

Flutter Testing Mastery

Unit Tests, Widget Tests & Integration Tests

Why Testing Matters

Catch Bugs Early

- Find issues before users do
- Reduce debugging time

Enable Confident Refactoring

- Change code without fear
- Ensure existing functionality works

Documentation

- Tests serve as living documentation
- Show how code should be used

Save Money

- Cheaper to fix bugs early
- Reduce maintenance costs

Types of Tests in Flutter

Flutter is a GUI app, so we need to add **Widget Tests** additional to Unit/Integration tests.

Unit Tests

- Test individual functions/classes
- Fast and isolated
- No Flutter framework dependencies

Widget Tests

- Test UI components
- Use Flutter test framework
- Verify rendering and interactions

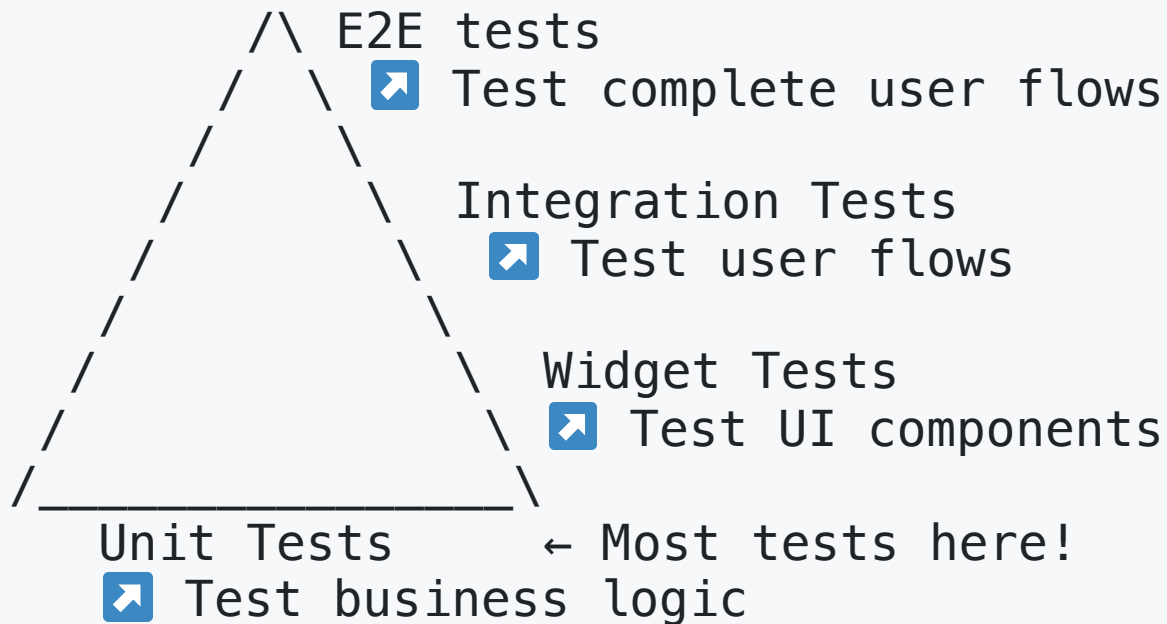
Integration Tests

- Test complete app flows
- Run on real devices/simulators
- End-to-end user scenarios

End 2 End (E2E) Tests

- Simulate **real user behavior** from start to finish
- Cover the **entire system**: UI → business logic → database → backend services
- Provide **highest confidence** that the app works in production
- **Slow and expensive** to run/maintain → use only for **critical paths** (e.g., login, checkout, payments)

Testing Pyramid



Testing Pyramid & Strategy: Many unit tests, fewer integration tests, minimal E2E tests

Real-world analogy:

Like testing individual car parts:

- **Unit Test** = Test the engine separately
- **Integration Test** = Test engine + transmission together
- **Widget Test** = Test the complete dashboard
- **End-to-End Test** = Test the complete car driving

Flutter Testing Commands

Run All Tests

Run all the tests in the `test/` directory.

```
flutter test
```

Run Specific Test File

```
flutter test test/unit/todo_model_test.dart
```


Run Integration Tests

```
flutter drive --target=integration_test/app_test.dart
```

Watch Mode (auto-run)

```
flutter test --watch
```

Test Coverage

Install lcov tool:

```
brew install lcov # Mac  
sudo apt-get install lcov # Linux/WSL2
```

Measuring Coverage:

```
flutter test --coverage  
genhtml coverage/lcov.info -o coverage/html  
open coverage/html/index.html # Mac command
```

Reading Coverage:

- **Lines:** Percentage of code lines executed
- **Functions:** Percentage of functions called
- **Branches:** Percentage of if/switch branches tested

Coverage Goals:

- **Unit tests:** 80-90%
- **Widget tests:** 60-70%
- **Integration tests:** Key user flows

Remember: 100% coverage \neq good tests!

Real-World Testing Strategy for Flutter

1. Start with Models

- Simple, pure functions
- High test coverage
- Build confidence

2. Add ViewModels

- Business logic testing
- State change verification
- Error handling

3. Test Key Widgets

- Critical UI components
- User interaction flows
- Visual appearance

4. Integration Tests

- Happy path scenarios
- Critical user journeys
- Platform-specific features

CI/CD Integration

We can automate testing process with GitHub Actions:

Automated Testing Pipeline:

```
# GitHub Actions example
- name: Run tests
  run: flutter test

- name: Check test coverage
  run: flutter test --coverage

- name: Run integration tests
  run: flutter drive --target=test_driver/app.dart
```

Continuous Testing

We need *Regression Testing* for this continuous testing.

Test Automation:

- Run tests on every commit (Regression Testing)
- Block merges with failing tests
- Monitor test performance

Test Maintenance:

- Update tests with code changes
- Remove obsolete tests
- Refactor test code

Team Practices:




- Code review includes tests
- Test-first development
- Pair programming on complex tests

Quality Gates:

- All tests must pass
- Minimum coverage threshold
- No critical issues in static analysis

TDD (Test-Driven Development)

Red-Green-Refactor Cycle:

1. **Red** : Write failing test
2. **Green** : Make test pass (minimal code)
3. **Refactor** : Improve code quality

```
// 1. Red - Write failing test
test('should calculate total price', () {
  expect(cart.totalPrice, 29.99);
});

// 2. Green - Make it pass
double get totalPrice => 29.99;

// 3. Refactor - Implement properly
double get totalPrice => items.fold(0, (sum, item) => sum + item.price);
```

Testing Best Practices



- Write tests first (TDD)
- Keep tests simple and focused
- Use meaningful test names
- Test both happy and sad paths
- Mock external dependencies
- Use proper test keys

✗ Don't:

- Test implementation details
- Write tests that depend on each other
- Mock everything
- Ignore failing tests
- Write tests just for coverage

Common Testing Mistakes

Testing Too Much

```
// Bad – Testing Flutter framework
expect(find.byType(Text), findsOneWidget);

// Good – Testing your logic
expect(viewModel.isValidEmail('test@example.com'), true);
```

Brittle Tests

```
// Bad – Depends on exact text
expect(find.text('There are 5 todos'), findsOneWidget);

// Good – Tests behavior
expect(viewModel.totalCount, 5);
```