Mobile Programming

Testing & CI/CD

Lecturer: Ali Najimi
Faculty of Computer Engineering
Sharif University of Technology
Fall 2027

Unit Tests | Widget Tests | Integration | GitHub Actions



Table of Contents

1. Why Testing Matters
2. Flutter Testing Pyramid
3. Unit Testing
4. Widget Testing
5. Integration Testing
6. Mocking & Dependency Injection
7. Testing State Management
8. Golden Tests
9. CI/CD with GitHub Actions
10. Code Coverage & Quality
11. Mini Lab: Test a Riverpod App
12. Summary & Key Takeaways

Why Testing Matters

Untested code is broken code. Mobile apps are especially vulnerable to regressions.

Why test mobile apps?

- **Device fragmentation:** Thousands of screen sizes, OS versions.
- **Network unreliability:** Must handle offline, slow, and error states.
- **State complexity:** Navigation, lifecycle, background/foreground.
- **Regression prevention:** Changes in one feature break another.
- **Confidence to refactor:** Tests catch what manual testing misses.

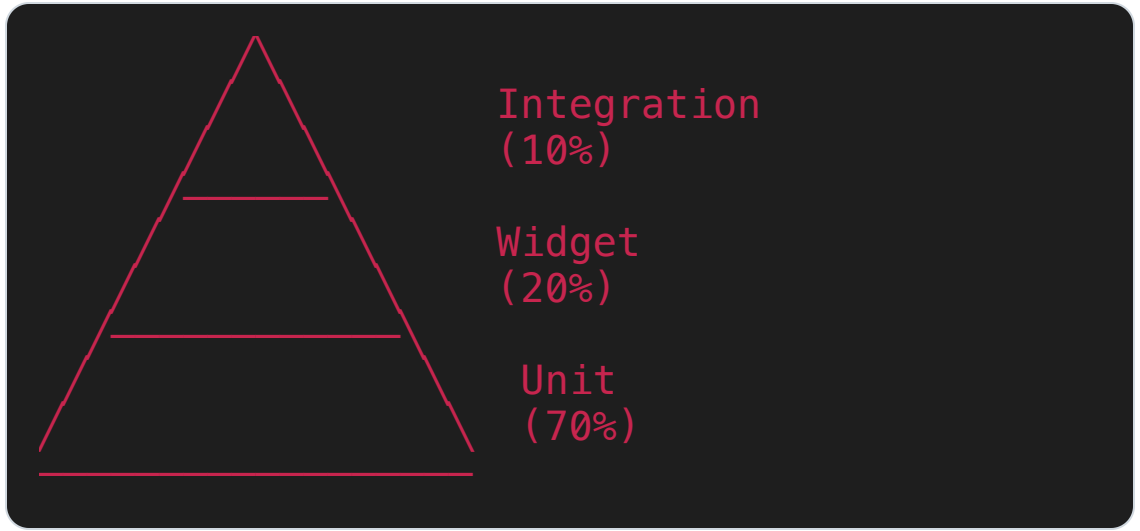
Testing ROI:

- Finding a bug in development costs **1x** to fix.
- Finding a bug in QA costs **10x**.
- Finding a bug in production costs **100x**.

Flutter Testing Pyramid

Three Levels:

1. **Unit Tests (70%)**
   - Test a single function, method, or class.
   - Fast, isolated, no UI.
   - `test/` directory.
2. **Widget Tests (20%)**
   - Test a single widget's rendering and interaction.
   - Medium speed, no full app.
   - `test/` directory.
3. **Integration Tests (10%)**
   - Test the full app running on a



Unit Testing

Test pure Dart logic — models, repositories, business logic.

```dart
// test/models/user_test.dart
import 'package:flutter_test/flutter_test.dart';

void main() {
  group('User', () {
    test('fromJson creates correct User', () {
      final json = {'id': 1, 'name': 'Ali', 'email': 'ali@example.com'};
      final user = User.fromJson(json);

      expect(user.id, 1);
      expect(user.name, 'Ali');
      expect(user.email, 'ali@example.com');
    });

    test('toJson produces correct map', () {
      final user = User(id: 1, name: 'Ali', email: 'ali@example.com');
      final json = user.toJson();

      expect(json['id'], 1);
      expect(json['name'], 'Ali');
    });
```

Unit Testing — Async & Errors

```dart
group('UserRepository', () {
  late UserRepository repo;
  late MockApiClient mockApi;

  setUp(() {
    mockApi = MockApiClient();
    repo = UserRepository(mockApi);
  });

  test('getAll returns list of users', () async {
    // Arrange
    when(() => mockApi.get('/users')).thenAnswer(
      (_) async => Response(
        data: [{'id': 1, 'name': 'Ali'}],
        statusCode: 200,
        requestOptions: RequestOptions(),
      ),
    );

    // Act
    final users = await repo.getAll();

    // Assert
    expect(users.length, 1);
    expect(users.first.name, 'Ali');
  });

  test('getAll throws on network error', () async {
    when(() => mockApi.get('/users')).thenThrow(DioException(
      type: DioExceptionType.connectionError,
      requestOptions: RequestOptions(),
    ));

    expect(() => repo.getAll(), throwsA(isA<NetworkException>()));
  });
});
```

Widget Testing

Test how widgets render and respond to user interaction.

```dart
testWidgets('CounterPage increments on tap', (tester) async {
  // Arrange: Load the widget
  await tester.pumpWidget(const MaterialApp(home: CounterPage()));

  // Assert: Initial state
  expect(find.text('Count: 0'), findsOneWidget);
  expect(find.text('Count: 1'), findsNothing);

  // Act: Tap the button
  await tester.tap(find.byType(ElevatedButton));
  await tester.pump(); // Rebuild after state change

  // Assert: Updated state
  expect(find.text('Count: 0'), findsNothing);
  expect(find.text('Count: 1'), findsOneWidget);
});

testWidgets('Shows loading indicator', (tester) async {
  await tester.pumpWidget(const MaterialApp(home: UserListPage()));
  expect(find.byType(CircularProgressIndicator), findsOneWidget);
});
```

Integration Testing

Test the full app on a real device or emulator.

```dart
// integration_test/app_test.dart
import 'package:integration_test/integration_test.dart';

void main() {
  IntegrationTestWidgetsFlutterBinding.ensureInitialized();

  group('App Integration Tests', () {
    testWidgets('Full login flow', (tester) async {
      await tester.pumpWidget(const MyApp());
      await tester.pumpAndSettle();

      // Enter credentials
      await tester.enterText(
        find.byType(TextFormField).first, 'ali@example.com');
      await tester.enterText(
        find.byType(TextFormField).last, 'password123');

      // Tap login
      await tester.tap(find.byType(ElevatedButton));
      await tester.pumpAndSettle();

      // Verify home screen loaded
      expect(find.text('Welcome'), findsOneWidget);
    });
```

Mocking & Dependency Injection

Use **mocktail** (or **mockito**) to create test doubles.

```dart
import 'package:mocktail/mocktail.dart';

// Create a mock
class MockUserRepository extends Mock implements UserRepository {}
class MockApiClient extends Mock implements ApiClient {}

void main() {
  late MockUserRepository mockRepo;

  setUp(() {
    mockRepo = MockUserRepository();
    // Register fallback values for complex types
    registerFallbackValue(User(id: 0, name: '', email: ''));
  });

  test('Bloc emits loaded state on success', () async {
    // Arrange
    final users = [User(id: 1, name: 'Ali', email: 'ali@example.com')];
    when(() => mockRepo.getAll()).thenAnswer((_) async => users);

    // Act
    final bloc = UserBloc(mockRepo);
    bloc.add(LoadUsers());

    // Assert
    await expectLater(
      bloc.stream,
      emitsInOrder([isA<UserLoading>(), UserLoaded(users)]),
    );
```

Testing State Management (Riverpod)

```dart
// Test a Riverpod provider
void main() {
  test('counterProvider starts at 0 and increments', () async {
    // Create a container with overrides
    final container = ProviderContainer();

    // Read initial state
    expect(container.read(counterProvider), 0);

    // Increment
    container.read(counterProvider.notifier).state++;
    expect(container.read(counterProvider), 1);

    container.dispose();
  });

  test('asyncUsersProvider loads users', () async {
    final container = ProviderContainer(
      overrides: [
        userRepositoryProvider.overrideWithValue(MockUserRepository()
          ..getAllReturns([User(id: 1, name: 'Ali', email: 'a@b.com')])),
      ],
    );

    final asyncValue = await container.read(asyncUsersProvider.future);
    expect(asyncValue.length, 1);
    expect(asyncValue.first.name, 'Ali');

    container.dispose();
  });
}
```

CI/CD with GitHub Actions

Automate testing, building, and deployment on every push.

```yaml
# .github/workflows/flutter-ci.yml
name: Flutter CI

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

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: subosito/flutter-action@v2
        with:
          flutter-version: '3.22.0'
      - run: flutter pub get
      - run: flutter analyze
      - run: flutter test --coverage
      - uses: codecov/codecov-action@v3
        with:
          files: coverage/lcov.info

  build-android:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: subosito/flutter-action@v2
      - run: flutter build apk --release --obfuscate
      - uses: actions/upload-artifact@v4
```

Code Coverage & Quality

Measure what matters:

- **Code Coverage:** Percentage of code exercised by tests.
  - Target: **80%+** for business logic.
  - Run: `flutter test --coverage`
  - Generate HTML: `genhtml coverage/lcov.info -o coverage/html`
- **Static Analysis:** `flutter analyze` catches issues before runtime.
- **Dart Code Metrics:** `dart_code_metrics` package for cyclomatic complexity, naming conventions.

```
# analysis_options.yaml
include: package:flutter_lints/flutter.yaml

linter:
  rules:
    prefer_const_constructors: true
    avoid_print: true
    no_leading_underscores_for_local_identifiers: true
```

Mini Lab

Test a Riverpod App

Objective: Write comprehensive tests for a Riverpod-powered app.

Requirements:

1. Unit tests for `User` model (`fromJson`, `toJson`).
2. Unit tests for `UserRepository` with mocked API client.
3. Widget test for `UserListPage` (loading → data → error states).
4. Provider test for `asyncUsersProvider`.
5. Run all tests with coverage.
6. Verify coverage ≥ 80% for `lib/`.

Summary: Key Takeaways

Testing Pyramid

- 70% Unit — Fast, isolated.
- 20% Widget — UI rendering.
- 10% Integration — Full app.

Tools

- `flutter test` — Unit & Widget.
- `integration_test` — E2E.
- `mocktail` — Test doubles.
- `coverage` — Measure coverage.

CI/CD

- GitHub Actions for automation.
- Run tests on every push.
- Build APK/IPA in CI.
- Upload artifacts for QA.

Best Practices

- Test behavior, not implementation.
- Use meaningful test names.
- Arrange → Act → Assert pattern.
- Keep tests independent.

Thank You! 🙌

Questions?

"A QA engineer walks into a bar.
Orders 1 beer. Orders 0 beers. Orders 99999999 beers.
Orders -1 beers. Orders a lizard. Orders NULL beers.
First real customer walks in and asks where the bathroom is.
The bar bursts into flames."