

9

Testing with JUnit, Mockito, MockK, and Compose

Activity 9.01 – developing with TDD

Solution

Perform the following steps to complete the activity:

1. Create an Android Studio project with an empty activity.
2. In `gradle/libs.versions.toml`, add the required dependencies:

```
[versions]
agp = "8.9.2"
kotlin = "2.0.21"
coreKtx = "1.16.0"
junit = "4.13.2"
junitVersion = "1.2.1"
espressoCore = "3.6.1"
lifecycleRuntimeKtx = "2.8.7"
activityCompose = "1.10.1"
composeBom = "2025.04.01"
mockk = "1.14.2"

[libraries]
androidx-core-ktx = { group = "androidx.core", name = "core-ktx",
```

```
version.ref = "coreKtx" }
junit = { group = "junit", name = "junit", version.ref = "junit" }
androidx-junit = { group = "androidx.test.ext", name = "junit",
    version.ref = "junitVersion" }
androidx-espresso-core = { group = "androidx.test.espresso",
    name = "espresso-core", version.ref = "espressoCore" }
androidx-lifecycle-runtime-ktx = { group = "androidx.lifecycle",
    name = "lifecycle-runtime-ktx",
    version.ref = "lifecycleRuntimeKtx" }
androidx-activity-compose = { group = "androidx.activity",
    name = "activity-compose", version.ref = "activityCompose" }
androidx-compose-bom = { group = "androidx.compose",
    name = "compose-bom", version.ref = "composeBom" }
androidx-ui = { group = "androidx.compose.ui", name = "ui" }
androidx-ui-graphics = { group = "androidx.compose.ui",
    name = "ui-graphics" }
androidx-ui-tooling = { group = "androidx.compose.ui",
    name = "ui-tooling" }
androidx-ui-tooling-preview = { group = "androidx.compose.ui",
    name = "ui-tooling-preview" }
androidx-ui-test-manifest = { group = "androidx.compose.ui",
    name = "ui-test-manifest" }
androidx-ui-test-junit4 = { group = "androidx.compose.ui",
    name = "ui-test-junit4" }
androidx-material3 = { group = "androidx.compose.material3",
    name = "material3" }
mockk = { group = "io.mockk", name = "mockk", version.ref = "mockk"
}
```

[plugins]

```
android-application = { id = "com.android.application",
    version.ref = "agp" }
kotlin-android = { id = "org.jetbrains.kotlin.android",
    version.ref = "kotlin" }
kotlin-compose = { id = "org.jetbrains.kotlin.plugin.compose",
    version.ref = "kotlin" }
```

3. Next, let's add the libraries in `app/build.gradle.kts`:

```
dependencies {  
    implementation(libs.androidx.core.ktx)  
    implementation(  
        libs.androidx.lifecycle.runtime.ktx  
    )  
    implementation(libs.androidx.activity.compose)  
    implementation(  
        platform(libs.androidx.compose.bom)  
    )  
    implementation(libs.androidx.ui)  
    implementation(libs.androidx.ui.graphics)  
    implementation(libs.androidx.ui.tooling.preview)  
    implementation(libs.androidx.material3)  
    testImplementation(libs.junit)  
    testImplementation(libs.mockk)  
    androidTestImplementation(libs.androidx.junit)  
    androidTestImplementation(  
        libs.androidx.espresso.core  
    )  
    androidTestImplementation(  
        platform(libs.androidx.compose.bom)  
    )  
    androidTestImplementation(  
        libs.androidx.ui.test.junit4  
    )  
    debugImplementation(libs.androidx.ui.tooling)  
    debugImplementation(  
        libs.androidx.ui.test.manifest  
    )  
}
```

4. Add the following strings to `res/values/strings.xml`:

```
<string name="item_x">Item %s</string>  
<string name="clicked_item_x">  
    Clicked Item %s
```

```
</string>  
<string name="press_me">Press Me</string>
```

5. In the app/androidTest folder, create a MainActivity test class, which will test clicking on a button with the **Press Me** label and then clicking on **Item 9**, and then assert that **Clicked Item 9** is displayed:

```
class MainActivityTest {  
  
    @get:Rule  
    val composeRule = createComposeRule()  
  
    @Test  
    fun verifyItemClicked() {  
        val scenario =  
            launch(MainActivity::class.java)  
            scenario.moveToState(Lifecycle.State.RESUMED)  
            composeRule.onNodeWithText(  
                getApplicationContext<Application>()  
                    .getString(R.string.press_me)  
            ).performClick()  
            composeRule.onNodeWithText(  
                getApplicationContext<Application>()  
                    .getString(  
                        R.string.item_x,  
                        "9"  
                    )  
            ).performClick()  
            composeRule.onNodeWithText(  
                getApplicationContext<Application>()  
                    .getString(  
                        R.string.clicked_item_x,  
                        "9"  
                    )  
            ).assertIsDisplayed()  
    }  
}
```

If you were to run the test at this point, it would fail because it cannot find the **Press Me** button.

6. In `MainActivity`, create a `@Composable` function called `ItemListScreen`, which will hold the user interface elements of the application. The screen will have a `LazyColumn` element:

```
@Composable
fun ItemListScreen(
    modifier: Modifier,
    itemCount: Int = 0,
    clickedItem: Int = 0,
    onPressMeClicked: () -> Unit,
    onItemClicked: (Int) -> Unit
) {
    LazyColumn(
        modifier = modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment =
            Alignment.CenterHorizontally
    ) {
        ...
    }
}
```

7. Inside the `LazyColumn` element, add the `Button` and `Text` elements and a list of items that will be clicked:

```
@Composable
fun ItemListScreen(
    ...
) {
    LazyColumn(
        ...
    ) {
        item {
            Button(onClick = onPressMeClicked) {
                Text(text = stringResource(id = R.string.press_me))
            }
            Text(
```

```

        text = stringResource(id = R.string.clicked_item_x,
            clickedItem),
        modifier = modifier
    )
}
items(itemCount) {
    val row = it + 1
    Column(modifier = Modifier.clickable(onClick = {
        onItemClickClicked(row)
    })) {
        Text(text = AnnotatedString(stringResource(id =
            R.string.item_x, row)))
    }
}
}
}

```

8. In the same file, add a `@Composable` function called `ItemList`. This will invoke `ItemListScreen` with a set of hardcoded values, which will be replaced later:

```

@Composable
fun ItemList(modifier: Modifier) {
    var clickedItem by remember { mutableStateOf(0) }
    ItemListScreen(
        modifier = modifier,
        itemCount = 0,
        clickedItem = clickedItem,
        onPressMeClicked = {},
        onItemClickClicked = {
            clickedItem = it
        }
    )
}

```

9. In the same file, modify the `onCreate` function to invoke `ItemList`:

```

override fun onCreate(savedInstanceState: Bundle?)
{
    super.onCreate(savedInstanceState)

```

```

enableEdgeToEdge()
setContent {
    Activity0901Theme {
        Scaffold(
            modifier = Modifier.fillMaxSize()
        ) { innerPadding ->
            ItemList(
                modifier = Modifier
                    .padding(innerPadding)
            )
        }
    }
}
}
}

```

If you run the application now, you will see the **Press Me** button and the **Clicked Item 0** text. If you run the defined test now, it will still fail.

10. Create the `RandomNumberGenerator` interface, which will contain a method to generate a random number:

```

interface RandomNumberGenerator {

    fun generateNumber(): Int

}

```

11. Create the `RandomNumberGeneratorImpl` class, which will implement `RandomNumberGenerator` and return -1 for the random number:

```

class RandomNumberGeneratorImpl() :
    RandomNumberGenerator
{

    override fun generateNumber(): Int {
        return -1
    }

}

```

12. In the test folder, write the `RandomNumberGeneratorImplTest` test, which will test that `generateNumber` returns the value 4:

```
class RandomNumberGeneratorImplTest {  
  
    private val randomNumberGenerator =  
        RandomNumberGeneratorImpl()  
  
    @Test  
    fun generateNumber_success() {  
        val result =  
            randomNumberGenerator.generateNumber()  
        assertEquals(4, result)  
    }  
}
```

If we run this test, it will fail because it returns -1 instead of 4.

13. Modify `RandomNumberGeneratorImpl` to have it return a number between 1 and 10:

```
class RandomNumberGeneratorImpl(  
    private val random: Random  
) : RandomNumberGenerator {  
  
    override fun generateNumber(): Int {  
        return random.nextInt(10) + 1  
    }  
}
```

14. Modify `RandomNumberGeneratorImplTest`. Have the `Random` object as a mockk instance and inject it into `RandomNumberGeneratorImpl`:

```
class RandomNumberGeneratorImplTest {  
  
    private val random = mockk<Random>()  
    private val randomNumberGenerator =  
        RandomNumberGeneratorImpl(random)  
  
    @Test  
    fun generateNumber_success() {
```



```
        every {
            random.nextInt(10)
        } returns 3
        val result =
            randomNumberGenerator.generateNumber()
        assertEquals(4, result)
    }
}
```

Now that the random feature works correctly, we will need to add it to the user interface, but we will also need to expose it to the tests.

15. Create the `RandomApplication` class, which will hold a reference to `RandomNumberGenerator` and initialize it with `RandomNumberGeneratorImpl` in the `onCreate` method:

```
open class RandomApplication : Application() {

    lateinit var randomNumberGenerator:
        RandomNumberGenerator

    override fun onCreate() {
        super.onCreate()
        randomNumberGenerator =
            RandomNumberGeneratorImpl(Random())
    }
}
```

16. In `AndroidManifest.xml`, add `RandomApplication` as a name in the application tag:

```
<application
    android:name=".RandomApplication"
    ...
>
```

17. In `MainActivity`, modify the `ItemList` function to introduce `RandomNumberGenerator` as a parameter:

```
@Composable
fun ItemList(
    modifier: Modifier,
    randomNumberGenerator: RandomNumberGenerator
```

```

    ) {
        var clickedItem by remember { mutableIntStateOf(0) }
        var itemCount by remember { mutableIntStateOf(0) }
        ItemListScreen(
            modifier = modifier,
            itemCount = itemCount,
            clickedItem = clickedItem,
            onPressMeClicked = {
                itemCount =
                    randomNumberGenerator.generateNumber()
            },
            onItemClick = {
                clickedItem = it
            }
        )
    }
}

```

18. In the same file, modify the `onCreate` function to add the `randomNumberGenerator` variable from `RandomApplication` as a parameter when calling `ItemList`:

```

override fun onCreate(savedInstanceState: Bundle?) {
    ...
    Activity0901Theme {
        Scaffold(
            modifier = Modifier.fillMaxSize()
        ) { innerPadding ->
            ItemList(
                modifier = Modifier
                    .padding(innerPadding),
                randomNumberGenerator = (
                    application as
                    RandomApplication
                ).randomNumberGenerator
            )
        }
    }
}

```

If you run the application now, then it has the desired requirements. However, if you run the test, it will still have an 80% likelihood of failing because the number generator will generate numbers less than 9.

19. In the `androidTest` folder, create the `TestRandomApplication` class, which will inherit from the `RandomApplication` class:

```
class TestRandomApplication : RandomApplication() {

    override fun onCreate() {
        super.onCreate()
        randomNumberGenerator =
            object : RandomNumberGenerator
            {
                override fun generateNumber(): Int {
                    return 10
                }
            }
    }
}
```

In the preceding snippet, we provide `randomNumberGenerator` a new implementation that will always return the number 10 to suit our testing requirements.

20. In the `androidTest` folder, create `RandomApplicationTestRunner`, which will run `TestRandomApplication` when the test is run on the device:

```
class RandomApplicationTestRunner :
    AndroidJUnitRunner()
{
    @Throws(Exception::class)
    override fun newApplication(
        cl: ClassLoader?,
        className: String?,
        context: Context?
    ): Application? {
        return super.newApplication(
            cl,
            TestRandomApplication::class.java.name,

```

```
        Context
    )
}
}
```

21. In `app/build.gradle.kts`, set `RandomApplicationTestRunner` as an instrumentation runner:

```
testInstrumentationRunner =
    "com.packt.android.RandomApplicationTestRunner"
```

22. If you run the test now, it should pass and have the following output:

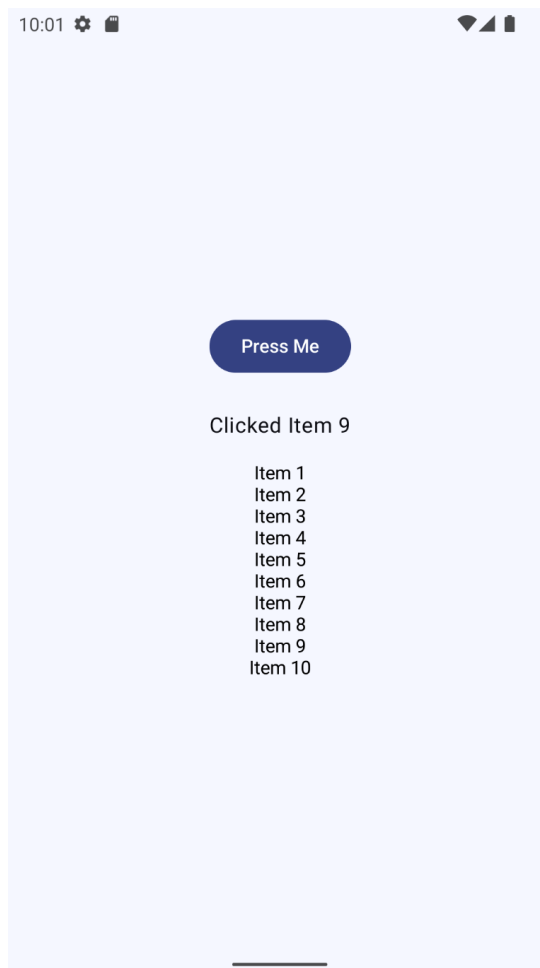


Figure 9.1 – Output of Activity 9.01