1 TornadoFX

1.1 Basic Structure

```
Components:
    -> Application: the entry point is "App" and specifies the
       initial "View"
        -> Stage:
        -> Scene:
            -> Model: The business code layer that holds core
               logic and data
            -> View: The visual display with various input and
               output controls
            -> Cotroller:The "middleman" mediating events
               between the Model and the View
            -> Fragment: The pop-ups
```

1.2 View

Many components are automatically maintained as singletons.

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

View contains display logic and layout of nodes, it's singleton.
When a View is declared, there must be a root property which can be any Parent type, and that will hold the View's content.
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")
    }
}
```

A BoderPane contains function TopView and BottomView, which can be embed by inject() delegate property to load other View. Then Each "child" View's root to assign them to the BorderPane.

```
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()
    //Here is a alter way: use find to insert a View
    val topView = find(TopView::class)
    val bottomView = find(BottomView::class)

    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")
}
```

Here is another alter way to use view builder to insert.

```
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")
```

```
}
```

1.3 Controllers

Controllers will help to finish a specific task in the back.

```
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()
            }
        }
    }
}

// this controller helps to write data
class MyController: Controller() {
    fun writeToDb(inputValue: String) {
    println("Writing␣$inputValue␣to␣database!")
    }
}
```

```
import javafx.collections.FXCollections
import tornadofx.*
class MyView : View() {
    val controller: MyController by inject()
    override val root = vbox {
        label("My␣items")
        listview(controller.values)
    }
}
// this controller helps to write data into local dataset.
```

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

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

Here is a fragment, which is a a way to do pop-up.

```
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")
}
```

This piece of code will open a new window.

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

```
open(), close() \\ Gives the access to open or close window.
findParentOfType(InternalWindow::class)
```

By using replaceWith, we can change windows.

```
button("Go␣to␣MyView2") {
    action {
        replaceWith(MyView2::class)
    }
}
```

We can even create some animation.

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

onDuck() and onUnDuck() will set action will View being replaced.

We can also passing parameters to Views, and we need specify configuring parameters for the target View.

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

Perform null check for the right parameters.

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

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.

1.4 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").toExternalForm())
```

However, in TornadoFX every Component has resources function!

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

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

val myJsonObject = resources.json("myobject.json")

val myJsonArray = resources.jsonArray("myarray.json")

val myStream = resources.stream("somefile")
```

Keyword with can be use to help assign components.

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

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.

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

class tView : View() {
    var firstNameField: TextField by singleAssign()
    var lastNameField: TextField by singleAssign()
    /*recommended you use the singleAssign() delegates to
 ensure the properties are only assigned once.*/
    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}")
            }
        }
    }
}
```

2 Builders for Basic Controls

2.1 Button

Button can optionally pass a text argument and a Button.() -¿ Unit lambda to modify its properties.

2.2 Label

label() extension function can be called 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

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

2.3 TextField

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

2.4 Password Mode

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

Check Box

```
val booleanProperty = SimpleBooleanProperty()

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

ComboBox

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

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

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

val selectedCity = SimpleStringProperty()

combobox(selectedCity, texasCities)
```

2.5 ToggleButton

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

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

2.6 Hyperlink and text

HyperLink

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

Text

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

Textflow

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

Tooltip

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

Shortcut

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

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