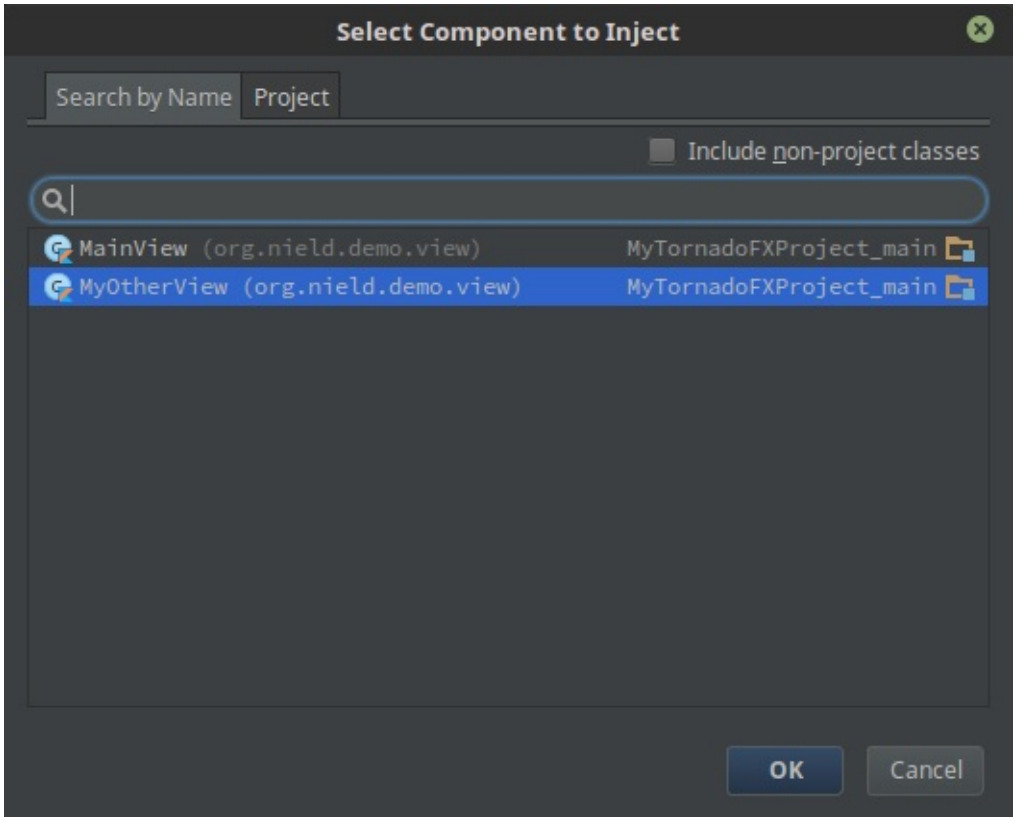
Figure 13.10

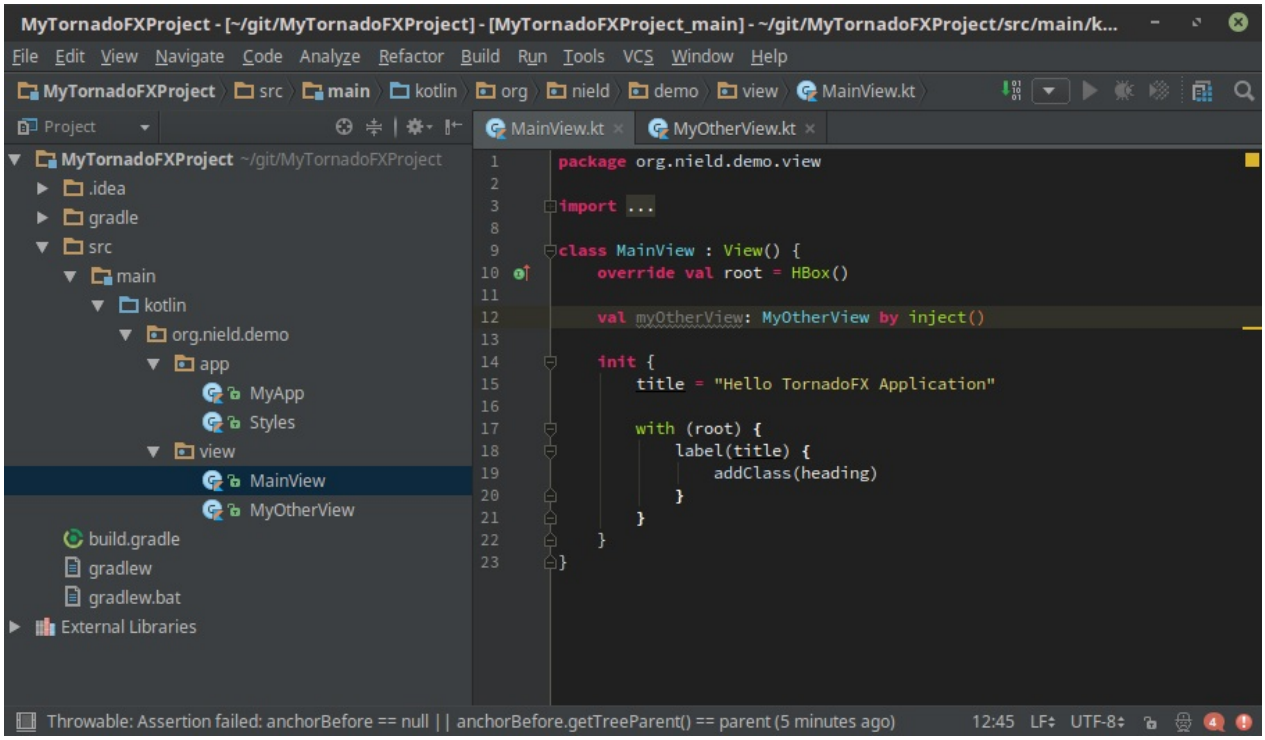




You can then use a dialog to select the `MyOtherView` as the injected property, then click *OK* (Figure 13.11).

Figure 13.11





Generating TornadoFX Properties

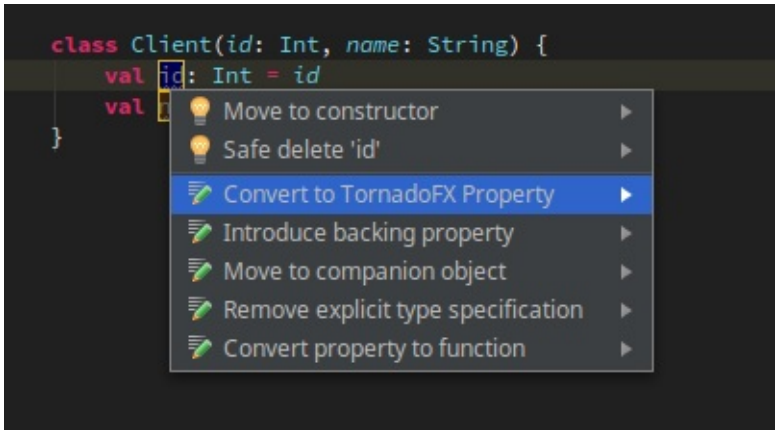
One of the most helpful features in the plugin is the ability to convert plain Kotlin properties into TornadoFX properties.

Say you have a simple domain class called `Client`.

```
class Client(id: Int, name: String) {
    val id: Int = id
    val name: String = name
}
```

If you click on a property and then the intent lightbulb, or press ALT+ENTER, you should see a menu popup with an option to convert it to a TornadoFX Property (Figure 13.12).

Figure 13.12



Do this for each property and your `Client` class should now look like this.

```
class Client(id: Int, name: String) {
    var id by property(id)
    fun idProperty() = getProperty(Client::id)

    var name by property(name)
    fun nameProperty() = getProperty(Client::name)
}
```

Your `Client` now uses JavaFX properties instead of plain properties. Notice the primary constructor will pass the intial values to the `property()` delegates, but you do not have to provide initial values if they are not desired.

This is a time-saving feature when creating domain types for data controls. Next we will cover how to generate `TableView` columns.

Generating Columns for a TableView

Another handy feature you can do with the plugin also is generating columns for a `TableView`. If you have a `TableView<Person>`, you can put the cursor on its declaration, press ALT + ENTER, and get a prompt to generate the columns (Figure 13.13).

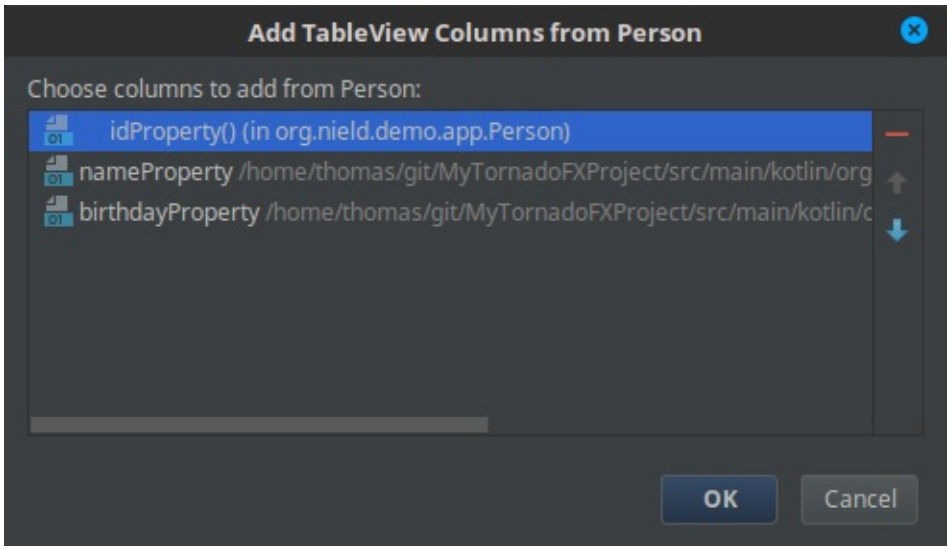
Figure 13.13

```
4   import javafx.beans.property.SimpleObjectProperty
5   import javafx.beans.property.SimpleStringProperty
6   import tornadofx.*
7   import java.time.LocalDate
8   import java.time.Period
9
10  class MyApp: App(MyView::class)
11
12  class MyView : View("My View") {
13
14      override val root = tableview<Person> {
15          isEditable = true
16          items = listOf(Person(1                    ,18))).observable()
17
18      }
19  }
20
21
22  class Person(id: Int, name: String, birthday: LocalDate) {
23      var id by property<Int>(id)
24      fun idProperty() = getProperty(Person::id)
25
26      val nameProperty = SimpleStringProperty(name)
27      var name by nameProperty
28
29      val birthdayProperty = SimpleObjectProperty(birthday)
30      var birthday by birthdayProperty
31
32      val age: Int get() = Period.between(birthday, LocalDate.now()).years
33  }
```

Run 'MyView' with Coverage
Run 'MyView' Ctrl+Shift+F10
Add TableView Columns...
Convert property initializer to getter
Convert to TornadoFX Property

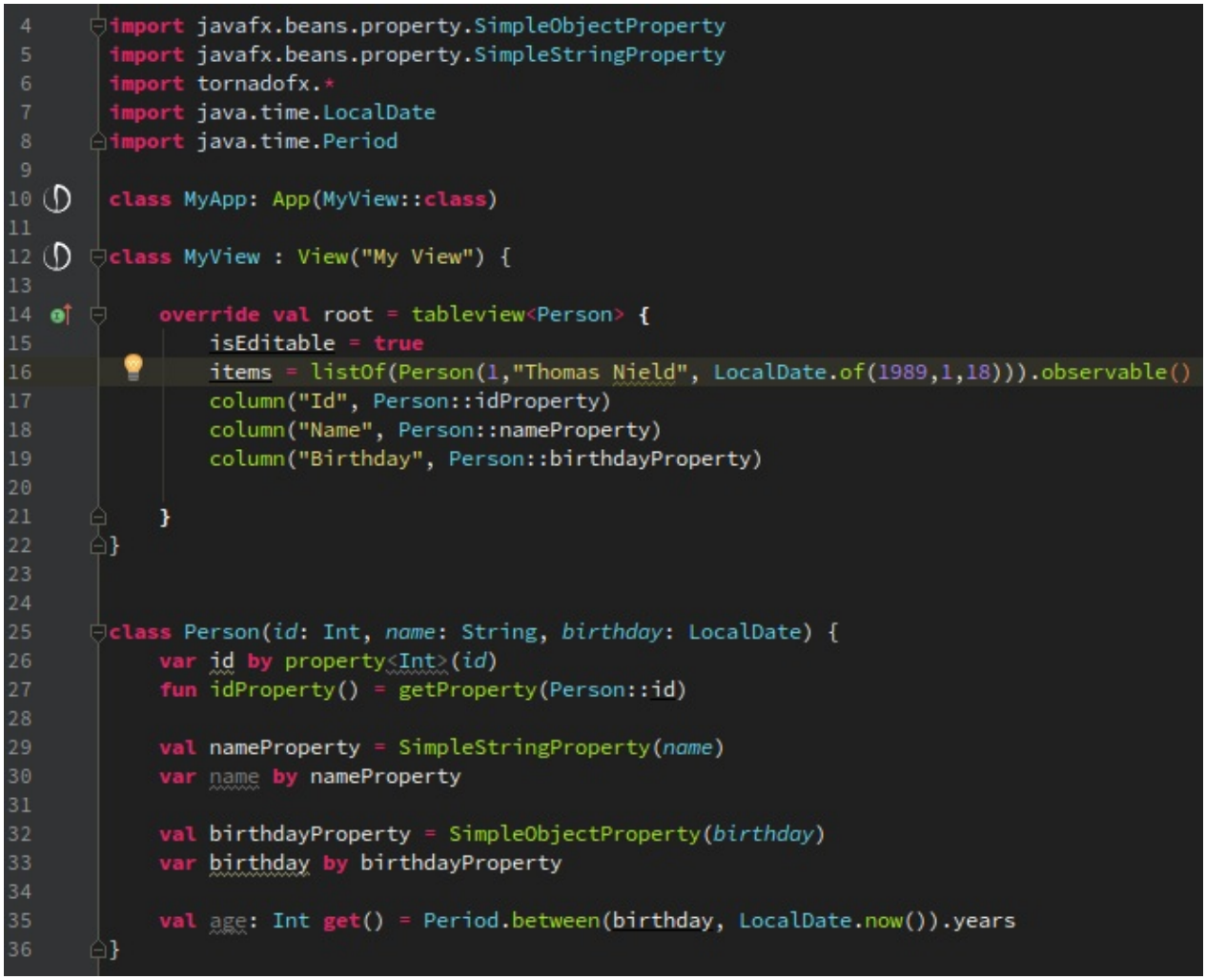
You will then see a dialog to confirm which `Person` properties to generate the columns on (Figure 14.14).

Figure 13.14



Press "OK" and the columns will then be generated for you (Figure 13.15).

Figure 13.15



```
import javafx.beans.property.SimpleObjectProperty
import javafx.beans.property.SimpleStringProperty
import tornadofx.*
import java.time.LocalDate
import java.time.Period

class MyApp: App(MyView::class)

class MyView : View("My View") {

    override val root = tableview<Person> {
        isEditable = true
        items = listOf(Person(1,"Thomas Nield", LocalDate.of(1989,1,18))).observable()
        column("Id", Person::idProperty)
        column("Name", Person::nameProperty)
        column("Birthday", Person::birthdayProperty)

    }
}


class Person(id: Int, name: String, birthday: LocalDate) {
    var id by property<Int>(id)
    fun idProperty() = getProperty(Person::id)

    val nameProperty = SimpleStringProperty(name)
    var name by nameProperty

    val birthdayProperty = SimpleObjectProperty(birthday)
    var birthday by birthdayProperty

    val age: Int get() = Period.between(birthday, LocalDate.now()).years
}
```

Note that at the time of writing this guide, for a given `TableView<T>`, this feature only works if the properties on `T` follow the JavaFX convention using the `Property` delgates.

Summary

The TornadoFX plugin has some time-saving conveniences that you are welcome to take advantage of. Of course, you do not have to use the plugin because it merely provides shortcuts and generates code. In time, there may be more features added to the plugin so be sure to follow the project on GitHub for future developments.

Part 2: Advanced Features

This section moves beyond the core features of TornadoFX, and showcases advanced features and specific framework capabilities. These chapters are not meant to be read sequentially, but rather cherrypicked for your specific interests and needs.

Property Delegates

Kotlin is packed with great language features, and delegated properties are a powerful way to specify how a property works and create re-usable policies for those properties. On top of the ones that exist in Kotlin's standard library, TornadoFX provides a few more property delegates that are particularly helpful for JavaFX development.

Single Assign

It is often ideal to initialize properties immediately upon construction. But inevitably there are times when this simply is not feasible. When a property needs to delay its initialization until it is first called, a lazy delegate is typically used. You specify a lambda instructing how the property value is initialized when its getter is called the first time.

```
val fooValue by lazy { buildExpensiveFoo() }
```

But there are situations where the property needs to be assigned later not by a value-supplying lambda, but rather some external entity at a later time. When we leverage type-safe builders we may want to save a `Button` to a class-level property so we can reference it later. If we do not want `myButton` to be nullable, we need to use the `lateinit` modifier.

```
class MyView: View() {

    lateinit var myButton: Button

    override val root = vbox {
        myButton = button("New Entry")
    }
}
```

The problem with `lateinit` is it can be assigned multiple times by accident, and it is not necessarily thread safe. This can lead to classic bugs associated with mutability, and you really should strive for immutability as much as possible (*Effective Java* by Bloch, Item #13).

By leveraging the `singleAssign()` delegate, you can guarantee that property is *only* assigned once. Any subsequent assignment attempts will throw a runtime error, and so will accessing it before a value is assigned. This effectively gives us the guarantee of immutability, although it is enforced at runtime rather than compile time.

```
class MyView: View() {

    var myButton: Button by singleAssign()

    override val root = vbox {
        myButton = button("New Entry")
    }
}
```

Even though this single assignment is not enforced at compile time, infractions can be captured early in the development process. Especially as complex builder designs evolve and variable assignments move around, `singleAssign()` is an effective tool to mitigate mutability problems and allow flexible timing for property assignments.

By default, `singleAssign()` synchronizes access to its internal value. You should leave it this way especially if your application is multithreaded. If you wish to disable synchronization for whatever reason, you can pass a `SingleAssignThreadSafetyMode.NONE` value for the policy.

```
var myButton: Button by singleAssign(SingleAssignThreadSafetyMode.NONE)
```

JavaFX Property Delegate

Do not confuse the JavaFX `Property` with a standard Java/Kotlin "property". The `Property` is a special type in `JavaFX` that maintains a value internally and notifies listeners of its changes. It is proprietary to JavaFX because it supports binding operations, and will notify the UI when it changes. The `Property` is a core feature of JavaFX and has its own JavaBeans-like pattern.

This pattern is pretty verbose however, and even with Kotlin's syntax efficiencies it still is pretty verbose. You have to declare the traditional getter/setter as well as the `Property` item itself.

```
 class Bar {
    private val fooProperty by lazy { SimpleObjectProperty<T>() }
    fun fooProperty() = fooProperty
    var foo: T
        get() = fooProperty.get()
        set(value) = fooProperty.set(value)
}
```

Fortunately, TornadoFX can abstract most of this away. By delegating a Kotlin property to a JavaFX `property()`, TornadoFX will get/set that value against a new `Property` instance. To follow JavaFX's convention and provide the `Property` object to UI components, you can

create a function that fetches the `Property` from TornadoFX and returns it.

```
class Bar {
    var foo by property<String>()
    fun fooProperty() = getProperty(Bar::foo)
}
```

Especially as you start working with `TableView` and other complex controls, you will likely find this pattern helpful when creating model classes, and this pattern is used in several places throughout this book.

Note you do not have to specify the generic type if you have an initial value to provide to the property. In the below example, it will infer the type as `String.

```
class Bar {
    var foo by property("baz")
    fun fooProperty() = getProperty(Bar::foo)
}
```

Alternative Property Syntax

There is also an alternative syntax which produces almost the same result:

```
import tornadofx.getValue
import tornadofx.setValue

class Bar {
    val fooProperty = SimpleStringProperty()
    var foo by fooProperty
}
```

Here you define the JavaFX property manually and delegate the getters and setters directly from the property. This might look cleaner to you, and so you are free to choose whatever syntax you are most comfortable with. However, the first alternative creates a JavaFX compliant property in that it exposes the `Property` via a function called `fooProperty()`, while the latter simply exposes a variable called `fooProperty`. For TornadoFX there is no difference, but if you interact with legacy libraries that require a property function you might need to stick with the first one.

Null safety of Properties

By default properties will have a Platform Type with uncertain nullability and completely ignore the null safety of Kotlin:

```
class Bar {
    var foo by property<String>()
    fun fooProperty() = getProperty(Bar::foo)

    val bazProperty = SimpleStringProperty()
    var baz by bazProperty

    init {
        foo = null
        foo.length // Will throw NPE during runtime

        baz = null
        baz.length // Will throw NPE during runtime
    }
}
```

To remedy this you can set the type of your property on the `var` (not on the Property-Object itself!). But keep in mind to set a default value on the property object when you set the var to be nullable or you will get an NPE anyways:

```
class Bar {
    var foo:String by property<String>("") // Non-nullable String with default value
    fun fooProperty() = getProperty(Bar::foo)

    val bazProperty = SimpleStringProperty() // No default needed
    var baz: String? by bazProperty // Nullable String

    init {
        foo = null // Will no longer compile
        foo.length

        baz = null
        baz.length // Will no longer compile
    }
}
```

FXML Delegate

If you have a given `MyView` View with a neighboring FXML file `MyView.fxml` defining the layout, the `fxid()` property delegate will retrieve the control defined in the FXML file. The control must have an `fx:id` that is the same name as the variable.

```
<Label fx:id="counterLabel">
```

Now we can inject this `Label` into our `View` class:

```
val counterLabel : Label by fxid()
```

Otherwise, the ID must be specifically passed to the delegate call.

```
val myLabel : Label by fxid("counterLabel")
```

Please read Chapter 10 to learn more about FXML.

Advanced Data Controls

This section will primarily address more advanced features you can leverage with data controls, particulary with the `TableView` and `ListView`.

TableView Advanced Column Resizing

The SmartResize policy brings the ability to intuitively resize columns by providing sensible defaults combined with powerful and dynamic configuration options.

To apply the resize policy to a `TableView` we configure the `columnResizePolicy`. For this exercise we will use a list of hotel rooms. This is our initial table with the `SmartResize` policy activated:

```
tableview(rooms) {
    column("#", Room::id)
    column("Number", Room::number)
    column("Type", Room::type)
    column("Bed", Room::bed)

    columnResizePolicy = SmartResize.POLICY
}
```

Here is a picture of the table with the SmartResize policy activated (Figure 5.7):

Figure 13.1

#	Number	Type	Bed
1	104	Bedroom	Queen
2	105	Bedroom	King
3	106	Bedroom	King
4	107	Suite	Queen
4	108	Bedroom	King
4	109	Conference Room	Queen
4	110	Bedroom	Queen
4	111	Playroom	King
4	112	Bedroom	Queen
4	113	Suite	King

The default settings gave each column the space it needs based on its content, and gave the remaining width to the last column. When you resize a column by dragging the divider between column headers, only the column immediately to the right will be affected, which avoids pushing the columns to the right outside the viewport of the `TableView`.

While this often presents a pleasant default, there is a lot more we can do to improve the user experience in this particular case. It is evident that our table did not need the full 800 pixels it was provided, but it gives us a nice chance to elaborate on the configuration options of the `SmartResize` policy.

The bed column is way too big, and it seems more sensible to give the extra space to the **Type** column, since it might contain arbitrary long descriptions of the room. To give the extra space to the **Type** column, we change its column definition (Figure 5.8):

```
column("Type", Room::type).remainingWidth()
```

Figure 13.2

#	Number	Type	Bed
1	104	Bedroom	Queen
2	105	Bedroom	King
3	106	Bedroom	King
4	107	Suite	Queen
4	108	Bedroom	King
4	109	Conference Room	Queen
4	110	Bedroom	Queen
4	111	Playroom	King
4	112	Bedroom	Queen
4	113	Suite	King

Now it is apparent the **Bed** column looks cramped, being pushed all the way to the left. We configure it to keep its desired width based on the content plus 50 pixels padding:

```
column("Bed", Room:bed").contentWidth(padding = 50.0)
```

The result is a much more pleasant visual impression (Figure 5.9) :

Figure 13.3

#	Number	Type	Bed
1	104	Bedroom	Queen
2	105	Bedroom	King
3	106	Bedroom	King
4	107	Suite	Queen
4	108	Bedroom	King
4	109	Conference Room	Queen
4	110	Bedroom	Queen
4	111	Playroom	King
4	112	Bedroom	Queen
4	113	Suite	King

This fine-tuning may not seem like a big deal, but it means a lot to people who are forced to stare at your software all day! It is the little things that make software pleasant to use.

If the user increases the width of the **Number** column, the **Type** column will gradually decrease in width, until it reaches its default width of 10 pixels (the JavaFX default). After that, the **Bed** column must start giving away its space. We don't ever want the **Bed** column to be smaller that what we configured, so we tell it to use its content-based width plus the padding we added as its minimum width:

```
column("Bed", Room:bed").contentWidth(padding = 50.0, useAsMin = true)
```

Trying to decrease the **Bed** column either by explicitly expanding the **Type** column or implicitly by expanding the **Number** column will simply be denied by the resize policy. It is worth noting that there is also a `useAsMax` choice for the `contentWidth` resize type. This would effectively result in a hard-coded, unresizable column, based on the required content width plus any configured padding. This would be a good policy for the **#** column:

```
column("#", Room::id).contentWidth(useAsMin = true, useAsMax = true)
```

The rest of the examples will probably not benefit the user, but there are still other options at your disposal. Try to make the **Number** column 25% of the total table width:

```
column("Number", Room::number).pctWidth(25.0)
```

When you resize the `TableView`, the **Number** column will gradually expand to keep up with our 25% width requirement, while the **Type** column gets the remaining extra space.

Figure 13.4

#	Number	Type	Bed
1	104	Bedroom	Queen
2	105	Bedroom	King
3	106	Bedroom	King
4	107	Suite	Queen
4	108	Bedroom	King
4	109	Conference Room	Queen
4	110	Bedroom	Queen
4	111	Playroom	King
4	112	Bedroom	Queen
4	113	Suite	King

An alternative approach to percentage width is to specify a weight. This time we add weights to both **Number** and **Type**:

```
column("Number", Room::number).weigthedWidth(1.0)
column("Type", Room::type).weigthedWidth(3.0)
```

The two weighted columns share the remaining space after the other columns have received their fair share. Since the **Type** column has a weight that is three times bigger than the **Number** column, its size will be three times bigger as well. This will be reevaluated as the `TableView` itself is resized.

Figure 13.5

#	Number	Type	Bed
1	104	Bedroom	Queen
2	105	Bedroom	King
3	106	Bedroom	King
4	107	Suite	Queen
4	108	Bedroom	King
4	109	Conference Room	Queen
4	110	Bedroom	Queen
4	111	Playroom	King
4	112	Bedroom	Queen
4	113	Suite	King

This setting will make sure we keep the mentioned ratio between the two columns, but it might become problematic if the `TableView` is resized to be very small. The the **Number** column would not have space to show all of its content, so we guard against that by specifying that it should never grow below the space it needs to show its content, plus some padding, for good measure:

```
column("Number", Room::number).weigthedWidth(1.0, minContentWidth = true, padding = 10
.0)
```

This makes sure our table behaves nicely also under constrained width conditions.

Dynamic content resizing

Since some of the resizing modes are based on the actual content of the columns, they might need to be reevaluated even when the table or it's columns aren't resized. For example, if you add or remove content items from the backing list, the required content measurements might need to be updated. For this you can call the `requestResize` function after you have manipulated the items:

```
SmartResize.POLICY.requestResize(tableView)
```

In fact, you can ask the TableView to ask the policy for you:

```
tableView.requestResize()
```

Statically setting the content width

In most cases you probably want to configure your column widths based on either the total available space or the content of the columns. In some cases you might want to configure a specific width, that that can be done with the `prefWidth` function:

```
column("Bed", Room::bed).prefWidth(200.0)
```

A column with a preferred width can be resized, so to make it non-resizable, use the `fixedWidth` function instead:

```
column("Bed", Room::bed).fixedWidth(200.0)
```

When you hard-code the width of the columns you will most likely end up with some extra space. This space will be awarded to the right most resizable column, unless you specify `remainingWidth()` for one or more column. In that case, these columns will divide the extra space between them.

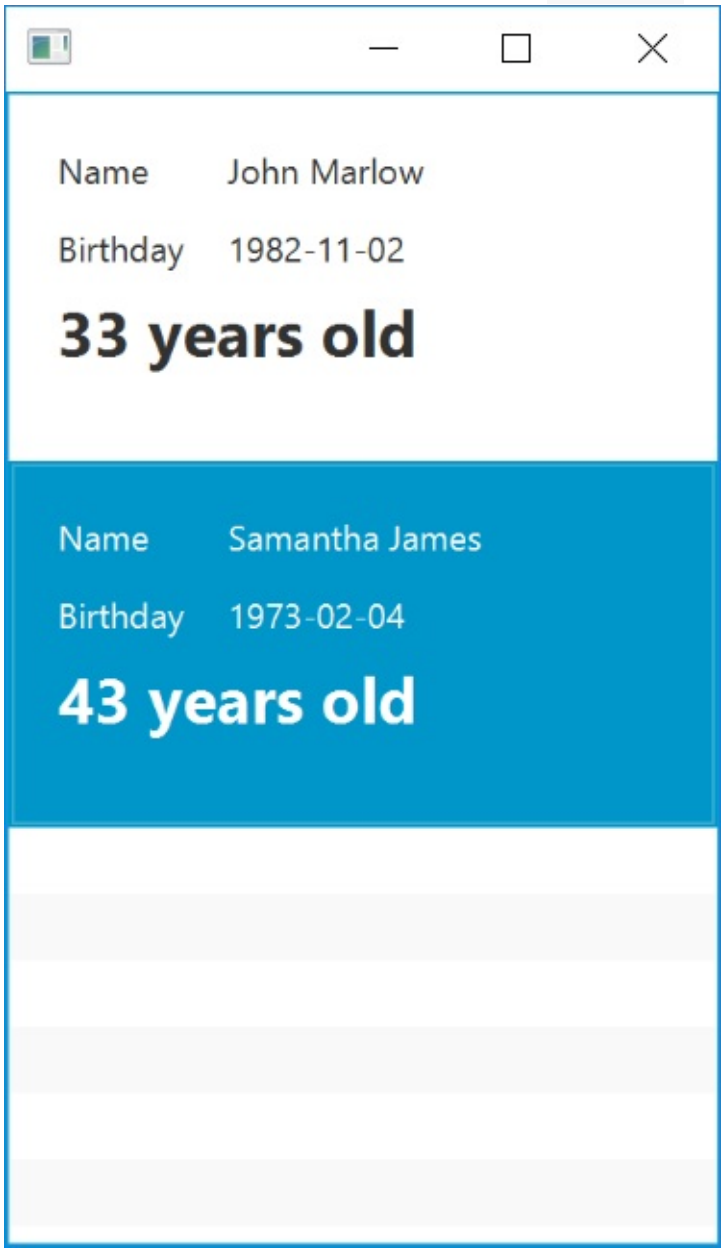
In the case where not all columns can be afforded their preferred width, all resizable columns must give away some of their space, but the `SmartResize` Policy makes sure that the column with the biggest reduction potential will give away its space first. The reduction potential is the difference between the current width of the column and its defined minimum width.

Custom Cell Formatting in ListView

Even though the default look of a `ListView` is rather boring (because it calls `toString()` and renders it as text) you can modify it so that every cell is a custom `Node` of your choosing. By calling `cellCache()`, TornadoFX provides a convenient way to override what kind of `Node` is returned for each item in your list (Figure 5.2).

```
class MyView: View() {

    val persons = listOf(
            Person("John Marlow", LocalDate.of(1982,11,2)),
            Person("Samantha James", LocalDate.of(1973,2,4))
    ).observable()

    override val root = listview(persons) {
        cellFormat {
            graphic = cache {
                form {
                    fieldset {
                        field("Name") {
                            label(it.name)
                        }
                        field("Birthday") {
                            label(it.birthday.toString())
                        }
                        label("${it.age} years old") {
                            alignment = Pos.CENTER_RIGHT
                            style {
                                fontSize = 22.px
                                fontWeight = FontWeight.BOLD
                            }
                        }
                    }
                }
            }
        }
    }
}

class Person(val name: String, val birthday: LocalDate) {
    val age: Int get() = Period.between(birthday, LocalDate.now()).years
}
```

Figure 5.2 - A custom cell rendering for `ListView`



The `cellFormat` function lets you configure the `text` and/or `graphic` property of the cell whenever it comes into view on the screen. The cells themselves are reused, but whenever the `ListView` asks the cell to update its content, the `cellFormat` function is called. In our example we only assign to `graphic`, but if you just want to change the string representation you should assign it to `text`. It is completely legitimate to assign it to both `text` and `graphic`. The values will automatically be cleared by the `cellFormat` function when a certain list cell is not showing an active item.

Note that assigning new nodes to the `graphic` property every time the list cell is asked to update can be expensive. It might be fine for many use cases, but for heavy node graphs, or node graphs where you utilize binding towards the UI components inside the cell, you should cache the resulting node so the Node graph will only be created once per node. This is done using the `cache` wrapper in the above example.

Assign If Null

If you have a reason for wanting to recreate the graphic property for a list cell, you can use the `assignIfNull` helper, which will assign a value to any given property if the property doesn't already contain a value. This will make sure that you avoid creating new nodes if `updateItem` is called on a cell that already has a graphic property assigned.

```
cellFormat {
    graphicProperty().assignIfNull {
        label("Hello")
    }
}
```

ListCellFragment

The `ListCellFragment` is a special fragment which can help you manage `ListView` cells. It extends `Fragment`, and includes some extra `ListView` specific fields and helpers. You never instantiate these fragments manually, instead you instruct the `ListView` to create them as needed. There is a one-to-one correlation between `ListCell` and `ListCellFragment` instances. Only one `ListCellFragment` instance will over its lifecycle be used to represent different items.

To understand how this works, let's consider a manually implemented `ListCell`, essentially the way you would do in vanilla JavaFX. The `updateItem` function will be called when the `ListCell` should represent a new item, no item, or just an update to the same item. When you use a `ListCellFragment`, you do not need to implement something akin to `updateItem`, but the `itemProperty` inside it will update to represent the new item automatically. You can listen to changes to the `itemProperty`, or better yet, bind it directly to a `ViewModel`. That way your UI can bind directly to the `ViewModel` and no longer need to care about changes to the underlying item.

Let's recreate the form from the `cellFormat` example using a `ListCellFragment`. We need a `ViewModel` which we will call `PersonModel` (Please see the `Editing Models and Validation` chapter for a full explanation of the `ViewModel`) For now, just imagine that the `ViewModel` acts as a proxy for an underlying `Person`, and that the `Person` can be changed while the observable values in the `ViewModel` remain the same. When we have created our `PersonCellFragment`, we need to configure the `ListView` to use it:

```
listview(personlist) {
    cellFragment(PersonCellFragment::class)
}
```

Now comes the `ListCellFragment` itself.

```
class PersonListFragment : ListCellFragment<Person>() {
    val person = PersonModel().bindTo(this)

    override val root = form {
        fieldset {
            field("Name") {
                label(person.name)
            }
            field("Birthday") {
                label(person.birthday)
            }
            label(stringBinding(person.age) { "$value years old" }) {
                alignment = Pos.CENTER_RIGHT
                style {
                    fontSize = 22.px
                    fontWeight = FontWeight.BOLD
                }
            }
        }
    }
}
```

Because this Fragment will be reused to represent different list items, the easiest approach is to bind the ui elements to the ViewModel's properties.

The `name` and `birthday` properties are bound directly to the labels inside the fields. The age string in the last label needs to be constructed using a `stringBinding` to make sure it updates when the item changes.

While this might seem like slightly more work than the `cellFormat` example, this approach makes it possible to leverage everything the Fragment class has to offer. It also forces you to define the cell node graph outside of the builder hierarchy, which improves refactoring possibilities and enables code reuse.

Additional helpers and editing support

The `ListCellFragment` also have some other helper properties. They include the `cellProperty` which will update whenever the underlying cell changes and the `editingProperty`, which will tell you if this the underlying list cell is in editing mode. There are also editing helper functions called `startEdit`, `commitEdit`, `cancelEdit` plus an `onEdit` callback. The `ListCellFragment` makes it trivial to utilize the existing editing capabilites of the `ListView`. A complete example can be seen in the TodoMVC demo application.

OSGi

This chapter is geared primarily towards folks who already have familiarity with OSGi, which stands for **Open Services Gateway Initiative**. The idea behind OSGi is adding and removing modules to a Java application without the need for restarting. TornadoFX supports OSGi and allows highly modular and dynamic applications.

If you have no interest in OSGi currently, you are welcome to skip this chapter. However, it is highly recommended to at least know what it is so you can identify moments in the future that make it handy.

TornadoFX comes with the metadata needed for an OSGi runtime to detect and enable it. When the `tornadofx.jar` is
loaded in an OSGi container, a number of services are automatically installed in the runtime. These services enable
some very interesting features which we will discuss.

OSGi Introduction

Please be familiar with the basics of OSGi before you continue this chapter. To get a quick overview of OSGi
technology you can check out the tutorials on the
OSGi Alliance website. The
Apache Felix tutorials
are also a good starting point reference for basic OSGi patterns.

Services

When the TornadoFX JAR is loaded, you can create your own TornadoFX bundle and create your
application any way you like. However, some usage patterns are so typical and useful that TornadoFX has built-in
support for them.

Dynamic Applications

The dynamic nature of OSGi lends itself well to GUI applications in general. The ability to have certain
functionality come and go as the environment changes can be powerful. JavaFX itself is unfortunately written in
a way that prevents you from starting another JavaFX application after the initial application shuts down. To
circumvent this shortcoming and enable you to stop and start your application as many times as you want, TornadoFX
provides a way to register your `App` class with an application proxy which will keep the JavaFX environment running
even when your application shuts down.

To get started, implement a `BundleActivator` that provides a means to `start()` and `stop()` an `App`. Registering your application for this functionality can be done by calling `context.registerApplication` with your
`App` class as the single parameter in your bundle `Activator`:

```
class Activator : BundleActivator {
    override fun start(context: BundleContext) {
        context.registerApplication(MyApp::class)
    }

    override fun stop(context: BundleContext) {
    }
}
```

If you prefer OSGi declarative services instead, this will have the same effect provided that you have the OSGi DS
bundle loaded:

```
@Component
class AppRegistration : ApplicationProvider {
    override val application = MyApp::class
}
```

Provided that the TornadoFX bundle is available in your container, this is enough to start your application
automatically once the bundle is activated. You can now stop and start it as many times as you like by stopping and
starting the bundle.

Dynamic Stylesheets

You can provide type-safe stylesheets to other TornadoFX bundles by registering them in an `Activator`:

```
class Activator : BundleActivator {
    override fun start(context: BundleContext) {
        context.registerStylesheet(Styles::class)
    }

    override fun stop(context: BundleContext) {
    }
}
```

Using OSGi Declarative Services the registration looks like this:

```
@Component
class StyleRegistration : StylesheetProvider {
    override val stylesheet = Styles::class
}
```

Whenever this bundle is loaded, every active `View` will have this stylesheet applied. When the bundle is unloaded,
the stylesheet is automatically removed. If you want to provide multiple stylesheets based on the same style
classes, it is a good idea to create one bundle that exports the `cssclass` definitions, so that your Views can
reference these styles, and the stylesheet bundles can create selectors based on them.

Dynamic Views

A cool aspect of OSGi is the ability to have UI elements pop up when they become available. A typical use case
could be a "dashboard" application. In this example, the base application bundle contains a `View` that can hold other
Views, and tells the TornadoFX OSGi Runtime that it would like to automatically embed Views if they meet certain
criteria.

For instance, we can create a `View` that contains a `VBox`. We tell the TornadoFX OSGi Runtime that we would like to have other Views
embedded into it if they are tagged with the discriminator **dashboard**:

```
class Dashboard : View() {
    override val root = VBox()

    init {
        title = "Dashboard Application"
        addViewsWhen { it.discriminator == "dashboard" }
    }
}
```

If the `addViewsWhen` function returns true, the `View` is added to the `VBox`. To offer up Views to this Dashboard,
another bundle would declare that it wants to export it's View by setting the `dashboard` discriminator. Here we register
a fictive `MusicPlayer` view to be docked into the dashboard when it's bundle becomes active.

```
class Activator : BundleActivator {
    override fun start(context: BundleContext) {
        context.registerView(MusicPlayer::class, "dashboard")
    }

    override fun stop(context: BundleContext) {
    }
}
```

Again, the OSGi Declarative Services way of exporting the View would look like this:

```
@Component
class MusicPlayerRegistration : ViewProvider {
    override val discriminator = "dashboard"
    override fun getView() = find(MusicPlayer::class)
}
```

The `addViewsWhen` function is smart enough to inspect the `VBox` and find out how to add the child View
it was presented. It can also figure out that if you call the function on a `TabPane` it would create a new `Tab`
and set the title to the child View title etc. If you would like to do something custom with the presented Views,
you can return `false` from the function so that the child View will not be added automatically and then do whatever
you want with it. Even though the Tab example is supported out of the box, you could do it explicitly like this:

```
tabPane.addViewsWhen {
    if (it.discriminator == "dashboard") {
        val view = it.getView()
        tabPane.tab(view.title, view.root)
    }
    false
}
```

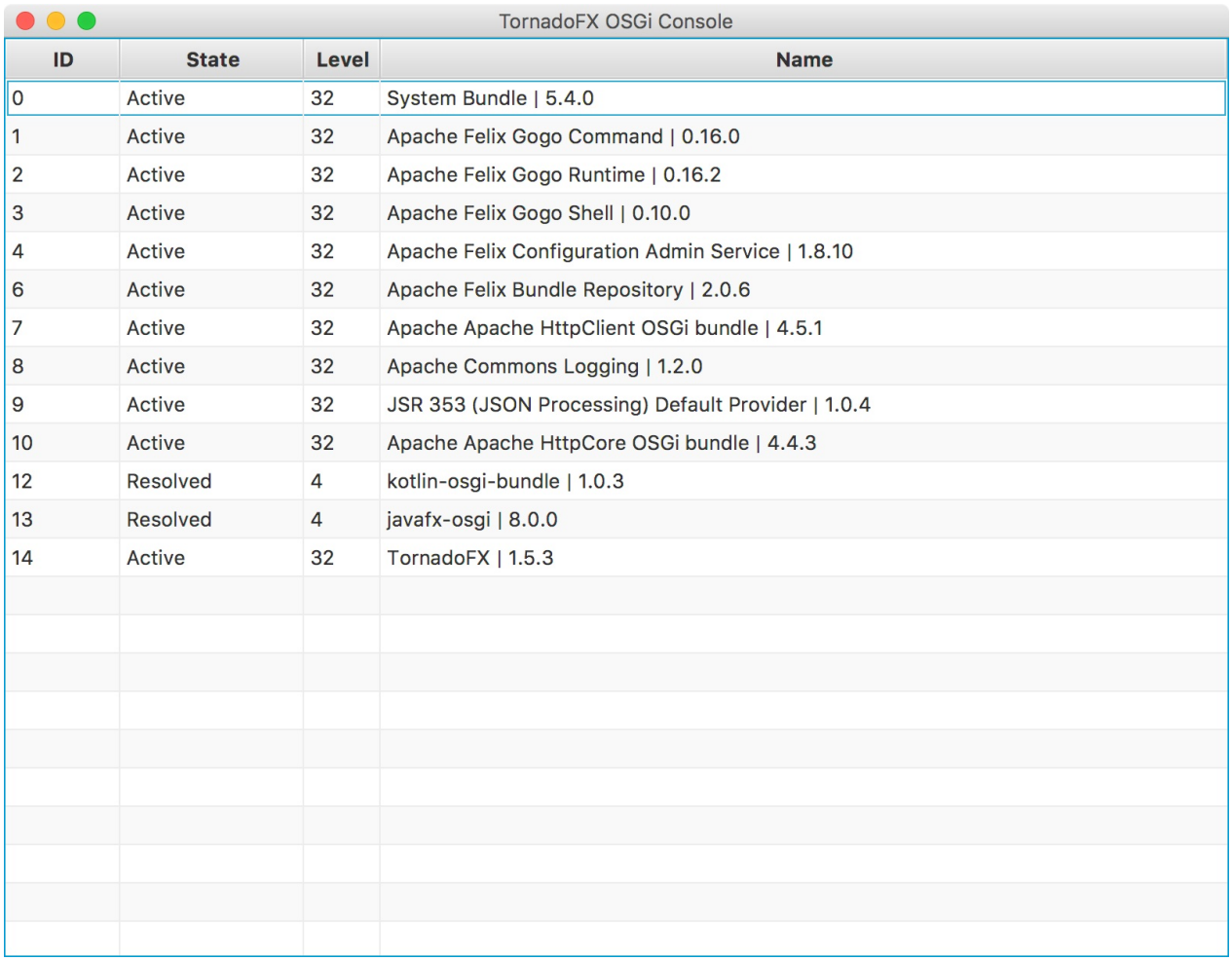
> Manual handling of dynamic Views

Create your first OSGi bundle

A good starting point is the `tornadofx-maven-osgi-project` template in the TornadoFX IntelliJ IDEA plugin. This
contains everything you need to build OSGi bundles from your sources. The OSGI IDEA plugin makes it very easy to
setup and run an OSGi container directly from the IDE. There is a screencast at https://www.youtube.com/watch?v=liOFCH5MMKk that shows these concepts in action.

OSGi Console

TornadoFX has a built in OSGi console from which you can inspect bundles, change their state and even install new
bundles with drag and drop. You can bring up the console with `Alt-Meta-O` or configure another shortcut by setting
`FX.osgiConsoleShortcut` or programmatically opening the `OSGIConsole` View.



Requirements

To run TornadoFX in an OSGi container, you need to load the required bundles. Usually this is a matter of dumping
these jars into the `bundle` directory of the container. Note that any jar that is to be used in an OSGi container
needs to be "OSGi enabled", which effectively means adding some OSGi specific entries the `META-INF/MANIFEST.MF`
file.

We provided a complete installation with Apache Felix and TornadoFX already installed at http://tornadofx.io/#felix. Remember to swap the `tornadofx.jar` for the latest version,
as this bundle is most likely lagging a couple of versions behind.

These are the required artifacts for any TornadoFX application running in an OSGi container. Your container might
already be bundle with some of these, so check the container documentation for further details.

Artifact	Version	Binary
JavaFX 8 OSGi Support	8.0	jar
TornadoFX	1.7.12	jar
Kotlin OSGI Bundle*	1.5.11	jar
Configuration Admin**	1.8.10	jar
Commons Logging	1.2	jar
Apache HTTP-Client	4.5.2	jar
Apache HTTP-Core	4.4.5	jar
JSON	1.0.4	jar

`*` The Kotlin OSGi bundle contains special versions of `kotlin-stdlib` and `kotlin-reflect` with the required OSGi
manifest information.

`**` This links to the Apache Felix implementation of the OSGi Config Admin interface. Feel free to use the implementation from your OSGi container instead. Some containers, like Apache Karaf, already
has the Config Admin bundle loaded, so you won't need it there.

Scopes

Scope is a simple construct that enables some interesting and helpful behavior in a TornadoFX application.

When you use `inject()` or `find()` to locate a `Controller` or a `View`, you will by default get back a singleton instance, meaning that wherever you locate that object in your code, you will get back the same instance. Scopes provide a way to make a `View` or `Controller` unique to a smaller subset of instances in your application.

It can also be used to run multiple versions of the same application inside the same JVM, for example with JPro, which exposes TornadoFX application in a web browser.

A Master/Detail example

In an MDI Application you can open an editor in a new window, and ensure that all the injected resources are unique to that window. We will leverage that technique to create a person editor that allows you to open a new window to edit each person.

We start by defining a table interface where you can double click to open the person editor in a separate window.

```
class PersonList : View("Person List") {
    val ctrl: PersonController by inject()

    override val root = tableview<Person>() {
        column("#", Person::idProperty)
        column("Name", Person::nameProperty)
        onUserSelect { editPerson(it) }
        asyncItems { ctrl.people() }
    }

    fun editPerson(person: Person) {
        val editScope = Scope()
        val model = PersonModel()
        model.item = person
        setInScope(model, editScope)
        find(PersonEditor::class, editScope).openWindow()
    }
}
```

The `edit` function creates a new `Scope` and injects a `PersonModel` configured with the selected user into that scope. Finally, it retrieves a `PersonEditor` in the context of the new scope and opens a new window.

When the `PersonEditor` is initialized, it will look up a `PersonModel` via injection. The default context for `inject` and `find` is always the scope that created the component, so it will look in the `personScope` we just created.

```
val model: PersonModel by inject()
```

Breaking Out of the Current Scope

When no scope is defined, injectable resources are looked up in the default scope. There is an item representing that scope called `DefaultScope`. In the above example, the editor might have called out to a `PersonController` to perform a save operation in a database or via a REST call. This `PersonController` is most probably stateless, so there is no need to create a separate controller for each edit window. To access the same controller in all editor windows, we supply the scope we want to find the controller in:

```
val controller: PersonController by inject(DefaultScope)
```

This effectively makes the `PersonController` a true singleton object again, with only a single instance in the whole application.

The default scope for new injected objects are always the current scope for the component that calls `inject` or `find`, and consequently all objects created in that injection run will belong to the supplied scope.

Keeping State in Scopes

In the previous example we used injection on a scope level to get a hold of our resources. It is also possible to subclass `Scope` and put arbitrary data in there. Each TornadoFX `Component` has a `scope` property that gives you access to that scope instance. You can even override it to provide the custom subclass so you don't need to cast it on every occasion:

```
override val scope = super.scope as PersonScope
```

Now whenever you access the `scope` property from your code, it will be of type `PersonScope`. It now contains a `PersonModel` that will only be available to this scope:

```
class PersonScope : Scope() {
    val model = PersonModel()
}
```

Let's change our previous example slightly to access the model inside the scope instead of using injection. First we change the editPerson function:

```
fun editPerson(person: Person) {
    val editScope = PersonScope()
    editScope.model.item = person
    find(PersonEditor::class, editScope).openWindow()
}
```

The custom scope already has an instance of `PersonModel`, so we just configure the item for that scope and open the editor. Now the editor can override the type of scope and access the model:

```
// Cast scope
override val scope = super.scope as PersonScope
// Extract our view model from the scope
val model = scope.model
```

Both approaches work equally well, but depending on your use case you might prefer one over the other.

Global application scope

As we hintet to initially, you can run multiple applications in the same JVM and keep them completely separate by using scopes. By default, JavaFX does not support multi tenancy, and can only start a single JavaFX application per JVM, but new technologies are emerging that leverages multitenancy and will even expose your JavaFX based applications to the web. One such technology is JPro.io, and TornadoFX supports multitenancy for JPro applications by leveraging scopes.

There is no special JPro classes in TornadoFX, but supporting JPro is very simple by leveranging scopes:

Using TornadoFX with JPro

JPro will create a new instance of your App class for each new web user. Also, to access the JPro WebAPI you need to get access to the stage created for each user. In this example we subclass `Scope` to create a special JProScope that contains the stage that was given to each application instance:

```
class JProScope(val stage: Stage) : Scope() {
    val webAPI: WebAPI get() = WebAPI.getWebAPI(stage)
}
```

The next step is to subclass `JProApplication` to define our entry point. This app class is in addition to our existing TornadoFX App class, which boots the actual application:

```
class Main : JProApplication() {
    val app = OurTornadoFXApp()

    override fun start(primaryStage: Stage) {
        app.scope = JProScope(primaryStage)
        app.start(primaryStage)
    }

    override fun stop() {
        app.stop()
        super.stop()
    }
}
```

Whenever a new user visits our site, the `Main` class is created, together with a new instance of our actual TornadoFX application.

In the `start` function we assign a new `JProScope` to the TornadoFX app instance and then call `app.start`. From there on out, all instances created using `inject` and `find` will be in the context of that JPro instance.

As usual, you can break out of the `JProScope` to access JVM level globals by supplying the `DefaultScope` or any other shared scope to the `inject` or `find` functions.

We should provide a utility function that makes it easy to access the JPro WebAPI from any Component:

```
val Component.webAPI: WebAPI get() = (scope as JProScope).webAPI
```

The `scope` property of any `Component` will be the `JProScope` so we can cast it and access the `webAPI` property we defined in our custom scope class.

Testing with Scopes

Since Scopes allow you to create separate instances of components that are usually singletons, you can leverage Scopes to test Views and even whole App instances.

For example, to generate a new Scope and lookup a View in that scope, you can use the following code:

```
val testScope = Scope()
val myView = find<MyView>(testScope)
```

EventBus

An `EventBus` is a versatile tool with a multitude of use cases. Depending on your coding style and preferences, you might want to reduce coupling between controllers and views by passing messages instead of having hard references to each other. The TornadoFX event bus can make sure that the messages are received on the appropriate thread, without having to do that concurrency house-keeping manually.

People use event buses for many different use cases. TornadoFX does not dictate when or how you should use them, but we want to show you some of the advantages it can provide to you.

Structure of the EventBus

As with any typical event bus implementation, you can fire events as well as subscribe and unsubscribe to events on the bus. You create an event by extending the `FXEvent` class. In some cases, an event can be just a signal to some other component to trigger something to happen. In other cases, the event can contain data which will be broadcast to the subscribers of this event. Let us look at a couple of event definitions and how to use them.

Picture a UI where the user can click a `Button` to refresh a list of customers. The `View` knows nothing of where the data is coming from or how it is produced, but it subscribes to the data events and uses the data once it arrives. Let us create two event classes for this use case.

First we define an event signal type to notify any listeners that we want some customer data:

```
import tornadofx.EventBus.RunOn.*

object CustomerListRequest : FXEvent(BackgroundThread)
```

This event object is an application-wide `object`. Because it will never need to contain data, it will simply be broadcast to say that we want the customer list. The `RunOn` property is set to `BackgroundThread`, to signal that the receiver of this event should operate off of the JavaFX Application Thread. That means it will be given a background thread by default, so that it can do heavy work without blocking the UI. In the example above, we have added a static import for the `RunOn` enum, so that we just write `BackgroundThread` instead of `EventBus.RunOn.BackgroundThread`. Your IDE will help you to make this import so your code looks cleaner.

A button in the UI can fire this event by using the `fire` function:

```
button("Load customers").action {
    fire(CustomerListRequest)
}
```

A `CustomerController` might listen for this event, and load the customer list on demand before it fires an event with the actual customer data. First we need to define an event that can contain the customer list:

```
class CustomerListEvent(val customers: List<Customer>) : FXEvent()
```

This event is a `class` rather than an `object`, as it will contain actual data and vary. Also, it did not specify another value for the `RunOn` property, so this event will be emitted on the JavaFX Application Thread.

A controller can now subscribe to our request for data and emit that data once it has it:

```
class CustomerController : Controller() {
    init {
        subscribe<CustomerListRequest> {
            val customers = loadCustomers()
            fire(CustomerListEvent(customers))
        }
    }

    fun loadCustomers(): List<Customer> = db.selectAllCustomers()
}
```

Back in our UI, we can listen to this event inside the customer table definition:

```
tableview<Customer> {
   column("Name", Customer::nameProperty)
   column("Age", Customer::ageProperty)

   subscribe<CustomerListEvent> { event ->
       items.setAll(event.customers)
   }
}
```

We tell the event bus that we are interested in `CustomerListEvent`s, and once we have such an event we extract the customers from the event and set them into the `items` property of the `TableView`.

Query Parameters In Events

Above you saw a signal used to ask for data, and an event returned with that data. The signal could just as well contain query parameters. For example, it could be used to ask for a specific customer. Imagine these events:

```
class CustomerQuery(val id: Int) : FXEvent(false)
class CustomerEvent(val customer: Customer) : FXEvent()
```

Using the same procedure as above, we can now signal our need for a specific `Customer`, but we now need to be more careful with the data we get back. If our UI allows for multiple customers to be edited at once, we need to make sure that we only apply data for the customer we asked for. This is quite easily accounted for though:

```
class CustomerEditor(val customerId: Int) : View() {
   val model : CustomerModel

   override val root = form {
       fieldset("Customer data") {
          field("Name") {
              textfield(model.name)
          }
          // More fields and buttons here
       }
   }

   init {
       subscribe<CustomerEvent> {
           if (it.customer.id == customerId)
               model.item = it.customer
       }
       fire(CustomerQuery(customerId))
   }
}
```

The UI is created before the interesting bit happens in the `init` function. First, we subscribe to `CustomerEvent`s, but we make sure to only act once we retrieve the customer we were asking for. If the `customerId` matches, we assign the customer to the `item` property of our `ItemViewModel`, and the UI is updated.

A nice side effect of this is that our customer object will be updated whenever the system emits new data for this customer, no matter who asked for them.

Events and Threading

When you create a subclass of `FXEvent`, you dictate the value of the `runOn` property. It is `ApplicationThread` by default, meaning that the subscriber will receive the event on the JavaFX Application Thread. This is useful for events coming from and going to other UI components, as well as backend services sending data to the UI. If you want to signal something to a backend service, one which is likely to perform heavy, long-running work, you should set `runOn` to `BackgroundThread`, making sure the subscriber will operate off of the UI thread. The subscriber now no longer needs to make sure that it is off of the UI thread, so you remove a lot of thread-related house keeping calls. Used correctly this is convenient and powerful. Used incorrectly, you will have a non responsive UI. Make sure you understand this completely before playing with events, or always wrap long running tasks in `runAsync {}`.

Scopes

The event bus emits messages across all scopes by default. If you want to limit signals to a certain scope, you can supply the second parameter to the `FXEvent` constructor. This will make sure that only subscribers from the given scope will receive your event.

```
class CustomerListRequest(scope: Scope) : FXEvent(BackgroundThread, scope)
```

The `CustomerListRequest` is not an object anymore since it needs to contain the scope parameter. You would now fire this event from any UI Component like this:

```
button("Load customers").action {
    fire(CustomerListRequest(scope))
}
```

The scope parameter from your `UIComponent` is passed into the `CustomerListRequest`. When customer data comes back, the framework takes care of discriminating on scope and only apply the results if they are meant for you. You do not need to mention the scope to the subscribe function call, as the framework will associate your subscription with the scope your are in at the time you create the subscription.

```
subscribe<CustomerListEvent> { event ->
   items.setAll(event.customers)
}
```

Invalidation of Event Subscribers

In many event bus implementations, you are left with the task of deregistering the subscribers when your UI components should no longer receive them. TornadoFX takes an opinionated approach to event cleanup so you do not have to think about it much.

Subscriptions inside `UIComponents` like `View` and `Fragment` are only active when that component is docked. That means that even if you have a `View` that has been previously initialized and used,event subscriptions will not reach it unless the `View` is docked inside a window or some other component. Once the view is docked, the events will reach it. Once it is undocked, the events will no longer be delivered to your component. This takes care of the need for you to manually deregister subscribers when you discard of a view.

For `Controllers` however, subscriptions are always active until you call `unsubscribe`. You need to keep in mind that controllers are lazily loaded, so if nothing references your controller, the subscriptions will never be registered in the first place. If you have such a controller with no other references, but you want it to subscribe to events right away, a good place to eagerly load it would be the `init` block of your `App` subclass:

```
class MyApp : App(MainView::class) {
    init {
        // Eagerly load CustomerController so it can receive events
        find(CustomerController::class)
    }
}
```

Duplicate Subscriptions

To avoid registering your subscriptions multiple times, make sure you do not register the event subscriptions in `onDock()` or any other callback function that might be invoked more than once for the duration of the component lifecycle. The safest place to create event subscriptions is in the `init` block of the component.

Should I use events for UI logic everywhere?

Using events for everything might seem like a noble idea, and some people might prefer it because of the loose coupling it facilitates. However, the `ItemViewModel` with injection is often a more streamlined solution to passing data and keeping UI state. This example was provided to explain how the event system works, not to convince you to write your UIs this way all the time.

Many feel that events might be better suited for passing signals rather than actual data, so you might also consider subscribing to signals and then actively retrieving the data you need instead.

Unsubscribe after event is processed

In some situations you might want to only want to trigger your listener a certain amount of times. Admittedly, this is not very convenient. You can pass the `times = n` parameter to subscribe to control how many times the event is triggered before it is unsubscribed:

```
object MyEvent : FXEvent()

class MyView : View() {
    override val root = stackpane {
        paddingAll = 100
        button("Fire!").action {
            fire(MyEvent)
        }
    }

    init {
        subscribe<MyEvent>(times = 2) {
            alert(INFORMATION, "Event received!", "This message should only appear twi
ce.")
        }
    }
}
```

You can also manually unsubscribe based on an arbitrary condition, or simply after the first run:

```
class MyView : View() {
    override val root = stackpane {
        paddingAll = 100
        button("Fire!").action {
            fire(MyEvent)
        }
    }

    init {
         subscribe<MyEvent> {
            alert(INFORMATION, "Event received!", "This message should only appear once.")
            unsubscribe()
        }
    }
}
```

Workspaces

Java Business applications have traditionally been based on one of the Rich Client Frameworks, namely the NetBeans Platform or Eclipse RCP. An important reason for choosing an RCP platform has been the workspace-like functionality they provide. Some important features of a workspace are:

- Common action buttons that tie to the state of the docked view (Save, Refresh, etc)
- Context-based UI nodes added to the common workspace interface
- Navigation stack for traversing visited views, controlled through back and forward buttons like a web browser
- Menu system with dynamic contributions and modifications

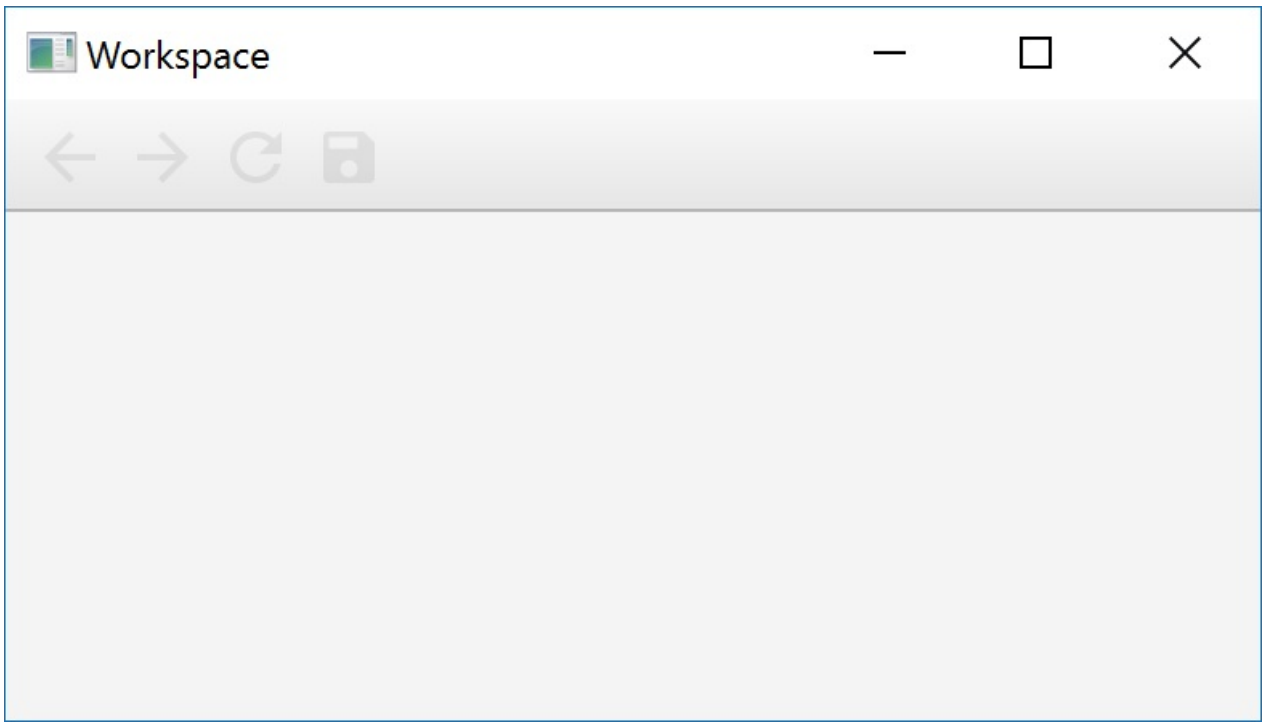
TornadoFX has begun to bridge the gap between the RCP platforms by providing Workspaces. While still in its infancy, the default functionality is a solid foundation for business applications in need of the features discussed above.

A Simple Workspace Example

To kick off a Workspace app, all you need to do is to subclass `App` and set the primary `View` to `Workspace::class`. The result can be seen below (Figure 16.1):

```
class MyApp : App(Workspace::class)
```

Figure 16.1

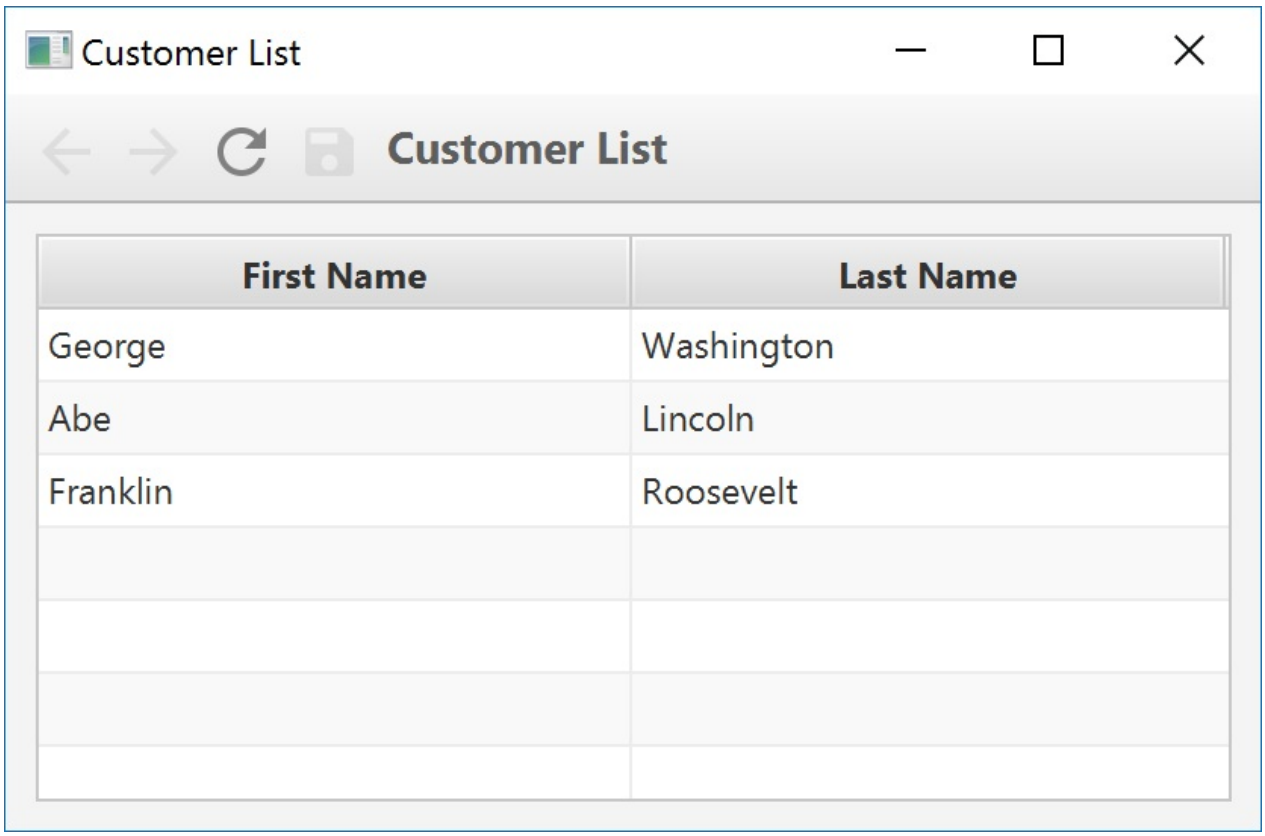


The resulting Workspace consists of a button bar with four default buttons and an empty content area below it. The content area can house any `UIComponent`. You add a component to the `content` area by calling `workspace.dock()` on it. If you show the `Workspace` without a docked `View`, it will by default only take up the space needed for the buttons. The window in Figure 16.1 was resized after it was opened.

Let's pretend we have a `CustomerList` component that we would like to dock in the `Workspace` as the application starts. We do this by overriding the `onBeforeShow` callback:

```
class MyApp : App(Workspace::class) {
    override fun onBeforeShow(view: UIComponent) {
        workspace.dock<CustomerList>()
    }
}
```

Figure 16.2



To keep things focused, we will leave out the `CustomerList` implementation code which simply displays a `TableView` with some Customers. What is interesting however, is that the **Refresh** button in the `Workspace` was enabled when the `CustomerList` was docked, while the **Save** button remained disabled.

Leveraging the Workspace buttons

Whenever a `UIComponent` is docked in the `Workspace`, the **Refresh**, **Save**, and **Delete** buttons will be enabled by default. This happens because the `Workspace` looks at the `refreshable`, `savable` and `deletable` properties in the docked component. Every `UIComponent` returns a boolean property with the default value of `true`, which the `Workspace` then connects to the enabled state of these buttons. In the `CustomerList` example, the TornadoFX maintainers made sure the **Save** button was always disabled by overriding this property:

```
override val savable = SimpleBooleanProperty(false)
```

We can achieve the same result by calling `disableSave()` in the `init` block, and the same goes for `disableRefresh()` and `disableDelete()`.

We did not touch the other buttons, so they remain `true` as per the default. Whenever the `Refresh` button is called, it will fire the `onRefresh` function in the `View`. You can override this to provide your refresh action:

```
class MyApp: App(MyWorkspace::class) {
    override fun onBeforeShow(view: UIComponent) {
        workspace.dock<MyView>()
    }
}

class MyWorkspace: Workspace() {
    override fun onRefresh() {
      customerTable.asyncItems { customerController.listCustomers() }
    }
}
```

Same goes for the **Delete** button. We will revisit the **Save** button and introduce a neat trick to only activate it when there are dirty changes later in this chapter.

Tabbed Views

You may at one point dock a View containing a `TabPane` inside of a `Workspace`, and then add tabs which represents further UIComponents. You can quite easily proxy the savable, refreshable and deletable state and actions from the `Workspace` onto the `View` represented by the currently active Tab. Consider a Customer Editor which has tabs for editing customer data, and one for editing contacts for that customer. Whenever the user selects one of the tabs, the buttons in the Workspace should interact with the state and actions from the selected tab view.

```
class CustomerEditor : View("Customer Editor") {
    override val root = tabpane {
        tab(CustomerBasicDataEditor::class)
        tab(ContactListEditor::class)
        connectWorkspaceActions()
    }
}
```

That single call to `connectWorkspaceActions()` takes care of everything for us. The actual implementation of the two sub views are omitted for brevity, but you can imagine that they share a `CustomerViewModel` injected into the scope they share for example.

The actual implementation of `connectWorkspaceActions` is quite simple, and reveals what's going on under the cover:

```
fun TabPane.connectWorkspaceActions() {
    savableWhen { savable }
    whenSaved { onSave() }

    deletableWhen { deletable }
    whenDeleted { onDelete() }

    refreshableWhen { refreshable }
    whenRefreshed { onRefresh() }
}
```

This function is declared inside `UIComponent`, so the `savableWhen`, `deletableWhen`, and `refreshableWhen` are performed on the UIComponent. Those state are then bound to the `savable`, `deletable` and `refreshable` state of the `TabPane`. But wait... a `TabPane` does not have those functions?! Yes, in TornadoFX it does :) You can probably guess that the implementation is again another proxy into the currently selected `Tab` in the `TabPane`, and a lookup of the UIComponent represented by the `content` property of that `Tab`. Whenever the `Tab` changes (or when the content of the tab changes), the underlying UIComponent is looked up, and the pertinent states are bound to the `Workspace`.

It would also be possible to bind these states and connect the actions more explicitly. You will never or seldom need to do that, but the following example might help your understanding of the proxy mechanism.

```
class TooExplicitCustomerEditor : View() {
    override val root = tab {
        ...
    }
    override val savable = root.savable
    override val refreshable = root.refreshable
    override val deletable = root.deletable

    override fun onSave() {
        root.onSave()
    }

    override fun onDelete() {
        root.onDelete()
    }

    override fun onRefresh() {
        root.onRefresh()
    }
}
```

As mentioned, you never need to do this and should always use the `connectWorkspaceActions` call, but you might want to override one of `onSave`, `onDelete` or `onRefresh` to perform some action in the main editor before calling the same action inside the active tab by calling `root.onXXX`. Let's say that the refresh call in the main editor reloads the customer, but you also want to have the contact list refresh if that view is currently active. This could be done like this:

```
class CustomerEditor : View() {
    val customerController : CustomerController by inject()
    val customer: CustomerModel by inject()

    override val root = tabpane {
        tab(CustomerBasicDataEditor::class)
        tab(ContactListEditor::class)
        connectWorkspaceActions()
    }

    override val onRefresh() {
        runAsync {
            customerController.getCustomer(customer.id.value)
        } ui {
            customer.item = it
            root.onRefresh()
        }
    }
}
```

This little trick enables you to handle the actual reload of the customer in the main view instead of reimplementing it in every tab.

Forwarding button state and actions

As we have seen, the currently docked View controls the Workspace buttons. Some times you dock nested Views inside the main View, and you would like that nested View to control the buttons and actions instead. This can easily be done with the `forwardWorkspaceActions` function. You can change the forwarding however you see fit, for example on focus or on click on some component inside the nested View.

```
class CustomerEditor : View() {
    override val root = hbox {
        val basicDataEditor = find<CustomerBasicDataEditor>()
        add(basicDataEditor)
        forwardWorkspaceActions(basicDataEditor)
        add(ContactListEditor::class)
    }
}
```

Modifying the default workspace

The default workspace only gives you basic functionality. As your application grows you will want to suplement the toolbar with more buttons and controls, and maybe a `MenuBar` above it. For small modifications you can augment it in the `onBeforeFunction` as we did above, but you will most probably want to subclass as the customizations become more advanced. The following code and image is taken from a real world CRM application:

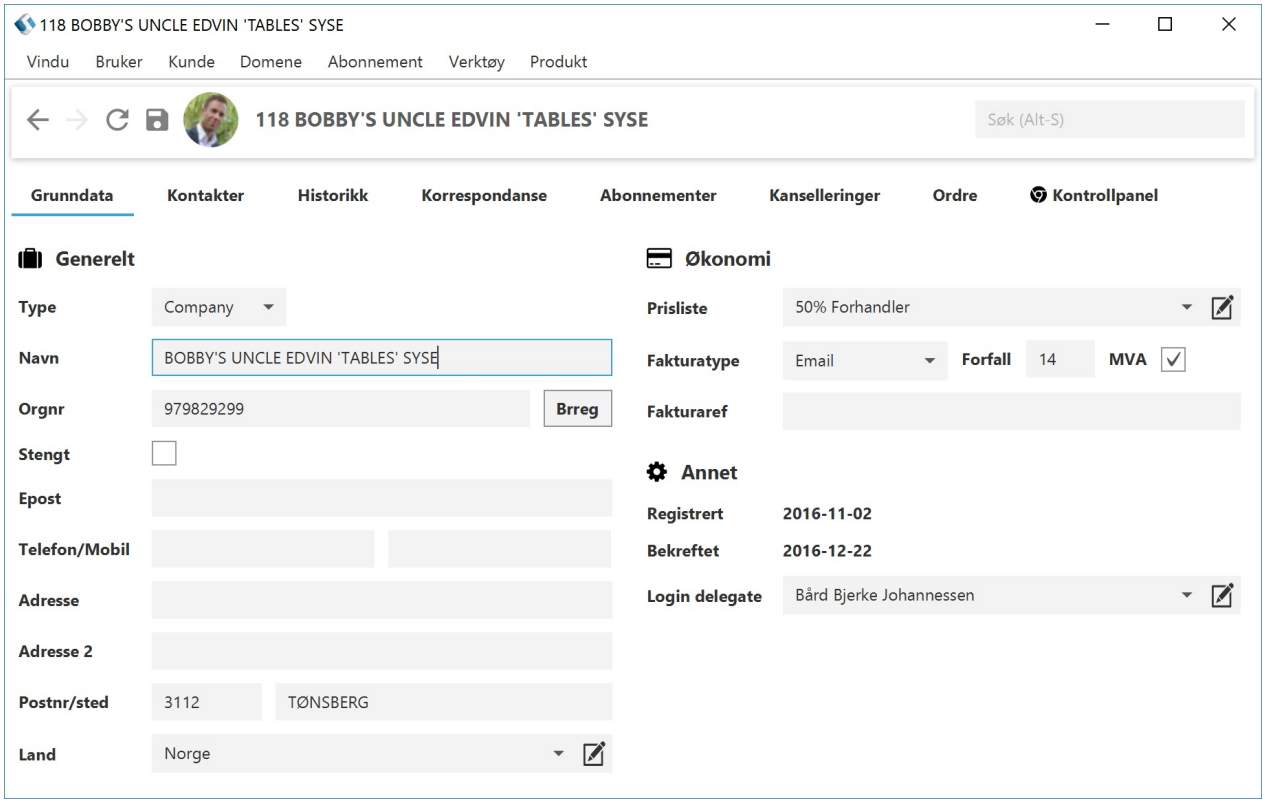
```
class CRMWorkspace : Workspace() {
    init {
        add(MainMenu::class)
        add(RestProgressBar::class)
        add(SearchView::class)
    }
}
```

The `CRMWorkspace` loads three other views into it. One providing a `MenuBar`, then the default `RestProgressBar` is added, and lastly a `SearchView` providing a search input field is added.

The Workspace has a pretty good idea about where to place whatever you add to it. For example, buttons will by default be added after the four default buttons, while other components are added to the far right of the ToolBar. The MenuBar is automatically added above the ToolBar, at the top of the screen.

Figure 16.3 shows how it looks in production, with a little bit of custom styling and a `CustomerEditor` docked into it. This application happens to be in Norwegian, and some of the information in the Customer card has been removed.

Figure 16.3



You will notice that the **Save** button is enabled in this View. This is because the `savable` property is bound to the dirty state property of the view model:

```
val model: CustomerModel by inject()
override val savable = model.dirty
```

When a customer is loaded, the **Save** button will stay disabled until an edit has been made. To save, we override the `onSave` function:

```
override fun onSave() {
    runAsync {
        customerController.save(customer.item)
    } ui { saved ->
        customer.update(saved)
    }
}
```

This particular `customerController.save` call will return the `Customer` from the server once it is saved. If the server made any changes to our customer object, they would have been reflected in the saved customer we got back. For that reason, we call `customer.update(saved)` which is function you get for free if you implement `JsonModel`. This makes sure that changes from the server is pushed back into the model. This is completely optional, and you might just want to do `customerController.save(customer.item)`.

Title and heading

When a view is docked, the title of the Workspace will match the title of that view. There is also a heading text in the workspace that by default shows the same text as the title. The heading can be overriden by assigning to the `heading` variable or binding to the `headingProperty` property. If you want to completely remove the heading, augment the workspace with `workspace.headingContainer.removeFromParent()` or just hide it. You can also put whatever nodes you want inside the heading container. You saw this trick in the CRM screenshot, where a Gravator icon was placed to the left of the customer name.

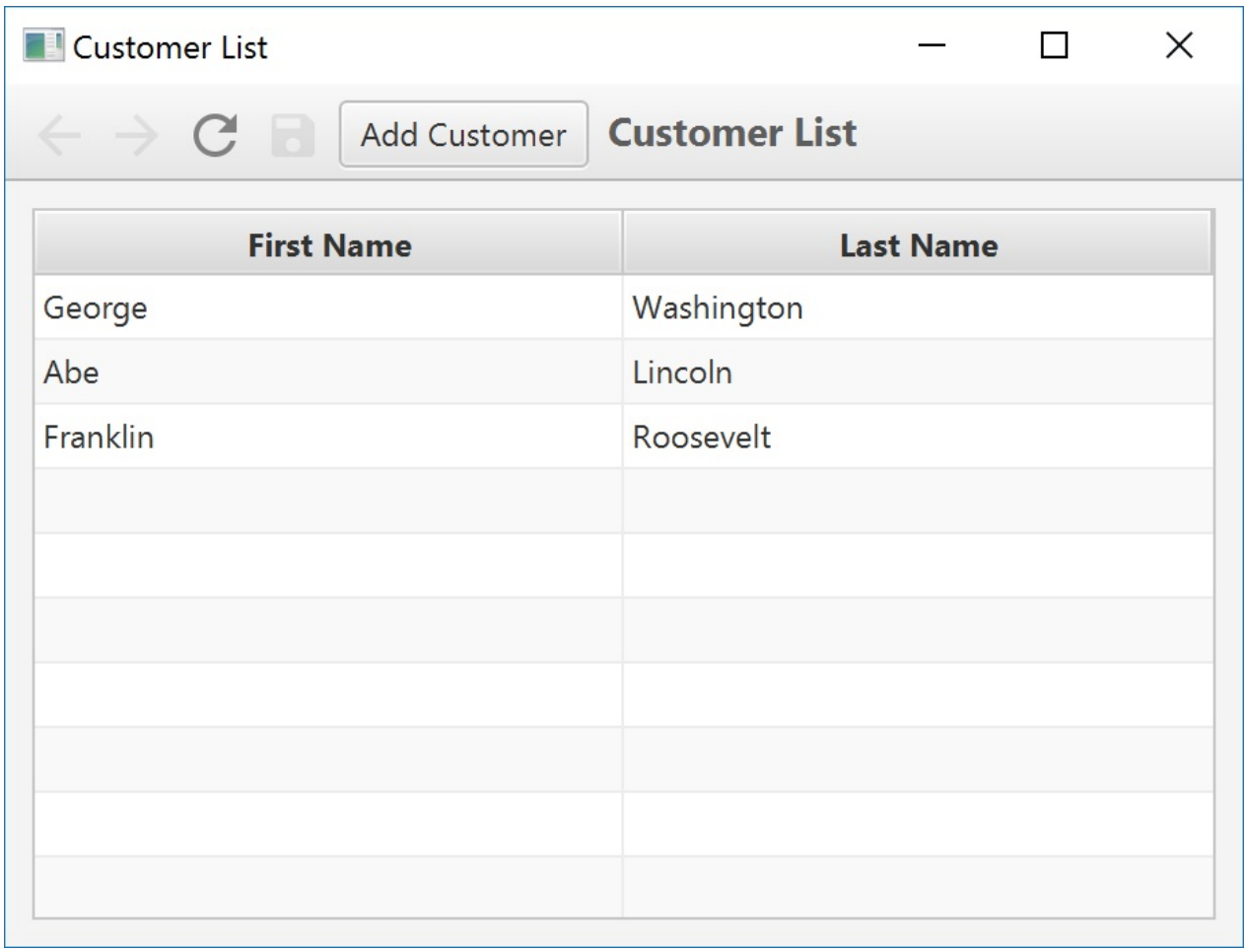
Dynamic elements in the ToolBar

Some views might need more buttons or functionality added to the ToolBar, but once you navigate away from the view it wouldn't make sense to keep them around. The Workspace will actually track whatever elements you add to it while a view is docked and remove those changes when the view is undocked. The perfect place to add these extra buttons would be the `onDock` call of the view.

Every `UIComponent` has a property called `workspace` which will point to the current Workspace for the current Scope. Let's add an "Add Customer" button to the Workspace whenever the `CustomerList` is docked:

```
override fun onDock() {
    with (workspace) {
        button("Add Customer").action {
            addCustomer()
        }
    }
}
```

The Workspace will now look like in Figure **16.4**

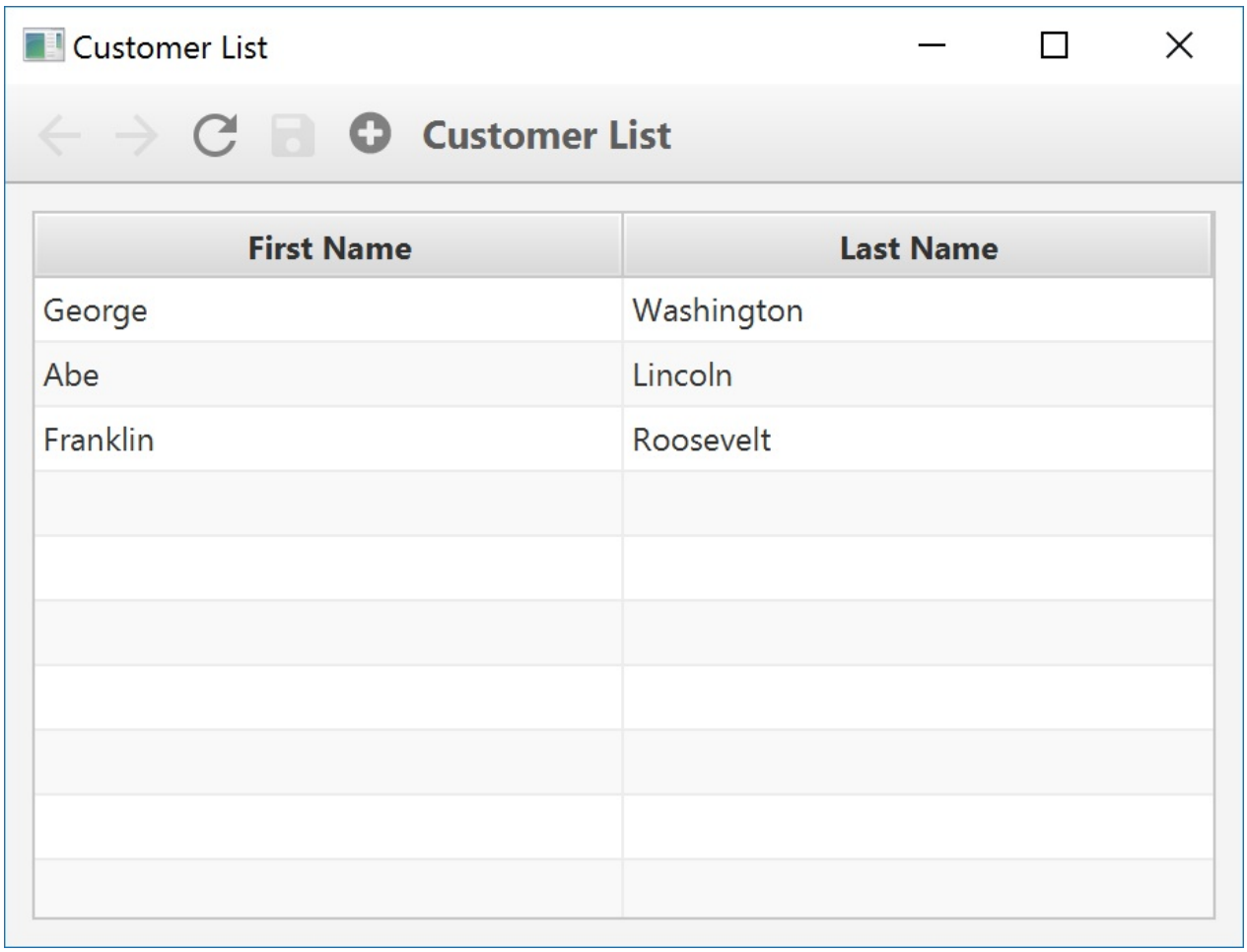


It looks like a default button. You can remove the border around the button by adding the `icon-only` css class to it. Optionally you can configure an icon for the graphic node if you like. The built in icons are svg shapes added in the built in `workspace.css` but feel free to add your icon in any way you see fit. Let's add an icon from the FontAwesomeFX library and make it look like the other buttons:

```
button("Add Customer") {
    addClass("icon-only")
    graphic = FontAwesomeIconView(PLUS_CIRCLE).apply {
        style {
            fill = c("#818181")
        }
        glyphSize = 18
    }
    action { addCustomer() }
}
```

In a real application you would use a css class so you don't need to configure the fill for every button you add. The result can be seen in Figure 16.5:

Figure 16.5



Navigating between docked views

Our Customer List is configured so that whenever you double click a customer you will be taken to an editor for that customer. The TableView binds the selected user to a `CustomerModel` view model object, and the action is performed like this:

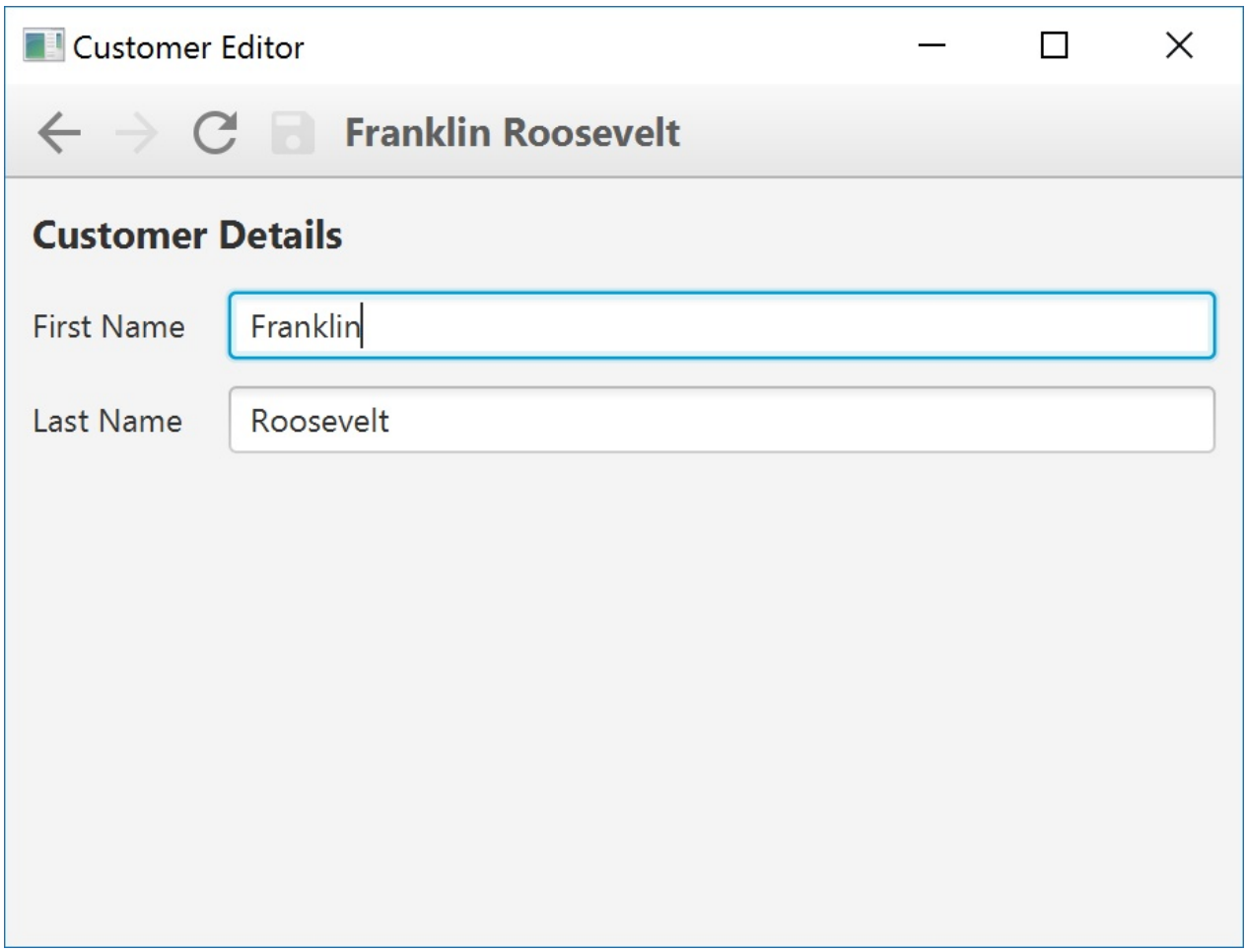
```
tableview(customers) {
    column("First Name", Customer::firstNameProperty)
    column("Last Name", Customer::lastNameProperty)
    bindSelected(model)
    onUserSelect { workspace.dock<CustomerEditor>() }
}
```

The only thing we need to do is actually dock the `CustomerEditor` when the user selects a row. Since the `CustomerEditor` will be looked up in the same scope we are currently in, it will have access to the selected customer as well:

```
class CustomerEditor : Fragment("Customer Editor") {
    val customer: CustomerModel by inject()
    override val savable = customer.dirty
    override val headingProperty = customer.fullName

    override val root = form {
        fieldset("Customer Details") {
            field("First Name") {
                textfield(customer.firstName)
            }
            field("Last Name") {
                textfield(customer.lastName)
            }
        }
    }

    override fun onSave() {
        customer.commit()
    }

    override fun onRefresh() {
        customer.rollback()
    }
}
```

The customer model is injected, and will contain the selected customer from the list. The `savable` property is bound to the `dirty` property of the model and the `headingProperty` is bound to a `StringBinding` called `fullName`, which concatinates the first and last names and updates whenever they are changed. The form fields bind to the name properties and lastly the `onSave` and `onRefresh` functions are implemented to react to the corresponding Workspace buttons.

Figure 16.6



We can see that the `title` and `heading` are indeed displaying separate information. Since we haven't made any edits yet, the **Save** button is disabled, while the **Refresh** button is available, and would roll back any changes made since the last commit.

The `back` button is enabled as well, and clicking it would navigate back to the Customer list. This is a very powerful feature which enables browser like navigation in your application with very little effort on your part. The Workspace keeps a navigation stack of configurable depth. By default it will contain 10 previously docked views. You can configure the `maxViewStackDepth` to change the number of views held in the navigation stack.

Alternative to overriding `onSave` and `onRefresh`

Some times you want to access an object in one of the workbench button actions but you want to avoid creating a variable for that object. Instead you can use the `whenSaved` and `whenRefreshed` callbacks, which can be configured from anywhere. Important: They are alternatives to `onSave` and `onRefresh` so you should only do one or the other. Let's say we want to refresh a TableView when the **Refresh** button is clicked. We can configure this inside the builder for the TableView:

```
tableview {
    whenRefreshed {
        asyncItems { controller.loadItems() }
    }
}
```

This is a handy alternative in some situations, but make sure you only choose one of the strategies.

Advanced scope navigation

When you leverage injected view models together with a navigation stack, some interesting challenges appear that need to be addressed. If you removed the **Back** button (`workspace.backButton.removeFromParent()`) or set the `maxViewStackDepth` to `0` you can disregard this particular challenge, but to leverage this powerful navigation paradigm, there are some things you need to think about.

Consider our prevous example with an injected `CustomerModel` that represents the currently selected customer in the `CustomerList` while also being used by the `CustomerEditor` to edit that same customer. Then let's assume that there is a way to search for a customer and edit it, perhaps using a `TextField` in the ToolBar of the Workspace as a search entry point. If you search for a new customer and go on to edit it, then navigate back to the previous customer editor, it would suddenly operate on the last customer you set in the `CustomerModel` . You can probably imagine the ensuing havoc.

Fortunately, the scoping support stretches far into the Workspace feature and provides some handy tools for this particular situation.

We need to find a way to contain the scope for the pair of CustomerList and CustomerEditor so they can work together while allowing other views to use the `CustomerModel` , but in a different scope. It's actually quite easy. Whenever you create a new `CustomerList` , also create a new Scope. If you were to do this manually, it would look something like this:

```
// Create a new scope, but keep the current workspace
val newScope = Scope(workspace)

// Find the CustomerList in the new scope
val customerList = find<CustomerList>(newScope)

// Dock the customerList in the workspace
workspace.dock(customerList)
```

Those three distinct operations can be performed in a single call:

```
workspace.dockInNewScope<CustomerList>()
```

When the CustomerList docks the CustomerEditor later on, it happens in this new scope. But what about the search field?

We would need to provide a separate scope for the `CustomerEditor` that should show the result of the search, but were we would also also need to inject the customer model containing the selected customer into the new scope. This following code is imagined inside the action that selects a customer from the search result:

```
fun editCustomer(customer: Customer) {
// Create a view model for the customer
val model = CustomerModel(customer)

// Create a new scope, but keep the current workspace
val newScope = Scope(workspace)

// Insert the customer model into the new scope
newScope.set(model)

// Find the CustomerEditor in the new scope
val editor = find<CustomerEditor>(newScope)

// Dock the editor
workspace.dock(editor)
}
```

That's a lot of steps. Fortunately, we can do that as well in a single call:

```
fun editCustomer(customer: Customer) {
    workspace.dockInNewScope<CustomerEditor>(CustomerModel(customer))
}
```

The `dockInNewScope` function takes a vararg list of injectable objects to insert into the new scope before looking up our CustomerEditor and docking it.

Separating scopes this way makes sure we can utilize injected view models without being afraid of other views stepping on our data. It is a pragmatic approach to an intricate problem. It also gives you a way of bleeding injectables into new scopes, should your use case require it.

Custom ViewStack optimizations

Some use cases might require you to make sure that the user cannot go back to a certain view after he has navigated to the prior view. You can remove your self from the View Stack on unDock like this:

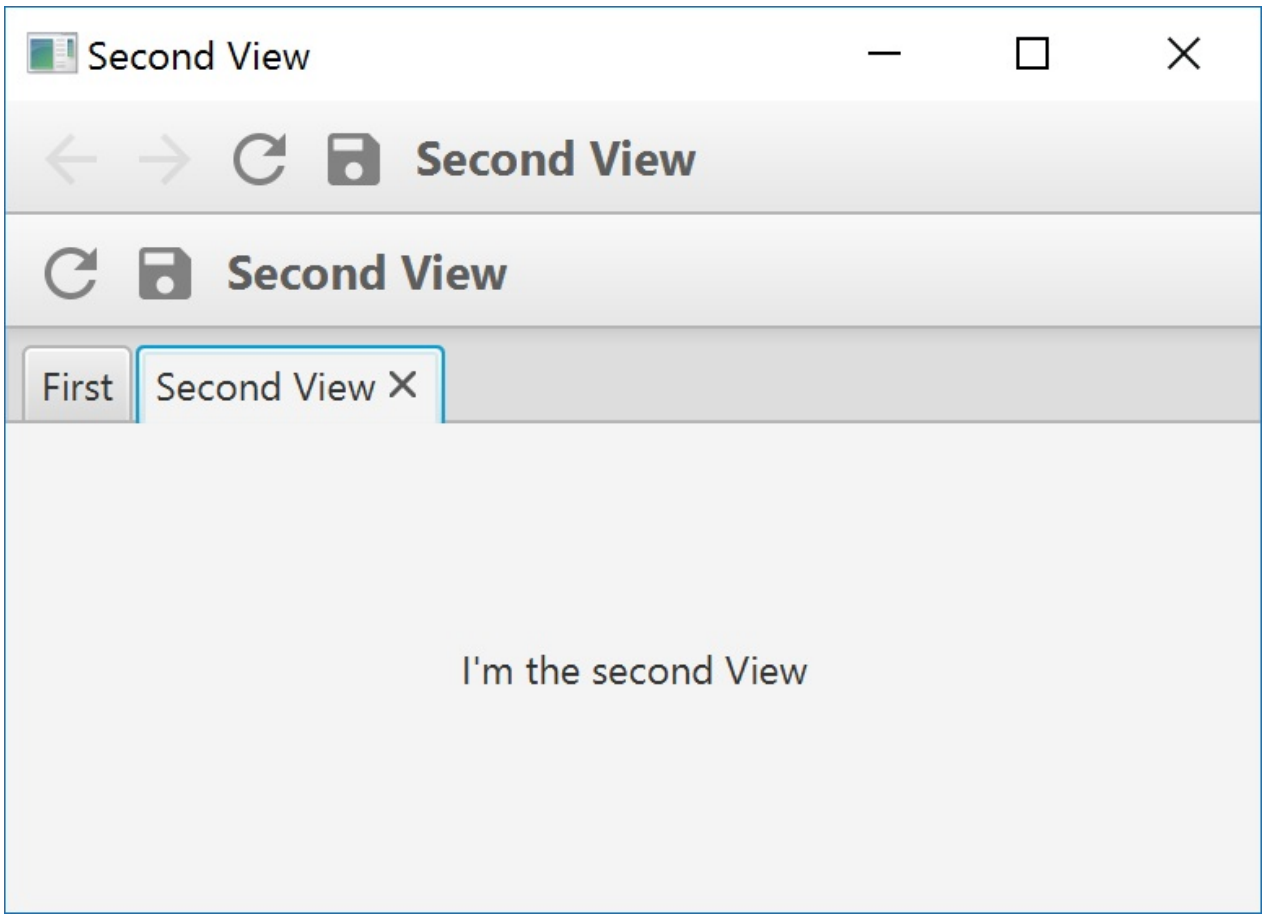
```
override fun onUndock() {
    workspace.viewStack.remove(this)
}
```

Docking multiple views in the editor area

The Workspace provides an alternative way to navigate between views. Instead of back and forward buttons, you can choose to dock multiple views inside a TabPane in the editor area. The Workspace has a `navigationMode` property that lets you change how the views are represented in the editor area. The default is `Workspace.NavigationMode.Stack`. The following example creates a tabbed Workspace that automatically docks two views inside it when it's created:

```
class TabbedWorkspace: Workspace("Tabbed Workspace", NavigationMode.Tabs) {
    init {
        dock<FirstView>()
        dock<SecondView>()
    }
}
```

Figure 16.7



> A Workspace in Tabs mode automatically hides the navigation buttons as they are no longer needed

You can create a starting point for this Workspace from a normal `App` class:

```
class TabbedWorkspaceApp : App(TabbedWorkspace::class)
```

The views docked inside the Workspace tabs will have their `onDock` function called whenever they are added and also when they are subsequently chosen as the active Tab. Correspondingly, the `onUndock` function is called whenever it is no longer the active Tab, as well as when it's removed from the TabPane using the close button on the tab.

You can control the closable state of a View docked inside the TabPane via the `closeable` property in `UIComponent`. It returns a `BooleanExpression` with the default value of `true` but you can override it to bind against another property or simply return another `SimpleBooleanValue(false)` to make it uncloseable. This example makes sure you cannot close the tab before the CustomerModel inside it is committed or rolled back:

```
class CustomerEditor : View("Customer Editor") {
    val customer: CustomerModel by inject()
    override val closeable = customer.dirty.not()
}
```

Drawer navigation

The Workspace has built in support for the Drawer control. You can access `workspace.leftDrawer` and `workspace.rightDrawer` to add items to each drawer. They will show up on either the left or right side whenever you have added one or more items to them.

Items added from a View in `onDock` will automatically be removed when the View is undocked. Items added directly in the Workspace subclass, from the `onBeforeShow` App callback or from any other place will stay until they are manually removed.

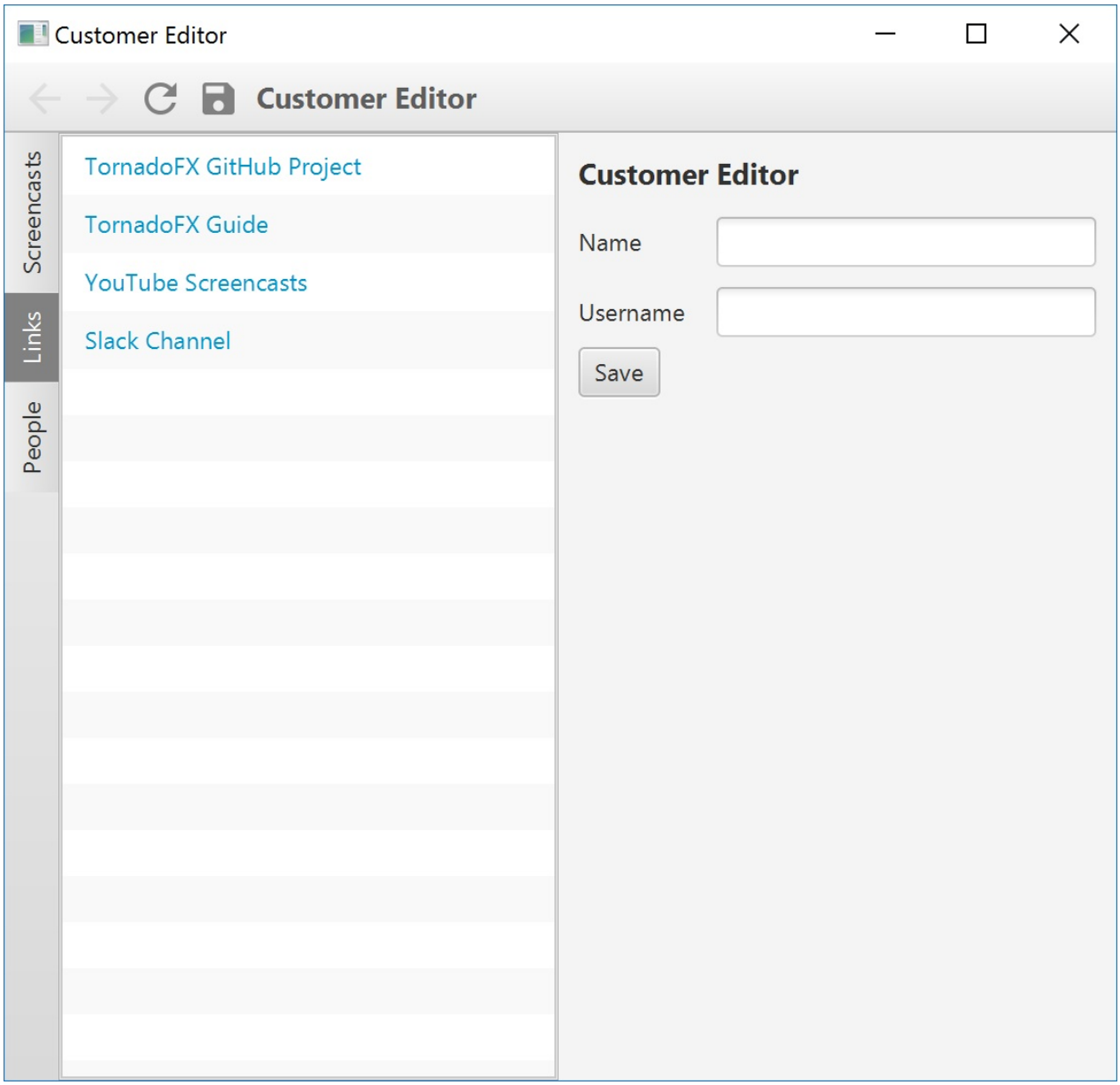
The combination of static and dynamic drawer items makes for a very powerful navigation and menu structure. Only your imagination is the limit!

The following example creates a customize Workspace primed with a docked Customer Editor in the editor area and the three drawer items we created in the Drawer chapter configured statically in the `leftDrawer` of the Workspace:

```
// A Form based View we will dock in the workspace editor area
class CustomerEditor : View("Customer Editor") {
    override val root = form {
        fieldset(title) {
            field("Name") { textfield() }
            field("Username") { textfield() }
            button("Save")
        }
    }
}

class DrawerWorkspace : Workspace() {
    init {
        // Dock the Customer Editor by default
        dock<CustomerEditor>()
    }

    init {
        // Add items to the left drawers
        with(leftDrawer) {
            item("Screencasts") {
                webview {
                    prefWidth = 470.0
                    engine.userAgent = iPhoneUserAgent
                    engine.load(TornadoFXScreencastsURI)
                }
            }
            item("Links") {
                listview(links) {
                    cellFormat { link ->
                        graphic = hyperlink(link.name).action {
                            hostServices.showDocument(link.uri)
                        }
                    }
                }
            }
            item("People") {
                tableview(people) {
                    column("Name", Person::name)
                    column("Nick", Person::nick)
                }
            }
        }
    }

    // Sample data and configuration omitted for this example
}
```

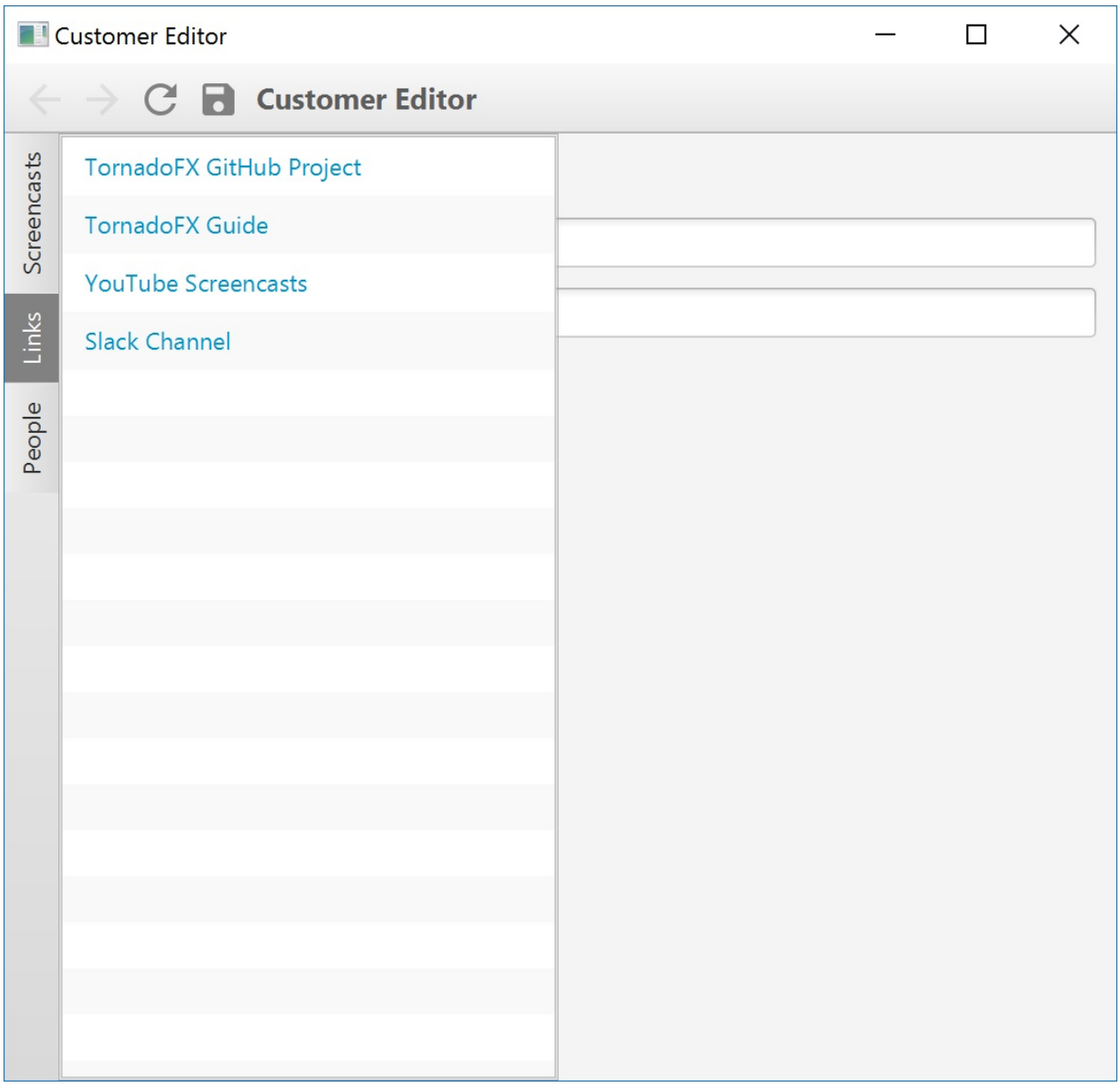
In Figure 16.8 we have expanded the *Links* drawer item. Notice how it pushes the Customer Editor to the right.

Figure 16.8



By right clicking the drawer and checking the `Floating drawers` option, the expanded drawer item content will instead float above the content, like in Figure 16.9:

Figure 16.9



This could be a good idea depending on the available space and the nature of the docked content. You can change the floating drawer mode in code as well, by setting `leftDrawer.floatingDrawers = true`.

Remember that Views can contribute drawer items programmatically in their `onDock` callback. Use this to provide extra tools for an advanced editor for example. They can easily communicate between each other using ViewModels. It is recommended to create a new scope to make it easier for these view parts to work in concert on shared data structures.

Vetoing navigation from the docked View

The currently docked View will receive a callback whenever the Back or Forward buttons of the Workspace is clicked. These functions are called `onNavigateBack` and `onNavigateForward`. The default implementation returns true to signal that the navigation

should proceed. You can however return false to stop the navigation and instead implement your own logic to decide what happens in the UI when one of the navigation buttons are clicked.

Layout Debugger

When you're creating layouts or working on CSS it some times help to be able to visualise the scene graph and make live changes to the node properties of your layout. The absolutely best tool for this job is definitely the Scenic View tool from FX Experience, but some times you just need to get a quick overview as fast as possible.

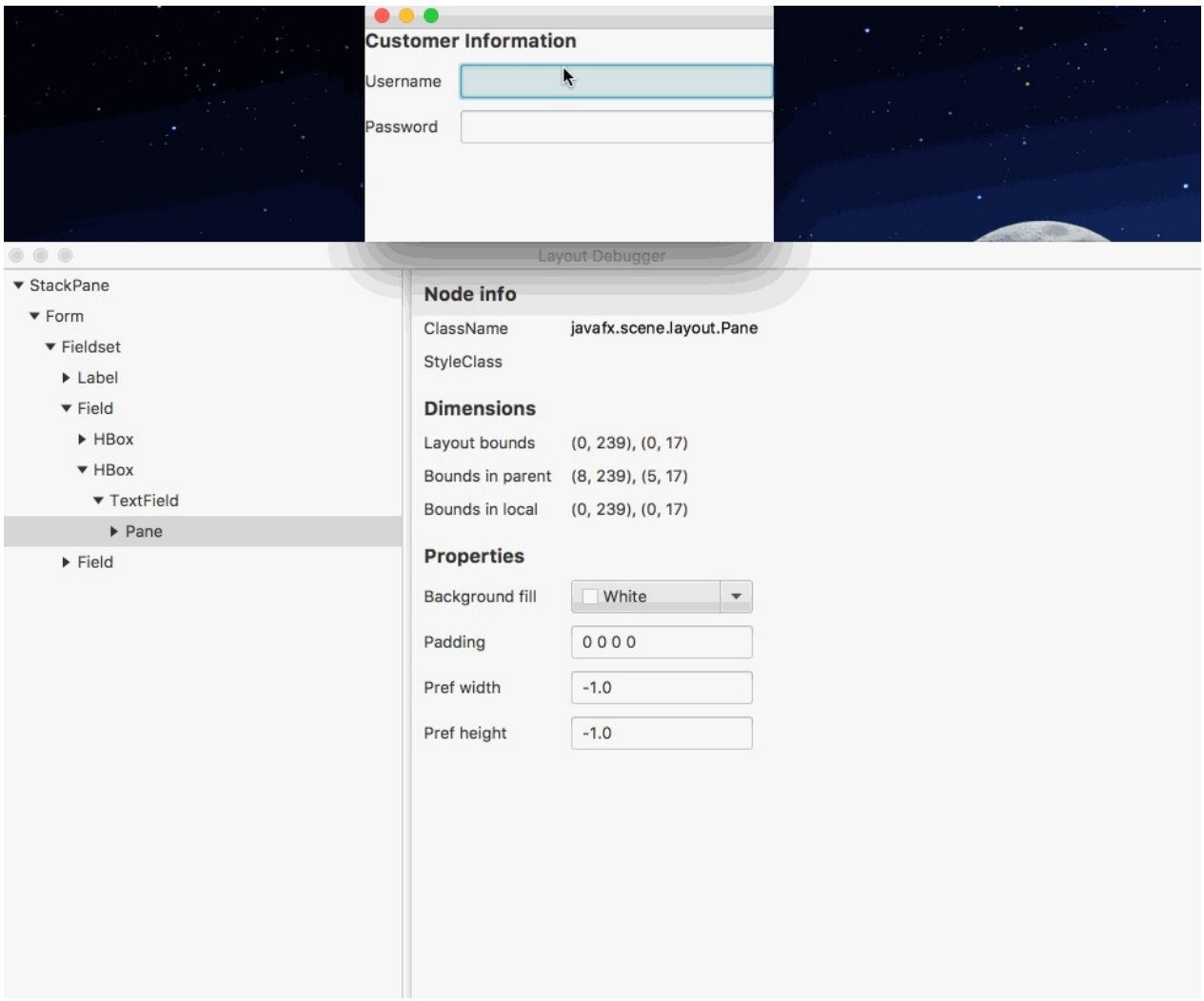
Debugging a scene

Simply hit **Alt-Meta-J** to bring up the built in debugging tool *Layout Debugger*. The debugger attaches to the currently active `Scene` and opens a new window that shows you the current scene graph and properties for the currently selected node.

Usage

While the debugger is active you can hover over any node in your View and it will be automatically highlighted in the debugger window. Clicking a node will also show you the properties of that node. Some of the properties are editable, like `backgroundColor`, `text`, `padding` etc.

When you hover over the node tree in the debugger, the corresponding node is also highlighted directly in the View.



Stop a debugging session

Close the debugger window by hitting `Esc` and the debugger session ends. You can debug multiple scenes simultaneously, each debugging session will open a new window corresponding to the scene you debug.

Configurable shortcut

The default shortcut for the debugger can be changed by setting an instance of `KeyCodeCombination` into `FX.layoutDebuggerShortcut`. You can even change the shortcut while the app is running. A good place to configure the shortcut would be in the `init` block of your `App` class.

Adding features

While this debugger tool is in no way a replacement for Scenic View, we will add features based on *reasonable* feature requests. If the feature adds value for simple debugging purposes and can be implemented in a small amount of code, we will try to add it, or better

yet, submit a pull request. Have a look at the source code to familiarise yourself with the tool.

Entering fullscreen

To enter fullscreen you need to get a hold of the current `stage` and call `stage.isFullScreen = true`. The primary stage is the active stage unless you opened a modal window via `view.openModal()` or manually created a stage. The primary stage is available in the variable `FX.primaryStage`. To open the application in fullscreen on startup you should override `start` in your app class:

```
class MyApp : App(MyView::class) {
    override fun start(stage: Stage) {
        super.start(stage)
        stage.isFullScreen = true
    }
}
```

In the following example we toggle fullscreen mode in a modal window via a button:

```
button("Toggle fullscreen") {
    setOnAction {
        with (modalStage) { isFullScreen = !isFullScreen }
    }
}
```

Logging

`Component` has a lazy initialized instance of `java.util.Logger` named `log`. Usage:

```
log.info { "Log message here" }
```

TornadoFX makes no changes to the logging capabilities of `java.util.Logger`. See the javadoc for more information.

Internationalization

TornadoFX makes it very easy to support multiple languages in your app.

Internationalization in Components

Each `Component` has access to a property called `messages` of type `ResourceBundle`. This can be used to look messages in the current locale and assign them to controls programmatically:

```
class MyView: View() {
    init {
        val helloLabel = Label(messages["hello"])
    }
}
```

> A label is programmatically configured to get it's text from a resource bundle

As well of the shorthand syntax `messages["key"]`, all other functions of the `ResourceBundle` class is available as well.

The bundle is automatically loaded by looking up a base name equal to the fully qualified class name of the `Component`. For a Component named `views.CustomerList`, the corresponding resource bundle in `/views/CustomerList.properties` will be used. All normal variants of the resource bundle name is supported, see ResourceBundle Javadocs for more information.

Internationalization in `FXML`

When an `FXML` file is loaded via the `fxml` delegate function, the corresponding `messages` property of the component will be used in exactly the same way.

```
<HBox>
    <Label text="%hello"/>
</HBox>
```

> The message with key `hello` will be injected into the label.

Default Global Messages

You can add a global set of messages with the base name `Messages` (for example `Messages_en.properties`) at the root of the class path.

Automatic lookup in parent bundle

When a key is not found in the component bundle, or when there is no bundle for the currrent component, the global resource bundle is consulted. As such, you might use the global bundle for all resources, and place overrides in the per component bundle.

Friendly error messages

In stead of throwing an exception when a key is not available in your bundle, the value will simply be `[key]`. This makes it easy to spot your errors, and your UI is still fully functional while you add the missing keys.

Configuring the locale

The default locale is the one retrieved from `Locale.getDefault()`. You can configure a different locale by issuing:

```
FX.locale = Locale("my-locale")
```

The global bundle will automatically be changed to the bundle corresponding to the new locale, and all subsequently loaded components will get their bundle in the new locale as well.

Overriding resource bundles

If you want to change the bundle for a component after it's been initialized, or if you simply want to load a spesific bundle without relying on the conventions, simply assign the new bundle to the `messages` property of the component.

If you want to use the overriden resource bundle to load `FXML`, make sure you change the bundle before you load the root view:

```
class MyView: View() {
    init { messages = ResourceBundle.getBundle("MyCustomBundle") }
    override val root = HBox by fxml()
}
```

> A manually overriden resource bundle is used by the `FXML` file corresponding to the View

The same technique can be used to override the global bundle by assigning to `FX.messages`.

Startup locale

You can override the default locale as early as the `App` class `init` function by assigning to `FX.locale`.

Controllers and Fragments as well

The same conventions are valid for `Controllers` and `Fragments`, since the functionality is made available to their common super class, `Component`.

Config settings and state

Saving application state is a common requirement for desktop apps. TornadoFX has several features which facilitates saving of UI state, preferences and general app configuration settings.

The `config` helper

Each component can have arbitrary configuration settings that will be saved as property files in a folder called `conf` inside the current program folder.

Below is a login screen example where login credentials are stored in the view specific config object.

```
class LoginScreen : View() {
    val loginController: LoginController by inject()
    val username = SimpleStringProperty(this, "username", config.string("username"))
    val password = SimpleStringProperty(this, "password", config.string("password"))

    override val root = form {
       fieldset("Login") {
           field("Username:") { textfield(username) }
           field("Password:") { textfield(password) }
           buttonbar {
                button("Login").action {
                    runAsync {
                        loginController.tryLogin(username.value, password.value)
                    } ui { success ->
                        if (success) {
                            with(config) {
                                set("username" to username.value)
                                set("password" to password.value)
                                save()
                            }
                            showMainScreen()
                        }
                    }
                }
            }
       }
    }

    fun showMainScreen() {
        // hide LoginScreen and show the main UI of the application
    }
}
```

> Login screen with credentials stored in the view specific config object

The UI is defined with the `TornadoFx` type safe builders, which basically contains a `form` with two `TextField`'s and a `Button`.
When the view is loaded, we assign the username and password values from the config object.
These values might be null at this point, if no prior successful login was performed.
We then bind the `username` and `password` to the corresponding `TextField`'s.

Last but not least, we define the action for the login button. Upon login, it calls the `loginController#tryLogin` function which takes the username and password from the `StringBindings` (which represent the input of the `TextField`s),
calls out to the service and returns true or false.

If the result is true, we update the username and password in the config object and calls save on it. Finally, we call `showMainScreen` which could hide the login screen and show the main screen of the application.

Please not that the above example is not a best practice for storing sensitive data, it merely illustrates how you can use the config object.

Data types and default values

`config` also supports other data types. It is a nice practice to wrap multiple operations on the config object in a `with` block.

```
// Assign to x, default to 50.0
var x = config.double("x", 50.0)

var showPrices = config.boolean("showPrices", boolean)

with (config) {
    set("x", root.layoutX)
    set("showPrices", showPrices)
    save()
}
```

Configurable config path

The `App` class can override the default path for config files by overriding `configBasePath`.

```
class MyApp : App(WelcomeView::class) {
    override val configBasePath: Paths.get("/etc/myapp/conf")
}
```

The path can also be relative, which means the path will be created inside the current working directory. By default, the base path is `conf`.

Override config path per component

By default, a file called `viewClass.properties` is created inside the `configBasePath`. This can be overriden per component:

```
class MyView : View() {
    override val configPath = Paths.get("some/other/path/myview.properties")
```

You can also create the View spesific config file below the `configBasePath`, which would make sense in most situations. You do this by accessing the App class through the `app` property of the View.

```
class MyView : View() {
    override val configPath = app.configBasePath.resolve("myview.properties")
```

Global application config

The App class also has a `config` property and a corresponding `configPath` property. By default, the configuration for the app class is named `app.config`. This can be overridden the same way you do for a View config.

The global configuration can be accessed by any component at any time in the life cycle of the application. Simply access `app.config` from anywhere to read or write your global configuration.

JSON configuration settings

The `config` object supports `JsonObject`, `JsonArray` and `JsonModel`. You set them using `config.set("key" to value)` and retrieve them using `config.jsonObject("key")`, `config.jsonArray("key")` and `config.jsonModel("key")`.

The `preferences` helper

As the `config` helper stores the information in a folder called `conf` per component (view, controller) the `preferences` helper will save settings into an OS specific way. In Windows systems they will be stored `HKEY_CURRENT_USER/Software/JavaSoft/...` on Mac os in `~/Library/Preferences/com.apple.java.util.prefs.plist` and on Linux system in `~/.java`. Where the `config` helper saves per component. The `preferences` helper is meant to be used application wide:

```
preferences("application") {
   putBoolean("boolean", true)
   putString("String", "a string")
}
```

Retrieving preferences:

```
var bool: Boolean = false
var str: String = ""
preferences("test app") {
    bool = getBoolean("boolean key", false)
    str = get("string", "")
}
```

The `preferences` helper is a TornadoFX builder around java.util.Preferences

JSON and REST

JSON has become the new standard for data exchange over HTTP. Working with JSON with the data types defined in `javax.json` is not hard, but a bit cumbersome. The TornadoFX JSON support comes
in two forms: Enhancements to the `javax.json` objects and functions and a specialized REST client that does HTTP as well as automatic conversion between JSON and your domain models.

To facilitate conversion between these JSON objects and your model objects, you can choose to implement the interface JsonModel and one or both of the functions `updateModel` and `toJSON`.

Later in this chapter we will introduce the REST client, but the JSON Support can also be used standalone. The REST client calls certain functions on JsonModel objects during the lifecycle of an HTTP request.

`updateModel` is called to convert a JSON object to your domain model. It receives a JSON object from which you can update the properties of your model object.

`toJSON` is called to convert your model object to a JSON payload. It receives a `JsonBuilder` where you can set the values of the model object.

```
class Person : JsonModel {
    val idProperty = SimpleIntegerProperty()
    var id by idProperty

    val firstNameProperty = SimpleStringProperty()
    var firstName by firstNameProperty

    val lastNameProperty = SimpleStringProperty()
    var lastName by lastNameProperty

    val phones = FXCollections.observableArrayList<Phone>()

    override fun updateModel(json: JsonObject) {
        with(json) {
            id = int("id")
            firstName = string("firstName")
            lastName = string("lastName")
            phones.setAll(getJsonArray("phones").toModel())
        }
    }

    override fun toJSON(json: JsonBuilder) {
```

```
        with(json) {
            add("id", id)
            add("firstName", firstName)
            add("lastName", lastName)
            add("phones", phones.toJSON())
        }
    }
}

class Phone : JsonModel {
    val idProperty = SimpleIntegerProperty()
    var id by idProperty

    val numberProperty = SimpleStringProperty()
    var number by numberProperty

    override fun updateModel(json: JsonObject) {
        with(json) {
            id = int("id")
            number = string("number")
        }
    }

    override fun toJSON(json: JsonBuilder) {
        with(json) {
            add("id", id)
            add("number", number)
        }
    }
}
```

> JsonModel with getters/setters and property() accessor functions to be JavaFX Property compatible

When you implement `JsonModel` you also get the `copy` function, which creates a copy of your model object.

Tornado FX also comes with special support functions for reading and writing JSON properties. Please see the bottom of Json.kt for an exhaustive list.

All the JSON retrieval functions accepts a vararg argument for the key in the JSON document. The first key available in the document will be used to retrieve the value. This makes it easier to work with slightly inconsistent JSON schemes or can be used as a ternary to provide a fallback value for example.

Configuring datetime

The `datetime(key)` function used to retrieve a `LocalDateTime` object from JSON will by default expect a value of "Seconds since epoch". If your external webservice expects "Milliseconds since epoch" instead,
you can either send `datetime(key, millis = true)` or configure it globally by setting `JsonConfig.DefaultDateTimeMillis = true`.

Generating JSON objects

The `JsonBuilder` is an abstraction over `javax.json.JsonObjectBuilder` that supports null values. Instead of blowing up, it silently dismisses the missing entry, which enables you to build your JSON object graph
more fluently without checking for nulls.

REST Client

The REST Client that makes it easy to perform JSON based REST calls. The underlying HTTP engine interface has two implementations. The default uses HttpURLConnection and there is also an implementation based on Apache HttpClient. It is easy to extend the `Rest.Engine` to support other http client libraries if needed.

To use the Apache HttpClient implementation, simply call `Rest.useApacheHttpClient()` in the `init` method of your App class and include the `org.apache.httpcomponents:httpclient` dependency in your project descriptor.

Configuration

If you mostly access the same api on every call, you can set a base uri so subsequent calls only need to include relative urls. You can configure the base url anywhere you like, but the `init` function of your `App` class is a good place to do it.

```
class MyApp : App() {
    val api: Rest by inject()

    init {
        api.baseURI = "http://contoso.com/api"
    }
}
```

Basic operations

There are convenience functions to perform `GET`, `PUT`, `POST` and `DELETE` operations.

```
class CustomerController : Controller() {
    val api = Rest by inject()

    fun loadCustomers(): ObservableList<Customer> =
        api.get("customers").list().toModel()
}
```

> CustomerController with loadCustomers call

So, what exactly is going on in the `loadCustomers` function? First we call `api.get("customers")` which will perform the call and return a `Response` object. We then call `Response.list()` which will consume the response and convert it to a `javax.json.JsonArray`. Lastly, we call the extension function `JsonArray.toModel()` which creates one `Customer` object per `JsonObject` in the array and calls `JsonModel.updateModel` on it. In this example, the type argument is taken from the function return type, but you could also write the above method like this if you prefer:

```
fun loadCustomers() = api.get("customers").list().toModel<Customer>()
```

How you provide the type argument to the `toModel` function is a matter of taste, so choose the syntax you are most comfortable with.

These functions take an optional parameter with either a `JsonObject` or a `JsonModel` that will be the payload of your request, converted to a JSON string.

The following example updates a customer object.

```
fun updateCustomer(customer: Customer) = api.put("customers/${customer.id}", customer)
```

If the api endpoint returns the customer object to us after save, we would fetch a JsonObject by calling `one()` and then `toModel()` to convert it back into our model object.

```
fun updateCustomer(customer: Customer) =
    api.put("customers/${customer.id}", customer).one().toModel<Customer>()
```

Query parameters

Query parameters needs to be URL encoded. The `Map.queryString` extension value will turn any map into a properly URL encoded query string:

```
val params = mapOf("id" to 1)
api.put("customers${params.queryString}", customer).one().toModel<Customer>()
```

This will call the URI `customers?id=1`.

Error handling

If an I/O error occurs during the processing of the request, the default Error Handler will report the error to the user. You can of course catch any errors yourself instead. To handle HTTP return codes, you might want to inspect the `Response` before you convert the result to JSON. Make sure you always call `consume()` on the response if you don't extract data from it using any of the methods `list()`, `one()`, `text()` or `bytes()`.

```
fun getCustomer(id: Int): Customer {
    val response = api.get("some/action")

    try {
        if (response.ok())
            return response.one().toModel()
        else if (response.statusCode == 404)
            throw CustomerNotFound()
        else
            throw MyException("getCustomer returned ${response.statusCode} ${response.
reason}")
    } finally {
        response.consume()
    }
}
```

> Extract status code and reason from `HttpResponse`

`response.ok()` is shorthand for `response.statusCode == 200`.

Authentication

Tornado FX makes it very easy to add basic authentication to your api requests:

```
api.setBasicAuth("username", "password")
```

To configure authentication manually, configure the `requestInterceptor` of the engine to add custom headers etc to the request. For example, this is how the basic authentication is implemented for the `HttpUrlEngine`:

```
requestInterceptor = { request ->
    val b64 = Base64.getEncoder().encodeToString("$username:$password".toByteArray(UTF
_8))
    request.addHeader("Authorization", "Basic $b64")
}
```

For a more advanced example of configuring the underlying client, take a look at how basic authentication is implemented in the `HttpClientEngine.setBasicAuth` function in Rest.kt.

Intercepting calls

You can for example show a login screen if an HTTP call fails with statusCode 401:

```
api.engine.responseInterceptor = { response ->
    if (response.statusCode == 401)
        showLoginScreen("Invalid credentials, please log in again.")
}
```

Setting timeouts

You can configure the read timeout for the default provider by using a `requestInterceptor` and casting the request to `HttpURLRequest` before yo operate on it.

```
api.engine.requestInterceptor = {
    (it as HttpURLRequest).connection.readTimeout = 5000
}
```

You can configure the `connectionTimeout` of the `HTTPUrlConnection` object above in the same way.

Connect to multiple API's

You can create multiple instances of the `Rest` class by subclassing it and configuring each subclass as you wish. Injection of subclasses work seamlessly. Override the `engine` property if you want to use another engine than the default.

Default engine for new Rest instances

The engine used by a new Rest client is configured with the `engineProvider` of the Rest class. This is what happens when you call `Rest.useApacheHttpClient`:

```
Rest.engineProvider = { rest -> HttpClientEngine(rest) }
```

> The `engineProvider` returns a concrete `engine` implementation that is given the current `Rest` instance as argument.

You can override the configured `engine` in a `Rest` instance at any time.

Proxy

A proxy can be configured either by implementing an interceptor that augments each call, or, preferably once per Rest client instance:

```
rest.proxy = Proxy(Proxy.Type.HTTP, InetSocketAddress("127.0.0.1", 8080))
```

Sequence numbers

If you do multiple http calls they will not be pooled and returned in the order you executed the calls. Any http request will return as soon as it is available. If you want to handle them in sequence, or even discard older results, you can use the `Response.seq` value which will contain a `Long` sequence number.

Progress indicator

Tornado FX comes with a HTTP ProgressIndicator View. This view can be embedded in your application and will show you information about ongoing REST calls. Embed the `RestProgressBar` into a *ToolBar* or any other parent container:

```
toolbar.add(RestProgressBar::class)
```

Dependency Injection

`View` and `Controller` are singletons, so you need some way to access the instance of a specific component. Tornado FX supports dependency injection, but you can also lookup components with the `find` function.

```
val myController = find(MyController::class)
```

When you call `find`, the component corresponding to the given class is looked up in a global component registry. If it did not exist prior to the call, it will be created and inserted into the registry before the function returns.

If you want to declare the controller referance as a field member however, you should use the `inject` delegate instead. This is a lazy mechanism, so the actual instance will only be created the first time you call a function on the injected resource. Using `inject` is always prefered, as it allows your components to have circular dependencies.

```
val myController: MyController by inject()
```

Third party injection frameworks

TornadoFX makes it easy to inject resources from a third party dependency injection framework, like for example Guice or Spring. All you have to do is implement the very simple `DIContainer` interface when you start your application. Let's say you have a Guice module configured with a fictive `HelloService`. Start Guice in the `init` block of your `App` class and register the module with TornadoFX:

```
val guice = Guice.createInjector(MyModule())

FX.dicontainer = object : DIContainer {
    override fun <T : Any> getInstance(type: KClass<T>)
        = guice.getInstance(type.java)
}
```

> The DIContainer implementation is configured to delegate lookups to `guice.getInstance`

To inject the `HelloService` configured in `MyModule`, use the `di` delegate instead of the `inject` delegate:

```
val MyView : View() {
    val helloService: HelloService by di()
}
```

The `di` delegate accepts any bean type, while `inject` will only allow beans of type `Injectable`, which includes TornadoFX's `View` and `Controller`. This keeps a clean separation between your UI beans and any beans configured in the external dependency injection framework.

Setting up for Spring

Above the setup for Guice is shown. Setting up for Spring, in this case using `beans.xml` as `ApplicationContext` is done as follows:

beans.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"
       xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context = "http://www.springframework.org/schema/context"
       xsi:schemaLocation = "http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:component-scan base-package="no.tornadofx.fxsample.springexample"/>
    <context:annotation-config/>
</beans>
```

This sets Spring up to scan for beans.

Application startup

```
class SpringExampleApp : App(SpringExampleView::class) {
    init {
        val springContext = ClassPathXmlApplicationContext("beans.xml")
        FX.dicontainer = object : DIContainer {
            override fun <T : Any> getInstance(type: KClass<T>): T = springContext.get
Bean(type.java)
        }
    }
}
```

This initialized the spring context and hooks it into tornadoFX via the `FX.dicontainer`. Now you can inject Spring beans like this:

```
val helloBean : HelloBean by di()
```

It is quite common in the Spring world to name a bean like so:

```
<bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld">
     <property name = "message" value = "Hello World!"/>
</bean>
```

The bean is then accessible using the `id`. This can be done in tornadoFX too:

```
class SpringExampleApp : App(SpringExampleView::class) {
    init {
        val springContext = ClassPathXmlApplicationContext("beans.xml")
        FX.dicontainer = object : DIContainer {
            override fun <T : Any> getInstance(type: KClass<T>): T = springContext.get
Bean(type.java)
            override fun <T : Any> getInstance(type: KClass<T>, name: String): T = spr
ingContext.getBean(type.java,name)
        }
    }
}
```

The second `getInstance` uses both the type of the bean and the id of the bean. Instantiating a bean is down as:

```
val helloBean : HelloBean by di("helloWorld")
```

Wizard

Some times you need to ask the user for a lot of information and asking for it all at once would result in a too complex user interface. Perhaps you also need to perform certain operations while or after you have requested the information.

For these situations, you can consider using a wizard. A wizard typically has two or more pages. It lets the user navigate between the pages as well as complete or cancel the process.

TornadoFX has a powerful and customizable Wizard component that lets you do just that. In the following example we need to create a new Customer and we have decided to ask for the basic customer info on the first page and the address information on the next.

Let's have a look at two simple input Views that gather said information from the user. The `BasicData` page asks for the name of the customer and the type of customer (Person or Company). By now you can probably `CustomerModel` guess how the `Customer` and `CustomerModel` objects look, so we won't repeat them here.

```
class BasicData : View("Basic Data") {
    val customer: CustomerModel by inject()

    override val root = form {
        fieldset(title) {
            field("Type") {
                combobox(customer.type, Customer.Type.values().toList())
            }
            field("Name") {
                textfield(customer.name).required()
            }
        }
    }
}

class AddressInput : View("Address") {
    val customer: CustomerModel by inject()

    override val root = form {
        fieldset(title) {
            field("Zip/City") {
                textfield(customer.zip) {
                    prefColumnCount = 5
                    required()
                }
                textfield(customer.city).required()
            }
        }
    }
}
```

By themselves, these views don't do much, but put together in a Wizard we start to see how powerful this input paradigm can be. Our initial Wizard code is only this:

```
class CustomerWizard : Wizard("Create customer", "Provide customer information") {
    val customer: CustomerModel by inject()

    init {
        graphic = resources.imageview("/graphics/customer.png")
        add(WizardStep1::class)
        add(WizardStep2::class)
    }
}
```

The result can be seen in Figure 21.1.

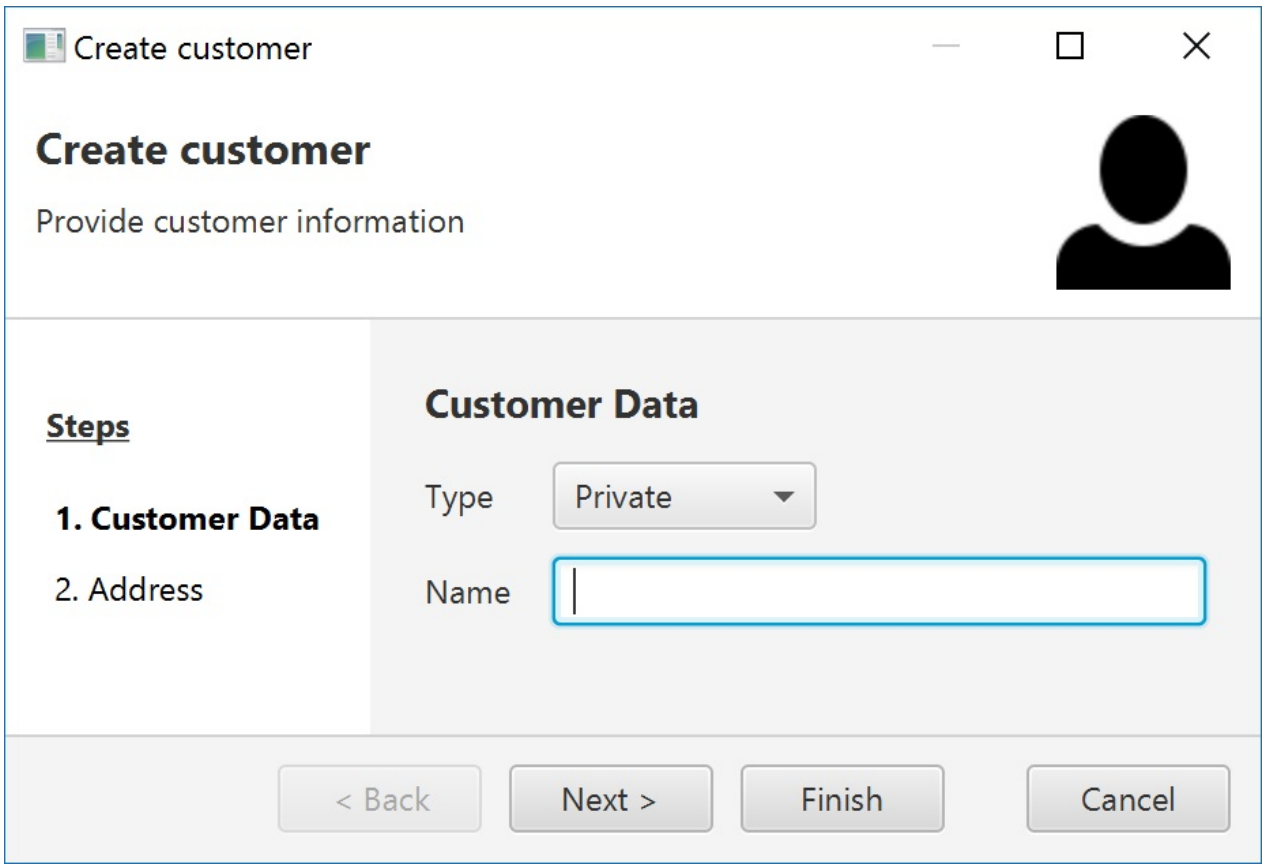


Figure 21.1

Just by looking at the Wizard the user can see what he will be asked to provide, how he can navigate between the pages and how to complete or cancel the process.

Since the Wizard itself is basically just a normal `View`, it will respond to the `openModal` call. Let's imagine a button that opens the Wizard:

```
button("Add Customer").action {
    find<CustomerWizard> {
        openModal()
    }
}
```

Page navigation

By default, the `Back` and `Next` buttons are available whenever there are more pages either previous or next in the wizard.

For `Next` navigation however, whether the wizard actually navigates to the next page is dependent upon the `completed` state of the current page. Every `View` has a `completed` property and a corresponding `isCompleted` variable you can manipulate.

When the `Next` or `Finish` button is clicked, the `onSave` function of the current page is called, and the navigation action is only performed if the current page's `completed` value is `true`. Every `View` is completed by default, that's why we can navigate to page number two without completing page one first. Let's change that.

In the `BasicData` editor, we override the `onSave` function to perform a partial commit of the `name` and `type` fields, because that's the only two fields the user can change on that page.

```
override fun onSave() {
    isComplete = customer.commit(customer.name, customer.type)
}
```

The commit function now controls the completed state of our wizard page, hence controller whether the user is allowed to navigate to the address page. If we try to navigate without filling in the name, we will be granted by the validation error message in Figure 21.2:

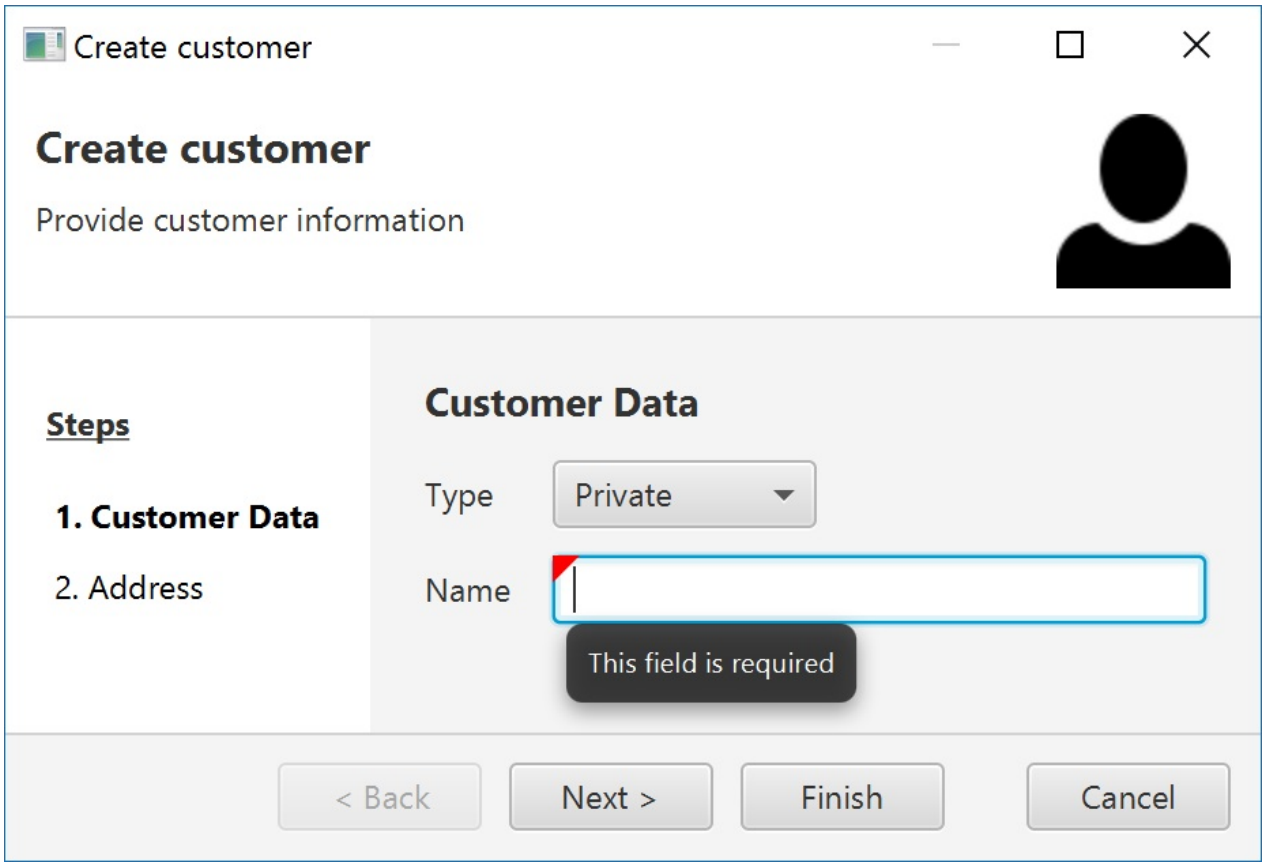


Figure 21.2

We could go on to do the same for the address editor, taking care to only commit the editable fields:

```
override fun onSave() {
    isComplete = customer.commit(customer.zip, customer.city)
}
```

If the user clicks the Finish button, the `onSave` function in the Wizard itself is activated. If the Wizard's `completed` state is true after the `onSave` call, the wizard dialog is closed, provided that the user calls `super.onSave()`. In such a scenario, the Wizard itself needs to handle whatever should happen in the `onSave` function. Another possibility is to configure a callback that will be executed whenever the wizard is completed. With that approach, we need access the completed customer object somehow, so we inject it into the wizard itself as well:

```
class CustomerWizard : Wizard() {
    val customer: CustomerModel by inject()
}
```

Let's revisit the button action that activated the wizard and add an `onComplete` callback that extracts the customer and inserts it into a database before it opens the newly created Customer object in a CustomerEditor View:

```
button("Add Customer").action {
    find<CustomerWizard> {
        onComplete {
            runAsync {
                database.insert(customer.item)
            } ui {
                workspace.dockInNewScope<CustomerEditor>(customer.item)
            }
        }
        openModal()
    }
}
```

Wizard scoping

In our example, both of the Wizard pages share a common view model, namely the `CustomerModel`. This model is injected into both pages, so it should be the same instance. But what if other parts of the application is already using the `CustomerModel` in the same scope we created the Wizard from? It turns out that this is not even an issue, because the `Wizard` base class implements `InjectionScoped` which makes sure that whenever you inject a `Wizard` subclass, a new scope is automatically activated. This makes sure that whatever resources we require inside the Wizard will be unique and not shared with any other part of the application.

It also means that if you need to inject existing data into a Wizard's scope, you must do so manually:

```
val wizard = find<MyWizard>()
wizard.scope.set(someExistingObject)
wizard.openModal()
```

Improving the visual cues

Un until now, the `Next` button was enabled whenever there was another page to navigate forward to. The `Finish` button was also always enabled. This might be fine, but you can improve the cues given to your users by only enabling those buttons when it would make sense to click them. By looking into the `Wizard` base class, we can see that the buttons are bound to the following boolean expressions:

```
open val canFinish: BooleanExpression = SimpleBooleanProperty(true)
open val canGoNext: BooleanExpression = hasNext
```

The `canFinish` expression is bound to the `Finish` button and the `canGoNext` expression is bound to the `Next` button. The `Wizard` class also includes some boolean expressions that are unused by default. Two of those are `currentPageComplete` and `allPagesComplete`. These expressions are always up to date, and we can use them in our `CustomerWizard` to improve the user experience.

```
class CustomerWizard : Wizard() {
    override val canFinish = allPagesComplete
    override val canGoNext = currentPageComplete
}
```

With this redefinition in place, the `Next` and `Finish` buttons will only be enabled whenever the new conditions are met. This is what we want, but we're not done yet. Remember how we only updated `isCompleted` whenever `onSave` was called? You might also remember that `onSave` was called whenever `Next` or `Finish` was clicked? It looks like we have ourselves a good old Catch22 situation here, folks!

The solution is however quite simple: Instead of evaluating the completed state on save, we will do it whenever a change is made to any of our input fields. We need to make sure that we supply the `autocommit` parameter to each binding in our ViewModel:

```
class CustomerModel : ItemViewModel<Customer>() {
    val name = bind(Customer::nameProperty, autocommit = true)
    val zip  = bind(Customer::zipProperty, autocommit = true)
    val city = bind(Customer::cityProperty, autocommit = true)
    val type = bind(Customer::typeProperty, autocommit = true)
}
```

The input fields in our wizard pages are bound to these properties, and whenever a change is made, the underlying Customer object will be updated. We no longer need to call `customer.commit()` in our `onSave` callback, but we do need to redefine the `complete` boolean expression in each wizard page.

Here is the new definition in the `BasicData` View:

```
override val complete = customer.valid(customer.name)
```

And here is the definition in the `AddressInput` View:

```
override val complete = customer.valid(customer.street, customer.zip, customer.city)
```

We bind the completed state of our wizard pages to an ever updating boolean expression which indicates whether the editable properties for that page is valid or not.

Remember to delete the `onSave` functions as we no longer need them. If you run the application with these changes you will see how much more expressive the Wizard becomes in terms of telling the user when he can proceed and when he can finish the process. Using this approach will also convey that any non-filled data is optional once the `Finish` button is enabled.

Here is the completely rewritten wizard and pages:

```
class CustomerWizard : Wizard() {
    val customer: CustomerModel by inject()

    override val canGoNext = currentPageComplete
    override val canFinish = allPagesComplete

    init {
        add(BasicData::class)
        add(AddressInput::class)
    }
}

class BasicData : View("Basic Data") {
    val customer: CustomerModel by inject()

    override val complete = customer.valid(customer.name)

    override val root = form {
        fieldset(title) {
            field("Type") {
                combobox(customer.type, Customer.Type.values().toList())
            }
            field("Name") {
                textfield(customer.name).required()
            }
        }
    }
}

class AddressInput : View("Address") {
    val customer: CustomerModel by inject()

    override val complete = customer.valid(customer.zip, customer.city)

    override val root = form {
        fieldset(title) {
            field("Zip/City") {
                textfield(customer.zip) {
                    prefColumnCount = 5
                    required()
                }
                textfield(customer.city).required()
            }
        }
    }
}
```

Styling and adapting the look and feel

There are many built in options you can configure to change the look and feel of the wizard. Common for them all is that they have observable/writable properties which you can bind to over just set in your wizard subclass. For each accessor below there will be a corresponding `accessorProperty`.

Modifying the steps indicator

Steps

The steps list is on the left of the wizard. It has the following configuration options:

Name	Description
showSteps	Set to `false` to remove the steps view completely
stepsText	Change the header from "Steps" to any desired String
showStepsHeader	Remove the header
enableStepLinks	Set to `true` to turn each step description into a hyperlink
stepLinksCommits	Set to `false` to no longer require that the current page is valid before navigating to the new page
numberedSteps	Set to `true` to add the index number before each step description

Navigation

You can change the text of the navigation buttons and control navigation flow with Enter:

Name	Description
backButtonText	Change the text of the `Back` button
nextButtonText	Change the text of the `Next` button
cancelButtonText	Change the text of the `Cancel` button
finishButtonText	Change the text of the `Finish` button
enterProgresses	Enter goes to next page when complete and finish on last page

Header area

Name	Description
showHeader	Set to `false` to remove the header
graphic	A node that will show up on the far right of the header

Structural modifications

The root of the `Wizard` class is a `BorderPane`. The header will be in the `top` slot, the steps are in the `left` slot, the pages are in the `center` slot and the buttons are in the `bottom` slot. You can change/hide/add styling and set properties to these nodes as you see fit to alter the design and layout of the Wizard. A good place to do this would be in the `onDock` callback of your wizard subclass. It is completely valid change the layout in any way you see fit, you can even remove the `BorderPane` and move the other parts into another layout container for example.