Testing Strategies and App Distribution

Practical Exercise – Unit Testing, Integration Testing, and App Store Distribution

Introduction to Mobile Testing and Deployment

Course: Mobile Application Development (Chapter 9)
Level: Technical University Students
Duration: 8–10 hours self-study + 1.5–2 hours online discussion
Format: Individual Assignment with Testing Framework Implementation (6–8 pages)

Exercise Overview

In this exercise, you will master comprehensive testing methodologies for mobile applications: understanding the testing pyramid and unit/integration/UI testing strategies, implementing automated tests using platform-specific frameworks (Espresso, XCTest, Detox), establishing test coverage metrics and test-driven development (TDD) practices, and mastering the complete app distribution workflow across App Store Connect and Google Play Console including compliance, app signing, versioning, and store optimization.

Key Learning Outcomes

- Understand testing pyramid, unit testing, integration testing, and UI testing strategies
- Master platform-specific testing frameworks (JUnit, Mockito, Espresso for Android; XCTest, XCUITest for iOS; Detox for cross-platform)
- Implement test-driven development (TDD) practices and establish testing culture
- Achieve meaningful test coverage (>70%) with focused, maintainable tests
- Understand app signing, certificates, provisioning profiles, and keystore management
- Master Google Play Console submission process: app store listing, APK/bundle signing, rollout strategy
- Master App Store Connect submission: TestFlight beta testing, review guidelines, compliance requirements

- Implement continuous integration/continuous deployment (CI/CD) pipelines for testing and distribution
- Understand compliance policies, security requirements, and store optimization
- Conduct performance testing and user acceptance testing (UAT) before release

---

Part A: Testing Fundamentals and Architecture

Task 1: Testing Pyramid and Test Strategy (Self-Study: 2–2.5 hours)

Objective: Master the testing pyramid, test classification, and strategic test planning.

Instructions:

1. Testing Pyramid Overview

The testing pyramid defines the ideal distribution of tests across different levels:

```
    △
   /|\
  / | \
 /  |  \  UI Tests (E2E) — 5-10%
/   |   \ (Slow, Fragile, Expensive)
```

/ |
/|
/ |
/ Integration \ Integration Tests — 15-25%
\ Tests / (Medium speed, moderate cost)
\ (DAO, /
\ Service) /
_

///Unit Testing/ Unit Tests — 60-75%\ (Fast, \ (Fast, Cheap, Stable)\ Isolated)_____

Unit Tests (60-75%)

- Test individual functions, methods, classes in isolation
- Run locally without device/emulator
- Complete in milliseconds

- Should NOT depend on external services, file systems, or databases
- Use mocking and stubbing for dependencies
- Examples: Calculating discount amount, validating email format, parsing JSON

Integration Tests (15-25%)

- Test interaction between multiple components (e.g., DAO + Repository + ViewModel)
- Use real databases (in-memory), file systems
- Test API client + data parsing
- Complete in seconds
- Examples: Database save/retrieve, API response parsing, data layer interaction

UI/End-to-End Tests (5-10%)

- Test complete user workflows from UI perspective
- Slow and fragile; require device/emulator
- Test actual user interactions (tap, scroll, input)
- Complete in seconds to minutes
- Examples: Login flow, place order workflow, navigation between screens

2. Test Categories and Examples

Functional Tests

- Verify correct behavior
- Example: "When user clicks submit with valid data, account is created"

Regression Tests

- Ensure previously working features still work
- Run after each code change
- Build test library as you fix bugs

Acceptance Tests

- Verify business requirements
- Often written by QA/business
- Example: "User can purchase item with credit card within 5 seconds"

Performance Tests

- Verify speed under load
- Example: "List loads within 500 ms"

3. Test-Driven Development (TDD) Cycle

Red Phase: Write test (FAILS)
↓
Green Phase: Write minimal code (test PASSES)
↓
Refactor: Improve code (test still PASSES)
↓
Repeat: Write next test

Benefits:

- Forces clear thinking about requirements
- Tests act as specifications
- Encourages modular design
- Reduces debugging time
- Higher confidence in changes

4. Testing Strategy Framework

Define Testing Priorities:

Feature Complexity	Risk Level	Unit Tests	Integration Tests	UI Tests
Simple utility	Low	✓	✗	✗
Data validation	Low	✓	✓	✗
Payment logic	Critical	✓✓	✓	✓
Complex workflow	High	✓	✓	✓
Login/Auth	Critical	✓✓	✓	✓
External API	High	✓	✓	✗

Test Coverage Targets:

- Overall: >70% code coverage
- Critical paths: 90%+
- Utilities: 80%+
- UI layer: 20-30% (lower priority)

Deliverable:

- [ ] Testing pyramid diagram with percentages
- [ ] Four test category examples with descriptions
- [ ] TDD cycle explanation with 2 concrete examples
- [ ] Testing strategy framework tailored to mobile app

Task 2: Unit Testing Framework and Patterns (Self-Study: 2–2.5 hours)

Objective: Master unit testing frameworks, mocking, and test organization.

Instructions:

1. Unit Testing Frameworks

Android (JUnit 4 + Mockito):

import org.junit.Test
import org.junit.Before
import org.mockito.Mock
import org.mockito.Mockito.*

class UserValidatorTest {
private lateinit var validator: UserValidator

```
@Before
fun setup() {
    validator = UserValidator()
}

@Test
fun testValidEmailFormat() {
    // Arrange
    val email = "user@example.com"

    // Act
    val result = validator.isValidEmail(email)

    // Assert
    assertTrue(result)
}

@Test
fun testInvalidEmailMissingAtSign() {
    val email = "userexample.com"
    val result = validator.isValidEmail(email)
    assertFalse(result)
}

@Test
fun testPasswordMinimumLength() {
    val password = "short"
    val result = validator.isValidPassword(password)
    assertFalse(result) // Should be at least 8 chars
```

```
}

@Test
fun testPasswordWithAllRequirements() {
    val password = "SecurePass123!"
    val result = validator.isValidPassword(password)
    assertTrue(result)
}

}
```

iOS (XCTest):

import XCTest
@testable import MyApp

class UserValidatorTests: XCTestCase {
var validator: UserValidator!

```
override func setUp() {
    super.setUp()
    validator = UserValidator()
}

func testValidEmailFormat() {
    // Arrange
    let email = "user@example.com"

    // Act
    let result = validator.isValidEmail(email)

    // Assert
    XCTAssertTrue(result)
}

func testInvalidEmailMissingAtSign() {
    let email = "userexample.com"
    let result = validator.isValidEmail(email)
    XCTAssertFalse(result)
}

func testPasswordMinimumLength() {
    let password = "short"
    let result = validator.isValidPassword(password)
    XCTAssertFalse(result)
}

func testPasswordWithAllRequirements() {
    let password = "SecurePass123!"
    let result = validator.isValidPassword(password)
    XCTAssertTrue(result)
```

```
}
```

}

2. Mocking and Stubbing Patterns

Android Example (Mockito):

class UserRepositoryTest {
@Mock
private lateinit var userDao: UserDao

```
@Mock
private lateinit var userApi: UserApi

private lateinit var userRepository: UserRepository

@Before
fun setup() {
    MockitoAnnotations.openMocks(this)
    userRepository = UserRepository(userDao, userApi)
}

@Test
fun testFetchUserFromNetwork() {
    // Arrange
    val userId = "123"
    val mockUser = User(id = "123", name = "John Doe")

    `when`(userApi.getUser(userId)).thenReturn(mockUser)

    // Act
    val result = userRepository.fetchUser(userId)

    // Assert
    assertEquals(mockUser, result)
    verify(userApi).getUser(userId)
    verify(userDao).insertUser(mockUser)
}

@Test
fun testFetchUserRetryOnNetworkError() {
    // Arrange
    val userId = "123"
    `when`(userApi.getUser(userId))
        .thenThrow(IOException("Network error"))
        .thenReturn(User(id = "123", name = "John"))

    // Act & Assert
    assertThrows(IOException::class.java) {
        userRepository.fetchUser(userId)
```

```
    }

    verify(userApi, times(1)).getUser(userId)
}
```

}

iOS Example (XCTest + Mock Objects):

class UserRepositoryTests: XCTestCase {
var repository: UserRepository!
var mockApi: MockUserAPI!
var mockDatabase: MockDatabase!

```
override func setUp() {
    super.setUp()
    mockApi = MockUserAPI()
    mockDatabase = MockDatabase()
    repository = UserRepository(api: mockApi, database: mockDatabase)
}

func testFetchUserFromNetwork() {
    // Arrange
    let userId = "123"
    let expectedUser = User(id: "123", name: "John Doe")
    mockApi.mockUser = expectedUser

    // Act
    let result = repository.fetchUser(userId)

    // Assert
    XCTAssertEqual(result.id, expectedUser.id)
    XCTAssertTrue(mockApi.getWasCalled)
    XCTAssertTrue(mockDatabase.insertWasCalled)
}

func testFetchUserRetryOnNetworkError() {
    // Arrange
    let userId = "123"
    mockApi.shouldThrowError = true

    // Act & Assert
    XCTAssertThrowsError(try repository.fetchUser(userId))
}
```

}

class MockUserAPI: UserAPI {
var mockUser: User?
var shouldThrowError = false
var getWasCalled = false

```
func getUser(_ id: String) throws -> User {
    getWasCalled = true
    if shouldThrowError {
        throw NSError(domain: "Network", code: -1)
    }
    return mockUser!
}
```

}

3. Test Organization Best Practices

AAA Pattern (Arrange-Act-Assert):
@Test
fun testUserLogin() {
// Arrange - Set up test data
val email = "user@example.com"
val password = "password123"

```
// Act - Execute the code under test
val result = userRepository.login(email, password)

// Assert - Verify results
assertTrue(result.isSuccess)
assertEquals("user@example.com", result.user.email)
```

}

Descriptive Test Names:

- ✓ Good: `testLoginFailsWithInvalidPassword`
- ✓ Good: `testCalculateDiscountFor50PercentOff`
- ✗ Bad: `test1`, `testLogic`, `testFunc`

Test Isolation:

- Each test should be independent
- Use `@Before`/`setUp()` to reset state
- No shared mutable state between tests
- Tests can run in any order

4. Common Unit Test Scenarios

Testing Coroutines (Kotlin):

class UserViewModelTest {
@get:Rule
val instantExecutorRule = InstantTaskExecutorRule()

```
private val testDispatcher = StandardTestDispatcher()

@Before
fun setup() {
    Dispatchers.setMain(testDispatcher)
}

@Test
fun testLoadUserSucceeds() = runTest {
    // Arrange
    val mockUser = User(id = "1", name = "John")
    `when`(repository.getUser("1")).thenReturn(mockUser)

    val viewModel = UserViewModel(repository)

    // Act
    viewModel.loadUser("1")
    advanceUntilIdle()

    // Assert
    assertEquals(mockUser, viewModel.user.value)
}
```

}

Testing LiveData (Android):

@Test
fun testUserLiveDataUpdates() {
// Arrange
val user = User(id = "1", name = "John")
val viewModel = UserViewModel(repository)

```
val testObserver = TestObserver<User>()
viewModel.user.observeForever(testObserver)

// Act
viewModel.setUser(user)

// Assert
assertEquals(user, testObserver.observedValues.last())
```

}

Deliverable:

- [ ] Three complete unit test classes (Android or iOS)
- [ ] Mocking/stubbing examples with 2 different scenarios
- [ ] AAA pattern demonstration in 3 tests

- [ ] Test naming conventions document
- [ ] Explanation of test isolation and independence

Part B: Integration and UI Testing

Task 3: Integration Testing Framework (Self-Study: 2 hours)

Objective: Master integration testing, test databases, and API mocking.

Instructions:

1. Integration Testing Strategy

Integration tests verify interaction between layers:

UI Layer
↓
ViewModel/Controller
↓
Repository ← Integration Test Here
↓
Local Database + Remote API

2. Android Integration Testing

Room Database Integration Test:

@RunWith(AndroidJUnit4::class)
class UserDatabaseTest {
@get:Rule
val instantExecutorRule = InstantTaskExecutorRule()

```
private lateinit var userDao: UserDao
private lateinit var database: AppDatabase

@Before
fun setup() {
    // Create in-memory test database
    database = Room.inMemoryDatabaseBuilder(
        ApplicationProvider.getApplicationContext(),
        AppDatabase::class.java
    ).allowMainThreadQueries().build()

    userDao = database.userDao()
}

@After
```

```
fun teardown() {
    database.close()
}

@Test
fun testInsertAndRetrieveUser() = runTest {
    // Arrange
    val user = User(id = "1", name = "John", email =
"john@example.com")

    // Act
    userDao.insertUser(user)
    val retrieved = userDao.getUserById("1")

    // Assert
    assertEquals(user.id, retrieved.id)
    assertEquals("John", retrieved.name)
}

@Test
fun testUpdateUserEmail() = runTest {
    // Arrange
    val user = User(id = "1", name = "John", email =
"john@example.com")
    userDao.insertUser(user)

    // Act
    val updated = user.copy(email = "newemail@example.com")
    userDao.updateUser(updated)
    val retrieved = userDao.getUserById("1")

    // Assert
    assertEquals("newemail@example.com", retrieved.email)
}

@Test
fun testDeleteUser() = runTest {
    // Arrange
    val user = User(id = "1", name = "John", email =
"john@example.com")
    userDao.insertUser(user)

    // Act
    userDao.deleteUser(user)
    val retrieved = userDao.getUserById("1")

    // Assert
    assertNull(retrieved)
}
```

}

Repository Integration Test:

@RunWith(AndroidJUnit4::class)
class UserRepositoryIntegrationTest {
private lateinit var userDao: UserDao
private lateinit var database: AppDatabase
private lateinit var mockApi: UserApi
private lateinit var repository: UserRepository

```
@Before
fun setup() {
    database = Room.inMemoryDatabaseBuilder(
        ApplicationProvider.getApplicationContext(),
        AppDatabase::class.java
    ).allowMainThreadQueries().build()

    userDao = database.userDao()
    mockApi = mock(UserApi::class.java)
    repository = UserRepository(userDao, mockApi)
}

@Test
fun testFetchUserAndSaveToDatabase() = runTest {
    // Arrange
    val remoteUser = User(
        id = "1",
        name = "John",
        email = "john@example.com"
    )
    `when`(mockApi.getUser("1")).thenReturn(remoteUser)

    // Act
    repository.fetchUser("1")
    val savedUser = userDao.getUserById("1")

    // Assert
    assertNotNull(savedUser)
    assertEquals(remoteUser.email, savedUser!!.email)
}

@Test
fun testReturnCachedUserWhenOffline() = runTest {
    // Arrange
    val cachedUser = User(id = "1", name = "John", email:
"john@example.com")
    userDao.insertUser(cachedUser)

    `when`(mockApi.getUser("1"))
        .thenThrow(IOException("Network error"))
```

```
    // Act
    val result = repository.fetchUserWithFallback("1")

    // Assert
    assertEquals(cachedUser, result)
}

@After
fun teardown() {
    database.close()
}
```

}

3. iOS Integration Testing

CoreData Integration Test:

class UserRepositoryIntegrationTests: XCTestCase {
var repository: UserRepository!
var coreDataStack: CoreDataStack!
var mockAPI: MockUserAPI!

```
override func setUp() {
    super.setUp()
    coreDataStack = CoreDataStack.testing()
    mockAPI = MockUserAPI()
    repository = UserRepository(
        api: mockAPI,
        coreDataStack: coreDataStack
    )
}

override func tearDown() {
    super.tearDown()
    coreDataStack.deleteAllData()
}

func testFetchAndSaveUserToCoreData() {
    // Arrange
    let expectedUser = User(id: "1", name: "John", email:
"john@example.com")
    mockAPI.mockUser = expectedUser

    // Act
    repository.fetchUser("1") { result in
        // Assert
        XCTAssertEqual(result.id, expectedUser.id)

        // Verify saved in CoreData
```

```
        let saved = self.coreDataStack.fetchUser(id: "1")
        XCTAssertNotNil(saved)
        XCTAssertEqual(saved?.email, expectedUser.email)
    }
}

func testReturnCachedUserWhenAPIFails() {
    // Arrange
    let cachedUser = User(id: "1", name: "John", email:
"john@example.com")
    coreDataStack.save(user: cachedUser)
    mockAPI.shouldThrowError = true

    // Act
    repository.fetchUserWithFallback("1") { result in
        // Assert
        XCTAssertEqual(result, cachedUser)
    }
}
```

}

class CoreDataStack {
static func testing() -> CoreDataStack {
let model = NSManagedObjectModel()
let container = NSPersistentContainer(
name: "TestModel",
managedObjectModel: model
)
// Use in-memory persistent store
let description = NSPersistentStoreDescription()
description.url = URL(fileURLWithPath: "/dev/null")
container.persistentStoreDescriptions = [description]
return CoreDataStack(container: container)
}
}

Deliverable:

- [ ] Android Room database integration tests (3+ scenarios)
- [ ] Android Repository integration test with mock API
- [ ] iOS CoreData integration test with mock API
- [ ] Test database setup and teardown explanation
- [ ] Comparison of unit vs. integration test benefits

Task 4: UI Testing with Espresso, XCTest, and Detox (Self-Study: 2.5 hours)

Objective: Master UI testing frameworks and end-to-end testing strategies.

Instructions:

1. Android UI Testing with Espresso

Setup:

dependencies {
androidTestImplementation 'androidx.test.espresso:espresso-core:3.5.1'
androidTestImplementation 'androidx.test 🏃 1.5.2'
androidTestImplementation 'androidx.test:rules:1.5.0'
androidTestImplementation 'androidx.test.uiautomator:uiautomator:2.3.0'
}

Login Flow UI Test:

@RunWith(AndroidJUnit4::class)
class LoginActivityTest {
@get:Rule
val activityRule = ActivityScenarioRule(LoginActivity::class.java)

```
@Test
fun testLoginWithValidCredentials() {
    // Arrange & Act
    onView(withId(R.id.email_input))
        .perform(typeText("user@example.com"))

    onView(withId(R.id.password_input))
        .perform(typeText("password123"))

    onView(withId(R.id.login_button))
        .perform(click())

    // Assert - Verify navigation to home
    onView(withId(R.id.home_fragment))
        .check(matches(isDisplayed()))
}

@Test
fun testLoginFailsWithInvalidEmail() {
    // Arrange & Act
    onView(withId(R.id.email_input))
        .perform(typeText("invalid-email"))

    onView(withId(R.id.password_input))
        .perform(typeText("password123"))

    onView(withId(R.id.login_button))
        .perform(click())
```

```
    // Assert
    onView(withId(R.id.error_message))
        .check(matches(isDisplayed()))
    onView(withText("Invalid email format"))
        .check(matches(isDisplayed()))
}

@Test
fun testLoginShowsLoadingIndicator() {
    // Arrange
    val mockRepository = mock(UserRepository::class.java)
    `when`(mockRepository.login("user@example.com", "password123"))
        .thenReturn(MutableLiveData<Result<User>>())

    // Act
    onView(withId(R.id.email_input))
        .perform(typeText("user@example.com"))
    onView(withId(R.id.password_input))
        .perform(typeText("password123"))
    onView(withId(R.id.login_button))
        .perform(click())

    // Assert
    onView(withId(R.id.loading_spinner))
        .check(matches(isDisplayed()))
}

@Test
fun testNavigateBackFromLogin() {
    // Act
    Espresso.pressBack()

    // Assert - Should close activity
    onView(withId(R.id.login_activity))
        .check(doesNotExist())
}
```

```
}
```

RecyclerView List Testing:

```
@RunWith(AndroidJUnit4::class)
class UserListActivityTest {
@get:Rule
val activityRule = ActivityScenarioRule(UserListActivity::class.java)

@Test
fun testDisplayUserList() {
    // Assert - Verify list items displayed
    onView(withId(R.id.user_list))
        .check(matches(isDisplayed()))
```

```
    onView(withId(R.id.user_item_name))
        .check(matches(withText(containsString("John"))))
}

@Test
fun testClickUserItem() {
    // Act
    onView(withId(R.id.user_list))
        .perform(
            actionOnItemAtPosition<UserAdapter.ViewHolder>(
                0,
                click()
            )
        )

    // Assert
    onView(withId(R.id.user_detail_activity))
        .check(matches(isDisplayed()))
}

@Test
fun testScrollListAndVerifyItems() {
    // Act
    onView(withId(R.id.user_list))
        .perform(scrollToPosition<UserAdapter.ViewHolder>(10))

    // Assert
    onView(atPosition(10, withId(R.id.user_item_name)))
        .check(matches(isDisplayed()))
}
```

}

2. iOS UI Testing with XCUITest

Setup:

import XCTest

class LoginViewControllerUITests: XCTestCase {
var app: XCUIApplication!

```
override func setUp() {
    super.setUp()
    continueAfterFailure = false
    app = XCUIApplication()
    app.launch()
}

func testLoginWithValidCredentials() {
```

```
    // Arrange
    let emailInput = app.textFields["emailTextField"]
    let passwordInput = app.secureTextFields["passwordTextField"]
    let loginButton = app.buttons["loginButton"]

    // Act
    emailInput.tap()
    emailInput.typeText("user@example.com")

    passwordInput.tap()
    passwordInput.typeText("password123")

    loginButton.tap()

    // Assert
    let homeView = app.otherElements["homeViewController"]
    XCTAssertTrue(homeView.waitForExistence(timeout: 5))
}

func testLoginFailsWithInvalidEmail() {
    // Arrange
    let emailInput = app.textFields["emailTextField"]
    let passwordInput = app.secureTextFields["passwordTextField"]
    let loginButton = app.buttons["loginButton"]

    // Act
    emailInput.tap()
    emailInput.typeText("invalid-email")
    passwordInput.tap()
    passwordInput.typeText("password123")
    loginButton.tap()

    // Assert
    let errorMessage = app.staticTexts["Invalid email format"]
    XCTAssertTrue(errorMessage.waitForExistence(timeout: 2))
}

func testLoginShowsLoadingState() {
    // Act
    let emailInput = app.textFields["emailTextField"]
    let passwordInput = app.secureTextFields["passwordTextField"]
    let loginButton = app.buttons["loginButton"]

    emailInput.typeText("user@example.com")
    passwordInput.typeText("password123")
    loginButton.tap()

    // Assert
    let spinner = app.activityIndicators["loadingSpinner"]
    XCTAssertTrue(spinner.waitForExistence(timeout: 1))
```

```
}
```

}

Table View Testing:

class UserListViewControllerUITests: XCTestCase {
var app: XCUIApplication!

```
override func setUp() {
    super.setUp()
    app = XCUIApplication()
    app.launch()
}

func testDisplayUserList() {
    // Assert
    let userTable = app.tables["userTableView"]
    XCTAssertTrue(userTable.exists)

    let firstCell = userTable.cells.element(boundBy: 0)
    XCTAssertTrue(firstCell.waitForExistence(timeout: 2))
}

func testTapUserCell() {
    // Arrange
    let userTable = app.tables["userTableView"]
    let firstCell = userTable.cells.element(boundBy: 0)

    // Act
    firstCell.tap()

    // Assert
    let detailViewController = app.navigationBars["User Details"]
    XCTAssertTrue(detailViewController.waitForExistence(timeout: 2))
}

func testScrollTableAndVerifyContent() {
    // Arrange
    let userTable = app.tables["userTableView"]

    // Act
    userTable.swipeUp(velocity: .fast)

    // Assert
    let cell10 = userTable.cells.element(boundBy: 10)
    XCTAssertTrue(cell10.isVisible)
}
```

}

3. Cross-Platform E2E Testing with Detox

Setup:

npm install detox-cli --global
npm install detox detox-test-utils detox-cli --save-dev
detox build-framework-cache
detox build-app --configuration ios.sim.release

Detox Test Examples:

describe('Login Flow', () => {
beforeAll(async () => {
await device.launchApp();
});

beforeEach(async () => {
await device.reloadReactNative();
});

it('should login with valid credentials', async () => {
// Arrange & Act
await element(by.id('emailInput')).typeText('user@example.com');
await element(by.id('passwordInput')).typeText('password123');
await element(by.id('loginButton')).tap();

```
// Assert
await waitFor(element(by.id('homeScreen')))
  .toBeVisible()
  .withTimeout(5000);
```

});

it('should show error for invalid email', async () => {
// Act
await element(by.id('emailInput')).typeText('invalid-email');
await element(by.id('passwordInput')).typeText('password123');
await element(by.id('loginButton')).tap();

```
// Assert
await expect(element(by.text('Invalid email format')))
  .toBeVisible();
```

});

it('should show loading during login', async () => {
// Act
await element(by.id('emailInput')).typeText('user@example.com');
await element(by.id('passwordInput')).typeText('password123');
await element(by.id('loginButton')).tap();

```
// Assert
await expect(element(by.id('loadingSpinner')))
```

```
  .toBeVisible();
```

});
});

describe('User List Navigation', () => {
beforeAll(async () => {
await device.launchApp();
});

it('should display user list', async () => {
// Assert
await waitFor(element(by.id('userList')))
.toBeVisible()
.withTimeout(3000);

```
await expect(element(by.id('userItem0'))).toBeVisible();
```

});

it('should navigate to user detail on tap', async () => {
// Act
await element(by.id('userItemo')).tap();

```
// Assert
await waitFor(element(by.id('userDetailScreen')))
  .toBeVisible()
  .withTimeout(2000);
```

});

it('should scroll list and display more users', async () => {
// Act
await waitFor(element(by.id('userList')))
.toBeVisible()
.withTimeout(2000);

```
await element(by.id('userList')).swipe('up', 'slow');

// Assert
await expect(element(by.id('userItem10'))).toBeVisible();
```

});
});

Detox Configuration (package.json):

{
"detox": {
"configurations": {
"ios.sim.debug": {
"device": {

```
"type": "iOS.Simulator",
"device": {
"type": "iPhone 14"
}
},
"app": "ios.debug"
},
"ios.sim.release": {
"device": {
"type": "iOS.Simulator",
"device": {
"type": "iPhone 14"
}
},
"app": "ios.release"
}
},
"apps": {
"ios.debug": {
"type": "ios.app",
"binaryPath": "ios/build/Build/Products/Release-iphonesimulator/MyApp.app",
"build": "xcodebuild -workspace ios/MyApp.xcworkspace -scheme MyApp -configuration
Release -derivedDataPath ios/build"
},
"ios.release": {
"type": "ios.app",
"binaryPath": "ios/build/Build/Products/Release-iphonesimulator/MyApp.app",
"build": "xcodebuild -workspace ios/MyApp.xcworkspace -scheme MyApp -configuration
Release -derivedDataPath ios/build"
}
},
"testRunner": "jest"
}
}
```


Deliverable:

- [ ] Three Espresso tests demonstrating different interactions
- [ ] Three XCUITest tests demonstrating different interactions
- [ ] Three Detox cross-platform tests
- [ ] Explanation of test selectors and wait strategies
- [ ] Best practices for UI test reliability and maintainability

---

Part C: App Distribution and Store Submission

Task 5: Android Distribution via Google Play Console (Self-Study: 2.5 hours)

Objective: Master app signing, Google Play Console submission, and distribution strategy.

Instructions:

1. App Signing and Keystore Management

Generate Signing Key (First Release):

Generate keystore file

keytool -genkey -v -keystore release.keystore
-keyalg RSA -keysize 2048 -validity 10000
-alias release_key

Output should be saved securely

Never commit keystore to version control!

Android Studio Guided Key Generation:

1. Build → Generate Signed App Bundle/APK
2. Create new keystore
3. Fill in key details:
    - Key Store Path: `release.keystore`
    - Password: Strong password (save securely)
    - Key Alias: `release_key`
    - Key Password: Same as keystore
    - Validity: 25+ years

Signing Configuration (build.gradle):

android {
signingConfigs {
release {

storeFile file("release.keystore")
storePassword System.getenv("KEYSTORE_PASSWORD")
keyAlias System.getenv("KEY_ALIAS")
keyPassword System.getenv("KEY_PASSWORD")
}
}

```
buildTypes {
    release {
        signingConfig signingConfigs.release
        minifyEnabled true
        shrinkResources true
        proguardFiles getDefaultProguardFile('proguard-android-
optimize.txt'), 'proguard-rules.pro'
    }
}
```

}

Security Best Practices:

- Never commit `release.keystore` to version control
- Store passwords in environment variables or secure vault
- Use separate signing keys for dev/test/production
- Rotate keys periodically (if possible)
- Document key generation process and backup location

2. App Bundle vs APK

Aspect	APK	App Bundle
Size	Larger (includes all architectures)	Smaller (optimized per device)
Play Store	Older method	Recommended
Installation	Direct install	Auto-optimized by Play Store
Architectures	All (arm, x86, etc.)	Per-architecture split
Minimum SDK	Single version	Single version
Languages	All languages	User's language

Generate Release Bundle:

// In build.gradle
android {
bundle {
language {
// Enable language split
enableSplit = true

```
}
density {
// Enable density split
enableSplit = true
}
abi {
// Enable ABI split
enableSplit = true
}
}
}
```

Build App Bundle

```
./gradlew bundleRelease
```

Output: app/release/app-release.aab

3. Google Play Console Submission

Pre-Submission Checklist:

- [ ] App signed with release key
- [ ] Version code incremented (`versionCode` in build.gradle)
- [ ] Version name set appropriately (e.g., "1.0.0")
- [ ] App name and description finalized
- [ ] Privacy policy URL provided
- [ ] Screenshots captured (2-5 per device type)
- [ ] App icon provided (512x512 px)
- [ ] Feature graphic (1024x500 px)
- [ ] All content ratings filled
- [ ] Content policy compliance reviewed

Upload to Play Console:

```
// AndroidManifest.xml version configuration
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
android:versionCode="1" android:versionName="1.0.0" package="com.example.myapp">
```

Staged Rollout Strategy:

Phase	Percentage	Duration	Purpose

Internal Testing	100%	24-48 hours	Internal QA, critical bugs
Closed Testing	10-25%	3-5 days	Early user feedback
Open Testing	50%	5-7 days	Larger user base, stability
Production	100%	Ongoing	Full release

Rollout Flow:
Internal Testing (5%)
↓ (No issues → proceed)
Closed Testing (10%)
↓ (Monitor crash rate < 0.5%)
Open Testing (50%)
↓ (Monitor user feedback)
Production (100%)

Store Listing Best Practices:

Title: [App Name] - [Main Feature]
Example: "BudgetApp - Smart Expense Tracker"

Short Description (80 chars):
"Track spending, save money, reach financial goals"

Full Description (4000 chars):

- What it does
- Key features (3-5 bullets)
- How to use (step by step)
- Privacy/security statements
- Technical requirements

Screenshots Order:

1. Main feature highlight
2. Key functionality
3. User benefit
4. Secondary feature
5. Call-to-action

Keywords (5):

- expense tracker
- budget app
- money management
- financial planning
- spending tracker

4. Compliance and Review

Google Play Policies (Critical):

- No malware, spyware, or deceptive behavior
- Privacy policy required for apps collecting personal data
- Age rating mandatory (IARC rating system)
- Ads must be clearly labeled
- In-app purchases clearly disclosed
- Contact information and support required

Submission Response:

- Google reviews within 24-48 hours
- If rejected: detailed explanation provided
- Fix and resubmit
- Appeals available for policy violations

5. Post-Release Management

Monitoring Metrics:

Critical Metrics:

- Crash rate: < 0.1%
- ANR (App Not Responding): < 0.5%
- User reviews: Target 4.0+ stars
- Installs/day
- Uninstall rate
- Session length
- User retention

Weekly Review Process:

1. Monitor crash reports
2. Review user feedback
3. Analyze performance metrics
4. Plan hot fixes if needed
5. Document issues for next version

Deliverable:

- [ ] Step-by-step app signing guide with keystore generation
- [ ] build.gradle signing configuration (with security notes)

- [ ] Google Play Console submission checklist (20+ items)
- [ ] Store listing example with title, description, keywords
- [ ] Rollout strategy document with percentages and timelines
- [ ] Post-release monitoring metrics and response procedures

---

Task 6: iOS Distribution via App Store Connect (Self-Study: 2.5 hours)

Objective: Master iOS app signing, TestFlight, and App Store submission.

Instructions:

1. iOS Code Signing and Certificates

Certificate Types:

Type	Purpose	Validity	Count
Development	Local testing	1 year	1+
Distribution	App Store	3 years	1+
Push Notification	APNS authentication	1 year	Per app
Website Push	Web notifications	3 years	Optional

Generate Signing Certificate:

1. **Request Certificate from Keychain Access:**
    - Keychain Access → Certificate Assistant → Request Certificate
    - Email: your Apple ID email
    - Common Name: Your name
    - Saved to disk
    - Create: `CertificateSigningRequest.certSigningRequest`
2. **Upload to Apple Developer Account:**
    - Log in to developer.apple.com
    - Certificates, Identifiers & Profiles
    - Create new certificate (select type)
    - Upload CSR
    - Download `.cer` file
    - Double-click to install

Create Provisioning Profile:

1. Go to Certificates, Identifiers & Profiles
2. Select Provisioning Profiles → New
3. Choose type: App Store (production)
4. Select App ID (e.g., `com.example.myapp`)
5. Select signing certificate
6. Name profile: `MyApp_Distribution`
7. Download: `MyApp_Distribution.mobileprovision`
8. Drag into Xcode

Xcode Build Configuration:

```
// In Xcode: Project Settings
Build Settings:
├─ Code Signing Style: Automatic or Manual
├─ Signing Certificate: iOS Distribution
├─ Provisioning Profile: MyApp_Distribution
├─ Development Team: Your Team ID
└─ Bundle Identifier: com.example.myapp
```

Check Signing Configuration:

```
# Verify code signing
codesign -dv /path/to/MyApp.app

# List provisioning profiles
ls ~/Library/MobileDevice/Provisioning\ Profiles/

# Verify certificate
security find-certificate -c "iPhone Distribution" ~/Library/Keychains/login.keychain-db
```

2. TestFlight Beta Distribution

Setup TestFlight:

1. **Create TestFlight Group:**
    - App Store Connect → App → TestFlight
    - Create new group: "Internal Testing"
    - Add testers (Apple ID emails)
2. **Upload Build to TestFlight:**

Build for archive

cd ios
xcodebuild -workspace MyApp.xcworkspace
-scheme MyApp
-configuration Release
-derivedDataPath build
-archivePath build/MyApp.xcarchive
archive

Export as IPA

xcodebuild -exportArchive
-archivePath build/MyApp.xcarchive
-exportPath build/Export
-exportOptionsPlist ExportOptions.plist

ExportOptions.plist (TestFlight):

method app-store signingStyle automatic teamID YOUR_TEAM_ID provisioningProfiles com.example.myapp MyApp_Distribution

3. **Upload to App Store Connect:**

Using altool (deprecated but still works)

xcrun altool --upload-app
--file build/Export/MyApp.ipa
--type ios
--username "apple@example.com"
--password "@keychain:Developer-altool-password"

Using Transporter (recommended)

Download from App Store → Transporter

Drag IPA file → Upload

4. **TestFlight Build Review:**
    - Apple reviews build (usually <24 hours)
    - Once approved, send invitations to testers
    - Testers access via TestFlight app

TestFlight Configuration:

Internal Testers:
├─ Immediate access (no Apple review)
├─ Good for: development, QA teams
└─ Typical: 25-50 testers

External Testers:
├─ Requires Apple review (~48 hours)
├─ Good for: beta feedback, user testing
└─ Typical: 10,000+ testers allowed

3. App Store Submission

Pre-Submission Checklist:

- [ ] Version number incremented (e.g., 1.0.0 → 1.0.1)
- [ ] Build tested on device
- [ ] Screenshots captured (all required sizes)
- [ ] App preview video recorded (optional but recommended)
- [ ] App description written (4000 chars max)
- [ ] Keywords optimized (30 chars × 3 keywords)
- [ ] Support URL provided
- [ ] Privacy policy URL provided
- [ ] App icon submitted (1024×1024 px)
- [ ] Age rating completed (via app questionnaire)
- [ ] App category selected
- [ ] Pricing and availability set
- [ ] Content rights confirmed
- [ ] Export compliance checked

App Store Listing Best Practices:

Name: [App Name] - [Main Feature]
Example: "Budget Master - Smart Finance Tracker"

Subtitle (30 chars):
"Track, analyze, save"

Keyword Examples:

- budget
- expense tracking
- financial planning
- money management
- savings

Description Structure:

1. Hook (what it does)
2. Key features (3-5 bullets)
3. How to use (3-4 steps)
4. Privacy/safety statement
5. Requirements/support

Screenshot Sequence:

1. Main feature (onboarding)
2. Core functionality
3. Data insights
4. Social features
5. Settings/customization

4. App Store Review Guidelines

Common Rejection Reasons:

Issue	Example	Fix
Metadata	Incomplete description	Add full app description
Performance	Crashes on older devices	Test on iOS 13+
Privacy	Missing privacy policy	Add privacy policy URL
Functionality	Broken login flow	Test all user flows
Ads	Intrusive ads	Follow Ad framework guidelines
In-app purchase	Not clearly labeled	Label all purchases

Review Status Tracking:

Submitted
↓ (24-48 hours)
In Review
↓
Approved / Rejected

↓
Ready for Sale (if approved)

If rejected:

1. Read detailed feedback
2. Make required changes
3. Increment build number
4. Upload new build
5. Resubmit

5. Continuous Delivery with Xcode Cloud

Setup Xcode Cloud:

// Xcode Cloud workflow configuration
name: Build and Test
on:
push:
branches: [main, develop]
pull_request:
branches: [main]

env:
SCHEME: MyApp

jobs:
build:
runs-on: macos-latest
steps:
- uses: actions/checkout@v2

```
  - name: Build
    run: |
      xcodebuild \
        -workspace MyApp.xcworkspace \
        -scheme $SCHEME \
        -configuration Release \
        -derivedDataPath build

  - name: Test
    run: |
      xcodebuild test \
        -workspace MyApp.xcworkspace \
        -scheme $SCHEME \
        -destination 'platform=iOS Simulator,name=iPhone 14'

  - name: Upload to TestFlight
    if: github.event_name == 'push' && github.ref == 'refs/heads/main'
    run: |
      # Upload to App Store Connect
```

```
      xcrun altool --upload-app \
        --file build/Export/MyApp.ipa \
        --type ios \
        --username ${{ secrets.APPLE_ID }} \
        --password ${{ secrets.APPLE_PASSWORD }}
```

Deliverable:

- [ ] iOS code signing guide (certificate + provisioning profile)
- [ ] TestFlight setup and upload instructions
- [ ] App Store Connect submission checklist
- [ ] App Store listing example (name, description, keywords)
- [ ] Build versioning strategy document
- [ ] App review guidelines and common rejection reasons
- [ ] Continuous delivery workflow configuration

Part D: Testing Automation and Continuous Integration

Task 7: CI/CD Pipeline and Test Automation (Self-Study: 1.5 hours)

Objective: Master automated testing in CI/CD pipelines.

Instructions:

1. GitHub Actions CI/CD Pipeline

Android CI Pipeline (.github/workflows/android-ci.yml):

name: Android CI

on:
push:
branches: [main, develop]
pull_request:
branches: [main]

jobs:
build:
runs-on: ubuntu-latest

```
steps:
- uses: actions/checkout@v3
```

```
- name: Set up JDK
  uses: actions/setup-java@v3
  with:
    java-version: 17
    distribution: 'temurin'
    cache: gradle

- name: Grant execute permission for gradlew
  run: chmod +x gradlew

- name: Run unit tests
  run: ./gradlew testDebugUnitTest

- name: Run integration tests
  run: ./gradlew connectedAndroidTest

- name: Build APK
  run: ./gradlew assembleRelease

- name: Build App Bundle
  run: ./gradlew bundleRelease

- name: Upload APK to artifacts
  uses: actions/upload-artifact@v3
  with:
    name: app-release.apk
    path: app/build/outputs/apk/release/app-release.apk

- name: Upload coverage reports
  uses: codecov/codecov-action@v3
  with:
    files: ./app/build/reports/coverage/debug/report.xml
```

lint:
runs-on: ubuntu-latest

```
steps:
- uses: actions/checkout@v3

- name: Set up JDK
  uses: actions/setup-java@v3
  with:
    java-version: 17
    distribution: 'temurin'

- name: Grant execute permission
  run: chmod +x gradlew

- name: Run lint checks
```

```
  run: ./gradlew lint

- name: Upload lint reports
  uses: github/codeql-action/upload-sarif@v2
  if: always()
  with:
    sarif_file: app/build/reports/lint-results.sarif
```

deploy-testflight:
if: github.ref == 'refs/heads/main' && github.event_name == 'push'
runs-on: macos-latest
needs: [build, lint]

```
steps:
- uses: actions/checkout@v3

- name: Build for TestFlight
  run: |
    xcodebuild -workspace ios/MyApp.xcworkspace \
      -scheme MyApp \
      -configuration Release \
      -derivedDataPath build \
      -archivePath build/MyApp.xcarchive \
      archive

- name: Export IPA
  run: |
    xcodebuild -exportArchive \
      -archivePath build/MyApp.xcarchive \
      -exportPath build/Export \
      -exportOptionsPlist ios/ExportOptions.plist

- name: Upload to TestFlight
  env:
    APPLE_ID: ${{ secrets.APPLE_ID }}
    APPLE_PASSWORD: ${{ secrets.APPLE_PASSWORD }}
  run: |
    xcrun altool --upload-app \
      --file build/Export/MyApp.ipa \
      --type ios \
      --username $APPLE_ID \
      --password $APPLE_PASSWORD
```

2. Test Coverage Reporting

Android Jacoco Coverage:

// In build.gradle
plugins {

id 'jacoco'
}

jacoco {
toolVersion = "0.8.8"
}

task jacocoTestReport(type: JacocoReport) {
dependsOn testDebugUnitTest

```
reports {
    xml.required = true
    csv.required = false
    html.required = true
}

afterEvaluate {
    classDirectories.setFrom(files(classDirectories.files.collect {
        fileTree(dir: it, exclude: [
            '**/R.class',
            '**/R$*.class',
            '**/BuildConfig.*',
            '**/Manifest*.*'
        ])
    }))
}
```

}

Coverage Reporting Configuration:

Generate coverage report

./gradlew jacocoTestReport

View HTML report

open app/build/reports/jacoco/jacocoTestReport/html/index.html

Upload to Codecov

curl -s https://codecov.io/bash | bash -s --
-f app/build/reports/coverage/debug/report.xml

3. Test Result Analysis

Test Metrics Dashboard:

Critical Metrics to Monitor:

Test Execution Time:
├─ Unit tests: < 5 minutes
├─ Integration tests: < 10 minutes
├─ UI tests: < 15 minutes
└─ Total pipeline: < 30 minutes

Test Coverage:
├─ Overall: > 70%
├─ Critical paths: > 90%
├─ UI layer: > 20%
└─ Utilities: > 80%

Test Health:
├─ Pass rate: > 99%
├─ Flaky tests: < 2%
├─ Test reliability: 100% repeatable
└─ No false positives

Deliverable:

- [ ] Complete GitHub Actions CI pipeline for Android
- [ ] Complete GitHub Actions CI pipeline for iOS
- [ ] Jacoco/coverage configuration and reporting
- [ ] Test metrics dashboard explanation
- [ ] Performance benchmarks for test execution
- [ ] Documentation for troubleshooting failed tests

Part E: Evaluation and Strategic Best Practices

Task 8: Testing Strategy Review and Optimization (Self-Study: 1 hour)

Objective: Develop comprehensive testing strategy and continuous improvement.

Instructions:

1. Testing Strategy Framework

Define Test Coverage Goals:

Critical Features (90%+ coverage):
├─ Authentication & Authorization
├─ Payment & Financial Transactions

├─ Data Validation & Security
├─ Error Handling
└─ User Privacy

Core Features (70%+ coverage):
├─ Main workflows
├─ Data persistence
├─ API integration
├─ Networking
└─ Caching strategies

Nice-to-Have (40%+ coverage):
├─ Analytics
├─ Preferences
├─ UI animations
├─ Accessibility
└─ Localization

2. Test Maintenance and Refactoring

Code Review for Tests:

Test Review Checklist:
- ☑ Test name clearly describes intent
- ☑ Test follows AAA pattern
- ☑ Test is isolated (no dependencies)
- ☑ Test doesn't repeat production code
- ☑ Assertions are specific (not generic)
- ☑ Mock/stub usage appropriate
- ☑ Test is fast (unit) or acceptable (integration)
- ☑ No hardcoded wait times
- ☑ No test-only code in production
- ☑ Proper setup/teardown

3. Team Best Practices

Testing Culture:
• Tests are first-class code
• Test coverage is tracked metric
• Failing tests = blocked code merge
• Code review includes test review
• Performance tests run regularly
• Test results visible to entire team
• Failed tests = immediate priority
• Share test knowledge across team

Deliverable:

- [ ] Testing strategy framework for specific app domain
- [ ] Test coverage goals document
- [ ] Test code review checklist

- [ ] Testing metrics dashboard design
- [ ] Team testing best practices guide
- [ ] Continuous improvement plan

Evaluation Criteria

Criteria	Excellent (9–10)	Good (7–8)	Acceptable (5–6)	Needs Improvement
Test Understanding	All test types mastered	3-4 test types clear	2 test types understood	Single test type
Framework Expertise	Multiple frameworks mastered	2+ frameworks solid	1 framework understood	Minimal framework knowledge
Test Implementation	Comprehensive test suite (>70%)	Good coverage (50-70%)	Basic tests (30-50%)	Minimal testing
Code Quality	Clean, idiomatic implementations	Generally good, some inconsistency	Readable, lacks depth	Difficult to follow
Distribution Mastery	Both platforms mastered	1 platform thoroughly understood	Basic submission knowledge	Minimal distribution knowledge
CI/CD Pipeline	Fully automated, optimized	Working automation	Basic pipeline	Manual processes
Communication	Clear explanations, excellent detail	Generally clear, good structure	Readable, lacks depth	Disorganized
Practical Application	Real-world applicable solutions	Mostly practical with minor gaps	Basic applicability	Theoretical only

---

Deliverable Checklist

Submit as DOCX (6–8 pages):

- [ ] Task 1: Testing pyramid + strategy framework + learning outcomes
- [ ] Task 2: Unit testing code examples (3+ tests) + mocking patterns
- [ ] Task 3: Integration testing examples (Android + iOS) + database setup
- [ ] Task 4: UI testing examples (Espresso + XCUITest + Detox)

- [ ] Task 5: Google Play Console guide + keystore management + submission checklist
- [ ] Task 6: App Store Connect guide + TestFlight setup + review guidelines
- [ ] Task 7: CI/CD pipeline configuration + test automation
- [ ] Task 8: Testing strategy review + metrics + best practices

---

Resources & References

[1] Google. (2024). Android Testing Fundamentals. https://developer.android.com/training/testing

[2] Apple. (2024). iOS Testing Documentation. https://developer.apple.com/documentation/xctest

[3] JUnit. (2024). JUnit 4 Documentation. https://junit.org/junit4/

[4] Mockito. (2024). Mockito Framework. https://site.mockito.org/

[5] Google. (2024). Espresso Testing Framework. https://developer.android.com/training/testing/espresso

[6] Apple. (2024). XCUITest Framework. https://developer.apple.com/documentation/xctest/xcuitest

[7] Wix. (2024). Detox E2E Testing Framework. https://wix.github.io/Detox/

[8] Google. (2024). Google Play Console Documentation. https://support.google.com/googleplay/android-developer/

[9] Apple. (2024). App Store Connect Help. https://developer.apple.com/support/app-store-connect/

[10] Google. (2024). Firebase Test Lab Documentation. https://firebase.google.com/docs/test-lab

[11] Microsoft. (2024). GitHub Actions Documentation. https://docs.github.com/en/actions

[12] Android Authority. (2024). Mobile Testing Best Practices. https://www.androidauthority.com/testing/

Timeline Guidance

Self-Study (8–10 hours):

- Hours 1–2.5: Task 1–2 (testing fundamentals and unit testing)
- Hours 2.5–4.5: Task 3–4 (integration and UI testing)
- Hours 4.5–7: Task 5–6 (Android and iOS distribution)
- Hours 7–8.5: Task 7 (CI/CD automation)

- Hours 8.5–10: Task 8 (strategy and best practices)

Online Sessions (1.5–2 hours):

- 0.5 hours: Testing strategy and pyramid discussion
- 0.5 hours: Framework demonstrations (Espresso, XCUITest)
- 0.5 hours: Distribution and submission walkthrough
- 0.5 hours: Q&A and optimization discussion

Extension Tasks (Optional)

1. Implement test-driven development for entire feature
2. Create performance regression testing framework
3. Build testing dashboard with metrics visualization
4. Conduct comparative testing across frameworks
5. Implement advanced mocking strategies for complex scenarios
6. Create custom test helpers and utilities library
7. Design multi-device testing strategy
8. Develop automated accessibility testing suite