Automated Testing

Part 1: Unit Tests

What To Test

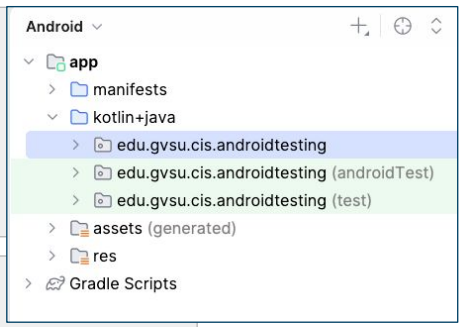
- Business Logic: verify correct operation of functions in ViewModel
- UI Logic: verify correct response when user interacts with the UI (of a single screen)
- Navigation Logic: verify correct screen flow between two screens

Where To Test

- Instrumented (“on-device”)
  - Tests which must run on Android device (physical or emulator)
  - Examples
    - End-to-end tests or integration tests
    - UI test
- Local (“on-host”)
  - Tests which can run on the development machine
  - Unit-tests or integration tests
  - UI test with Robolectric

Test Files Organization & Library Setup

```
├─ androidTest
│  ├─ test files which require Android runtime environment
│  └─ (tests which must run on the Android emulator)
└─ test
   └─ tests which can run on the Java Virtual Machine
```




```
dependencies {
    implementation(“libs.needed.for.non_test.environemt”)
    // libraries needed by androidTest files
    androidTestImplementation(“_____”)

    // libraries needed by test files
    testImplementation(“______”)
}
```

Scope of Test

	Dependencies	Execute On			Access Network
		JVM	Emuator	Device	
Unit Tests	Minimal	Yes	No	No	No
Component Tests	Multiple classes	Yes	Maybe	No	No
Feature Tests	External Components	Yes	Yes	Maybe	Mocked
Application Tests	Everything	No	Yes	Yes	Mocked

Available Tools

- JUnit (first release 1997), latest version 6.0.x (Feb 2026)
  - Basic building blocks of Java Testing
- **Robo(e)lectric**: run tests inside simulated Android environment on JVM (does not require running on an emulator)
  - Roborazzi plugins
- Espresso (for View-based apps)
- **Jetpack Compose Testing APIs**
- UI Automator
- Screenshot Library (@PreviewTest)

Part 1: Unit Tests

Scenario

- Simple ViewModel
- ViewModel with repository/data layer dependency
- ViewModel with coroutine/suspendable functions

Case 1: Simple ViewModel

```
class AppViewModel: ViewModel() {
    private val _availableTickets = MutableStateFlow(250)
    val availableTickets = _availableTickets.asStateFlow()

    fun purchaseTicket(count: Int) {
        if (count < 0) return
        if (count <= _availableTickets.value)
            _availableTickets.update { it - count }
        else
            throw IllegalStateException("Attempt to buy more than available")
    }
}
```

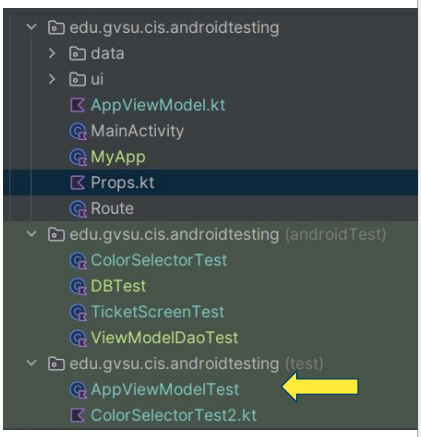
JUnit Quick Review

```
@RunWith(MyPreferredRunner::class)
class MyTestScaffold {
    @BeforeClass fun thisExecutesOnceBeforeAllTestsInThisFile() {     }
    @Before      fun thisExecutesBeforeEachTest() {     }

    @Test        fun firstTest() {     }
    @Test        fun secondTest() {     }

    @After       fun thisExecutesAfterEachTest() {     }
    @AfterClass  fun thisExecutesOnceAfterAllTestsInThisFile() {     }
}
```


Test Cases




```
// In the “test” folder
class AppViewModelTest {
    private lateinit var viewModel: AppViewModel
    @Before
    fun setup() {
        viewModel = AppViewModel()
    }

    @Test
    fun validPurchaseShouldDecrementAvailableTickets() {
        val oldCount = viewModel.availableTickets.value
        viewModel.purchaseTicket(2)
        assertThat(viewModel.availableTickets.value).isEqualTo(oldCount - 2)
    }
    @Test
    fun negativePurchaseCountShouldBeIgnored() {
        val oldCount = viewModel.availableTickets.value
        viewModel.purchaseTicket(-2)
        assertThat(viewModel.availableTickets.value).isEqualTo(oldCount)
    }
}
```

Test Cases For Verifying Exception

```
class AppViewModelTest {
    private lateinit var viewModel: AppViewModel
    @Before
    fun setup() {
        viewModel = AppViewModel()
    }

    @Test
    fun purchaseMoreThanAvailableShallThrowException() {
        val oldCount = viewModel.availableTickets.value
        assertThrows(IllegalStateException::class.java) {
            viewModel.purchaseTicket(oldCount + 1)
        }
    }
}
```


Case 2: ViewModel + Data Layer Dependency

```
class AppViewModel(val dao: TicketSalesDao): ViewModel() {
    private val _availableTickets = MutableStateFlow(250)
    val availableTickets = _availableTickets.asStateFlow()

    fun purchaseTicket(count: Int) {
        if (count < 0) return
        if (count <= _availableTickets.value)
            _availableTickets.update { it - count }
            viewModelScope.launch(Dispatchers.IO) {
                val p = Purchase(count = count, buyer = “Tee Cat”)
                dao.insert(p)
            }
        else
            throw IllegalStateException(“Attempt to buy more than available”)
    }
}
```

```
@Dao
interface TicketSalesDao {
    @Insert
    fun insert(x: Purchase)
}
data class Purchase(
    val count: Int,
    val buyer: String)
```

Test Objectives & Strategy

Objectives

- Verify that a valid purchase is saved to the local DB
- Verify that the (new) DB record holds the correct information

Strategy

- Option 1 (*slow*): Test using a real DB. Must keep a “TestDB” separate from “ProductionDB”. Inspect the DB table for the desired record(s)
- Option 2 (*slow*): Test using a DB emulator. Inspect the DB table for the desired record(s)
- Option 3 (*fast*): Verify that the operation to insert into DB is initiated.

Confirm X Flies from G. Rapids to Atlanta

Option A: Check at the departure gate in GRR airport that X is about to fly to ATL. (*and **trust** the airline does the rest*)

“about to call the insert() function”

Option B: Check at the arrival gate in ATL airport that X has arrived

“A new record has been inserted into the DB”

Movie Stunt Doubles ⇒ Test Doubles

Actor: Tom Holland
Movie: Spider Man

Image Credit: Reddit

Test Tools

- What we need: ability to “inspect/**spy**” into the internal data operations
- Test Doubles
  - A **Mock**/Fake object: a test proxy which has the same behavior as the real object
    - Fake DB: **destructive operations do not actually harm the real object**
  - A **Stub** object: is a mock whose behavior can be controlled at your preference
    - Fake Timer: **slow operations can be sped up**
    - Fake Authenticator: **easy to configure to accept, reject, password expiration, and other conditions**
  - A **Spy** object: is a test proxy which lets you monitor the operations (function calls) on a mocked object

Using Mockk (Pronounced: Mock-"Kay")

MockK vs. Mockito

	Mockito	MockK
Development Language	Java	Kotlin
Annotations	@Mock, @InjectMocks, @Spy	@MockK, @InjectMockks, @SpyK
Coroutine Support	❌	✔
Object mocking	✔	✔
Static method mocking	✔	✔

Library Setup: MockK

```
dependencies
{
    testImplementation("io.mockk:mockk:1.14.9")
    androidTestImplementation("io.mockk:mockk-android:1.14.9")
}
```


Creating a Spy using spyk<XXXX>

```
class AppViewModel(val dao: TicketSalesDao): ViewModel() {

   // in function body
   viewModelScope.launch {
      dao.insert(_____)
   }

}
```

```
class AppViewModelDoaTest {
   private lateinit var viewModel: AppViewModel
   private lateinit var spyDao: TicketSalesDao

   @Before
   fun setup() {
       spyDao = spyk<TicketSalesDao>()
       viewModel = AppViewModel(spyDao)
   }
}
```

`verify`: Confirm Spy Operations

```
// Using any<T>, slot<T>, and capture()
verify {
   // confirm printGreeting is called with a string
   helloSpy.printGreering(any<String>())

   // confirm it is called with a string of length at least 5
   val argPlaceHolder = slot<String>()
   helloSpy.printGreeting(capture(argPlaceHolder))
   val actualArg: String = actualArg.captured
   assert(actualArg.length > 5)
}
```


`coVerify`: Confirm Suspendable Operations

```
coVerify {
    // confirm a suspend function is called with a string & an Int args
    helloSpy.printToLogFile(any<String>(), any<Int>())
}
```

Test Cases

```
data class Purchase(
    val count: Int,
    val buyer: String)
```

```
class AppViewModelDoaTest {
    private lateinit var viewModel: AppViewModel
    private lateinit var spyDao: TicketSalesDao

    @Before
    fun setup() {
        spyDao = spyk<TicketSalesDao>()
        viewModel = AppViewModel(spyDao)
    }

    @Test
    fun validPurchaseSavedToDB() {
        viewModel.purchaseTicket(2)
        coVerify {
            spyDao.insert(any<Purchase>())
        }
    }
```

```
    @Test
    fun validPurchaseSavedWithCorrectCount() {
        viewModel.purchaseTicket(2)
        val actualPurchase = slot<Purchase>()
        coVerify {
            spyDao.insert(capture(actualPurchase))
        }
        assert(actualPurchase.captured.count == 2)
    }
}
```

*must use **coVerify** because `.insert()` is a `suspend` function*