# Flutter Testing: Unit, Widget & Integration

**A practical guide for developers to test Flutter apps effectively. Learn unit tests, widget tests, and integration tests to build bug-free, reliable applications.**

**Website reference:** haas.dev | https://dev-roast-app.vercel.app

---

## Introduction

Many Flutter developers skip testing, which leads to bugs, crashes, and unmaintainable apps.

This guide shows **how to test Flutter apps properly**, from small units of logic to full app workflows, ensuring your apps are stable, scalable, and professional-grade.

---

## Step 1: Understanding Testing Levels

1. **Unit Testing**
    - Tests **individual functions or classes**.
    - Fast, isolated, and doesn't require Flutter widgets.
2. **Widget Testing**
    - Tests **UI components in isolation**.
    - Ensures widgets render correctly and respond to inputs.
3. **Integration Testing**
    - Tests **full app workflows**.
    - Simulates real user interactions and API calls.

---

## Step 2: Setting Up Flutter Testing

1. Add dependencies in `pubspec.yaml`:

```
dev_dependencies:
  flutter_test:
    sdk: flutter
  mockito: ^5.0.0
  integration_test:
    sdk: flutter
```

2. Create a `test/` folder for unit & widget tests.
3. Create `integration_test/` folder for integration tests.

---

## Step 3: Unit Testing

1. Example: Testing a simple calculator function

```
int add(int a, int b) => a + b;

void main() {
  test('Addition works correctly', () {
    expect(add(2, 3), 5);
    expect(add(-1, 1), 0);
  });
}
```

2. **Best practices:**
    - Test **business logic** only.
    - Use **mock objects** for dependencies.
    - Keep tests **fast and isolated**.

---

## Step 4: Widget Testing

1. Example: Testing a button widget

```
testWidgets('Counter increments smoke test', (WidgetTester tester) async {
  await tester.pumpWidget(MyApp());

  // Verify initial value
  expect(find.text('0'), findsOneWidget);

  // Tap the increment button
  await tester.tap(find.byIcon(Icons.add));
  await tester.pump();

  // Verify incremented value
  expect(find.text('1'), findsOneWidget);
});
```

2. **Tips:**
    - Use `pumpWidget` to render your widget.
    - Use `tester.tap` and `tester.enterText` to simulate interactions.
    - Check the UI updates correctly after state changes.

---

## Step 5: Integration Testing

1. Example: Testing a login flow

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

void main() {
  IntegrationTestWidgetsFlutterBinding.ensureInitialized();

  testWidgets('Login flow test', (WidgetTester tester) async {
    await tester.pumpWidget(MyApp());

    await tester.enterText(find.byKey(Key('emailField')), 'test@example.com');
    await tester.enterText(find.byKey(Key('passwordField')), 'password123');
    await tester.tap(find.byKey(Key('loginButton')));
    await tester.pumpAndSettle();

    expect(find.text('Welcome'), findsOneWidget);
  });
}
```

2. **Best practices:**
    - Simulate real user flows.
    - Test important app paths (login, navigation, API interactions).
    - Keep integration tests **less frequent but thorough**.

---

## Step 6: Mocking & Dependency Injection

1. Use **Mockito** to mock repositories, APIs, and services.

```
class MockUserRepository extends Mock implements UserRepository {}
```

2. Inject mocks into your ViewModel or use `Provider` to replace real instances.

2. Inject mocks into your ViewModel or use Provider to replace real instances.
3. Benefits:
    - Isolate tests from network or database.
    - Test edge cases reliably.

---

## Step 7: Organizing Tests

1. Separate tests by type:

```
test/
 ├─ unit/
 ├─ widget/
integration_test/
 ├─ login_flow_test.dart
 ├─ todo_app_test.dart
```

2. Name tests **clearly** to describe functionality.
3. Keep tests **fast, small, and readable**.

---

## Step 8: Mini Project Example

**Objective:** Test a Flutter To-Do app

1. **Unit Test:** Validate `Task` entity logic (e.g., markComplete()).
2. **Widget Test:** Verify `AddTaskForm` displays errors correctly.
3. **Integration Test:** Add a task via UI and check it appears in task list.

   This ensures **all app layers are tested**, and bugs are caught early.

---

## Summary / Key Takeaways

- Testing ensures **bug-free, maintainable apps**.
- Use **unit tests** for logic, **widget tests** for UI, and **integration tests** for workflows.
- Mock dependencies to isolate tests.
- Organize tests by type for clarity and efficiency.
- Regular testing builds confidence in app stability.

---

Visit **haas.dev** for more resources and guides.
https://dev-roast-app.vercel.app

---