# Automated Testing

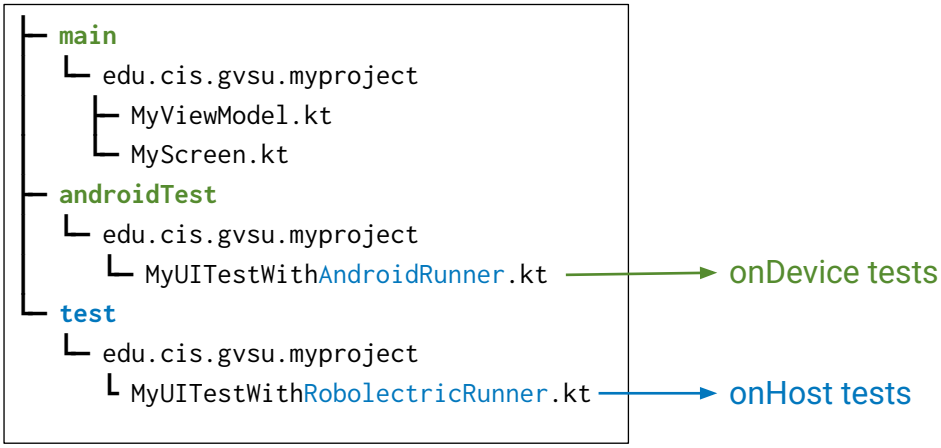
## Part 2: UI Testing

# UI Tests: What to tests

- Screen UI Tests: user interactions with a **single screen** (unit test)
- Navigation Tests: user moving through navigation flows involving **multiple screens** (integration test)



# Test Files Organization



# Library Setup

```
dependencies
{
    androidTestImplementation "androidx.test:runner:1.3.0"
    androidTestImplementation "androidx.test:rules:1.3.0"

    // UI testing with Compose
    androidTestImplementation("androidx.compose.ui:ui-test-junit4:1.10.6")

    // UI testing with Robolectric
    androidTestImplementation("org.robolectirc:robolectric:4.16")
}
```




```
Column {
    IconButton(onClick = onSettingsClick) {

    }
    Text(
        text = stringResource(“Ticket Purchase”),

     )
    OutlinedTextField()
    Button(onClick = {
        viewModel.purchaseTicket(numTickets.text.toString().toInt() ?: 0)
        },

        enabled = numTickets.text.toString().toIntOrNull() != null) {
        Text(text = stringResource(R.string.purchase))
     }
}
```


```
Column {
    IconButton(onClick = onSettingsClick,
       modifier = Modifier.testTag("settingsbutton")) {
    }
    Text(
        text = stringResource(“Ticket Purchase”),
        modifier = Modifier.testTag("windowtitle")
     )
    OutlinedTextField(modifier = Modifier.testTag("ticketinput"))
    Button(onClick = {
        viewModel.purchaseTicket(numTickets.text.toString().toInt() ?: 0)
        },
        modifier = Modifier.testTag("purchasebutton"),
        enabled = numTickets.text.toString().toIntOrNull() != null) {
        Text(text = stringResource(R.string.purchase))
     }
}
```

```
// In Test code:
onNodeWithTag(“windowtitle”)
```



# UI Test + AndroidRunner

```
@RunWith(AndroidJUnit4::class)   // file under androidTest/XXXXX.kt
class TicketScreenTest {
   @get:Rule
   val composeTestRule = createComposeRule()
   lateinit var viewModel: AppViewModel

   @Before
   fun setup() {
       // We don’t have to Mock the TicketSalesDao
       viewModel = mockk<AppViewModel>()
       composeTestRule.setContent { TicketPurchase(viewModel = viewModel) }
   }
   @Test fun firstTest() {    }
   @Test fun secondTest() {    }
}
```

# UI Test + RobolectricTestRunner

```
@RunWith(RobolectricTestRunner::class)   // file under test/XXXXX.kt
class TicketScreenTest {
   @get:Rule
   val composeTestRule = createComposeRule()
   lateinit var viewModel: AppViewModel

   @Before
   fun setup() {
       // We don't have to Mock the TicketSalesDao
       viewModel = mockk<AppViewModel>()
       composeTestRule.setContent { TicketPurchase(viewModel = viewModel) }
   }
   @Test fun firstTest() {    }
   @Test fun secondTest() {    }
}
```



# onNode____(): find a UI element

```
onNode(/* matching criteria */).some_operation_here()

onNodeWithTag("unique_tag_in_your_Composable_fun").some_operation_here()

onNodeWithContentDescription("Confirm").some_operation_here()

onNodeWithText("Confirm").some_operation_here()
```

# onNode functions: `assertXYZ() & performXYZ`

```
onNode___(/* matching criteria */).assertExists()
onNode___(/* matching criteria */).assertIsEnabled()
onNode___(/* matching criteria */).assertIsDisplayed()
onNode___(/* matching criteria */).assertIsToggleable()
onNode___(/* matching criteria */).assertDoesNotExist()
... many more ...
```

```
onNode___(/* matching criteria */).performTextInput(___)
onNode___(/* matching criteria */).performTouchInput(___)
onNode___(/* matching criteria */).performClick(___)
onNode___(/* matching criteria */).performMouseInput(___)
onNode___(/* matching criteria */).performScrollTo(___)
```

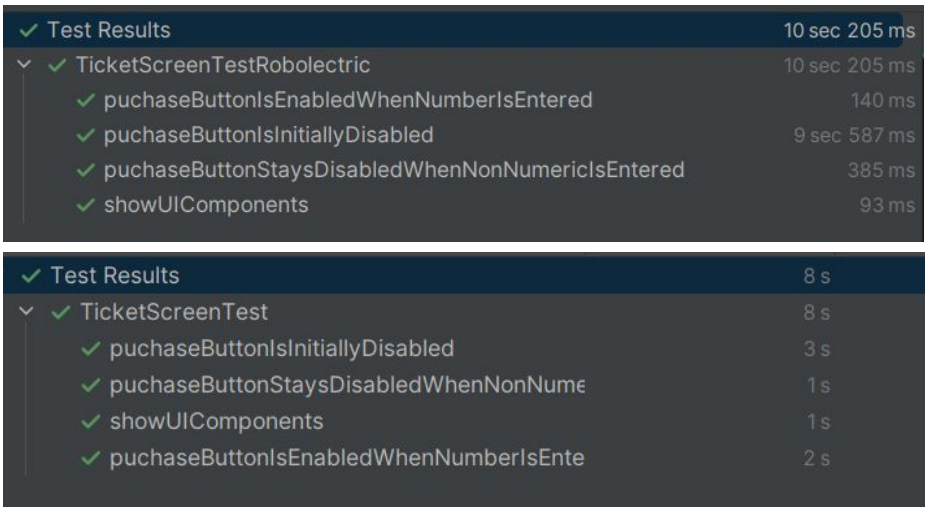
Compose UI Testing Cheatsheet



# UI Test + AndroidRunner


```
@Test
fun puchaseButtonIsInitiallyDisabled() {
   composeTestRule.onNodeWithTag("purchasebutton").assertIsNotEnabled()
}
@Test
fun puchaseButtonStaysDisabledWhenNonNumericIsEntered() {
   composeTestRule.apply {
       onNodeWithTag("purchasebutton").assertIsNotEnabled()
       onNodeWithTag("ticketinput").performTextInput("AB")
       onNodeWithTag("purchasebutton").assertIsNotEnabled()
   }
}
@Test
fun puchaseButtonIsEnabledWhenNumberIsEntered() {
   composeTestRule.apply {
       onNodeWithTag("purchasebutton").assertIsNotEnabled()
       onNodeWithTag("ticketinput").performTextInput("2")
       onNodeWithTag("purchasebutton").assertIsEnabled()
   }
}
```

# UI Test Runner: Android vs. Robolectric






# Mini Integration Test
# UI + ViewModel

```
@RunWith(RobolectricTestRunner::class)
class TicketScreenViewModelTestRobolectric {
   @get:Rule val composeTestRule = createComposeRule()
   lateinit var viewModel: AppViewModel
   @Before fun setup() {
       val dao = mockk<TicketSalesDao>()
       coEvery { dao.insert(any<Purchase>()) } just runs
       viewModel = spyk(AppViewModel(dao))
       composeTestRule.setContent { TicketPurchase(viewModel = viewModel) }
   }

   @Test fun clickBuyShallCallPurchaseTicketInViewModel() {
       composeTestRule.apply {
           onNodeWithTag("purchasebutton").assertIsNotEnabled()
           onNodeWithTag("ticketinput").performTextInput("2")
           onNodeWithTag("purchasebutton").apply {
               assertIsEnabled()
               performClick()
           }
           // Confirm that the operation to the DB is initiated
           verify { viewModel.purchaseTicket(2) }
       }
   }
}
```



# Stubbing Function Call Behavior

```
class XYZ {
  fun saveToFile(f: String): Boolean {
  }
}
```

```
coEvery {
    mockObj.saveToFile(any<String>())
} just runs

coEvery {
    mockObj.saveToFile(“a_file_name_here”)
} return true

coEvery {
    mockObj.saveToFile(“a_file_name_here”)
} throws Exception(“Storage is full”)
```

Use “`every`” for non-suspend function calls

# Setup Function (with comment)

```
lateinit var viewModel: AppViewModel

@Before fun setup() {
    // create a Mock/Stub
    val dao = mockk<TicketSalesDao>()

    // Override the behavior of the stub
    // Each time the suspendable fun .insert() is called, just assume it executes
    coEvery { dao.insert(any<Purchase>()) } just runs

    // Use the mocked object in creating the dependency
    // Attach a spy (monitor) on the viewmodel object
    viewModel = spyk(AppViewModel(dao))
    composeTestRule.setContent { TicketPurchase(viewModel = viewModel) }
}
```



# Purchase Ticket Action

```
@Test fun puchaseButtonShallInvokeViewModelFunction() {
    composeTestRule.apply {
        // Confirm the “Buy” button is disabled
        onNodeWithTag("purchasebutton").assertIsNotEnabled()
        // Enter 2 for number of purchase
        onNodeWithTag("ticketinput").performTextInput("2")
        onNodeWithTag("purchasebutton").apply {
            // Confirm the “Buy” button is enabled
            assertIsEnabled()
            // Click the “Buy” button
            performClick()
        }
        // Confirm that the operation to the DB is initiated
        verify { viewModel.purchaseTicket(2) }
    }
}
```

# UI Test + RoboRazzi

## (Twist of Paparazzi)



## Why RoboRazzi?

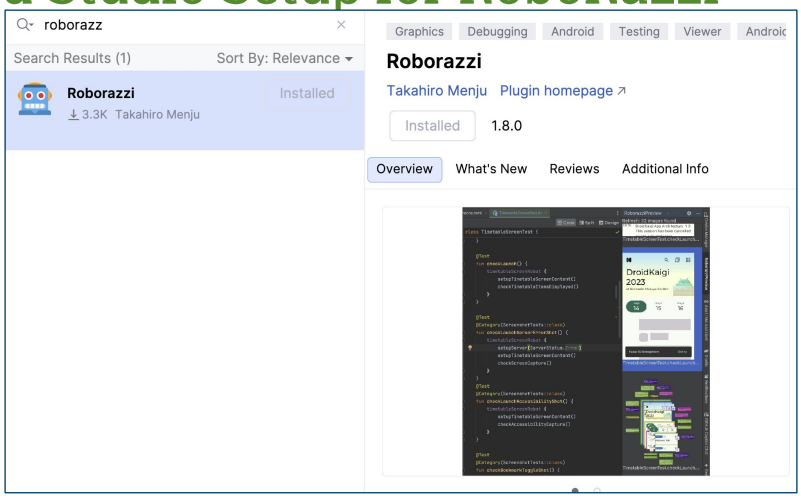
- Can be used for UI testing by comparing actual and desired screenshots
- Also useful for debugging your UI test cases
  - Grab multiple screenshots throughout to see what the “emulator” thinks

# Library Setup

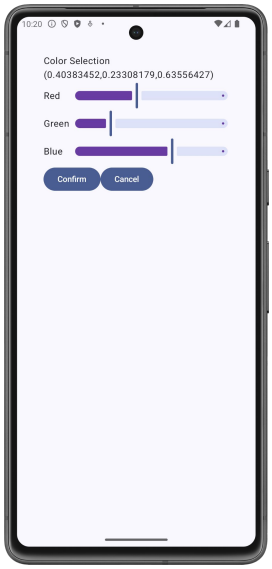
```
plugin {
    id("io.github.takahirom.roborazzi")
}
dependencies
{
    testImplementation("io.github.takahirom.roborazzi:roborazzi:1.8.0")
    testImplementation("io.github.takahirom.roborazzi:roborazzi-compose:1.8.0")
    testImplementation("io.github.takahirom.roborazzi:roborazzi-junit-rule:1.8.0")
}
```



# Android Studio Setup for RoboRazzi



# Screen to Test




```
var redSlider = rememberSliderState(valueRange = 0f..1f)
val thumbColor = remember {
   derivedStateOf {
       Color(redSlider.value, greenSlider.value, blueSlider.value)
   }
Row {
   Text( (${redSlider.value},${greenSlider.value},${blueSlider.value})")
   Row {
      Text("Red", modifier = Modifier.weight(1f))
      Slider(
         state = redSlider, modifier = Modifier.weight(6f)
             .testTag("redSlider"),
         colors = SliderDefaults.colors(activeTrackColor = thumbColor)
      )
  }
  // greenSlider and blueSlider are similar
}
```



# Test Class Setup

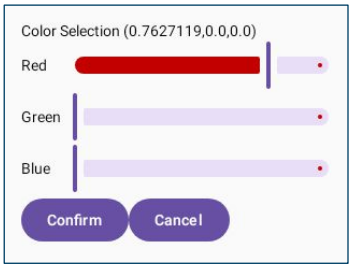
```
@RunWith(RobolectricTestRunner::class)
@GraphicsMode(GraphicsMode.Mode.NATIVE)
class MyTestClassHere {
    @Before
    fun setup() {
    }

    @Test
    fun sampleTest() {
        val uiElement = onNode(______)

        // Image save to ______/sampleTest.uiElement.png
        uiElement.captureRoboImage()
    }
}
```

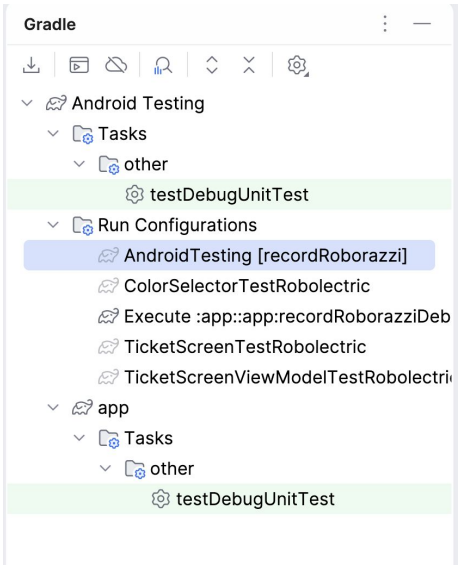
# Slider Behavior Test

```
fun redSliderAffectsThumbColor() {
   rule.apply {
       val slider = onNodeWithTag("redSlider")
       slider.performTouchInput {
           // Mouse down at the center of the slider
           down(center)
           // Move 30% to the right in 500 msec
           moveTo(position = center + percentOffset(x = 0.3f), delayMillis = 500)
           // Mouse up
           up()
       }
       slider.captureRoboImage()       // screenshot of the slider
       onRoot().captureRoboImage()     // screenshot of the whole screen
       onNode(hasRegex("""\(0\.\d{2,},0\.0,0\.0\)""".toRegex())).assertExists()
   }
}
```






# Gradle Task: recordRoborazzi




```
// After running the test
// Execute Gradle Task (if you don't find it
// under Run Configurations)
Tasks ⇒ other ⇒ recordRoborazzi
```