Get started

Samples & tutorials

Development

- User interface
- Data & backend
- Accessibility & internationalization
- Platform integration
- Packages & plugins
- Add Flutter to existing app
- Tools & techniques
- Migration notes

Testing & debugging

- Debugging tools
- Debugging apps programmatically
- Using an OEM debugger
- Flutter's build modes
- Handling errors
- Testing

Performance & optimization

Deployment

Resources

Reference

- Widget index
- API reference
- Package site

Testing Flutter apps

Contents

The more features your app has, the harder it is to test manually. Automated tests help ensure that your app performs correctly before you publish it, while retaining your feature and bug fix velocity.

Automated testing falls into a few categories:

- A *unit test* tests a single function, method, or class.
- A *widget test* (in other UI frameworks referred to as *component test*) tests a single widget.
- An *integration test* tests a complete app or a large part of an app.

Generally speaking, a well-tested app has many unit and widget tests, tracked by code coverage, plus enough integration tests to cover all the important use cases. This advice is based on the fact that there are trade-offs between different kinds of testing, see below.

	Unit	**Widget**	**Integration**
Confidence	Low	Higher	Highest
Maintenance cost	Low	Higher	Highest
Dependencies	Few	More	Most
Execution speed	Quick	Quick	Slow

Unit tests

A *unit test* tests a single function, method, or class. The goal of a unit test is to verify the correctness of a unit of logic under a va of conditions. External dependencies of the unit under test are generally mocked out. Unit tests generally don't read from or write disk, render to screen, or receive user actions from outside the process running the test.

Recipes

- An introduction to unit testing
- Mock dependencies using Mockito

Widget tests

A *widget test* (in other UI frameworks referred to as *component test*) tests a single widget. The goal of a widget test is to verify th the widget's UI looks and interacts as expected. Testing a widget involves multiple classes and requires a test environment that provides the appropriate widget lifecycle context.

For example, the Widget being tested should be able to receive and respond to user actions and events, perform layout, and instantiate child widgets. A widget test is therefore more comprehensive than a unit test. However, like a unit test, a widget test's environment is replaced with an implementation much simpler than a full-blown UI system.

Recipes

- An introduction to widget testing
- Find widgets
- Tap, drag, and enter text

Integration tests

An *integration test* tests a complete app or a large part of an app. The goal of an integration test is to verify that all the widgets ar services being tested work together as expected. Furthermore, you can use integration tests to verify your app's performance.

Generally, an *integration test* runs on a real device or an OS emulator, such as iOS Simulator or Android Emulator. The app under t is typically isolated from the test driver code to avoid skewing the results.

Recipes

- An introduction to integration testing
- Handle scrolling
- Performance profiling

Continuous integration services

Continuous integration (CI) services allow you to run your tests automatically when pushing new code changes. This provides tin feedback on whether the code changes work as expected and do not introduce bugs.

For information on running tests on various continuous integration services, see the following:

- Continuous delivery using fastlane with Flutter
- Test Flutter apps on Travis
- Test Flutter apps on Cirrus
- Codemagic CI/CD for Flutter
- Flutter CI/CD with Bitrise