

# Programmazione Mobile

**Nicola Noviello**

[nicola.noviello@unimol.it](mailto:nicola.noviello@unimol.it)

Corso di Laurea in Informatica  
Dipartimento di Bioscienze e Territorio  
Università degli Studi del Molise  
Anno 2023/2024

# Lezione: Test in Flutter

- Unit testing
- Widget testing
- Integration testing



# Testing di App Flutter

# Test di App Flutter

Più un'App cresce in termini di funzionalità, più aumentano gli sviluppatori che lavorano sullo stesso progetto, più è difficile testare manualmente la propria applicazione. I test automatici aiutano a garantire che l'App funzioni correttamente prima di pubblicarla, mantenendo al contempo la velocità di implementazione di nuove funzionalità e correzioni di bug.

I test automatizzati si suddividono in più categorie:

- **Unit Test:** testa una singola funzione, metodo o classe
- **Widget Test** (in altri framework UI noto come *component test*): testa un singolo widget
- **Integration Test:** testa un'App completa o una grande parte di un'App

In generale, un'App ben testata ha molti Unit Test e Widget Test, tali da raggiungere un'alta "code coverage", più un numero sufficiente di integration test per coprire tutti i casi d'uso critici e importanti.



# Schemi comparativi sui test: Confidence (fiducia)

- **Unit Test:** *Bassa fiducia*. I test unitari testano singole funzioni, metodi o classi, quindi non garantiscono che l'intero sistema funzioni correttamente insieme
- **Widget Test:** *Fiducia più alta*. Testando singole componenti UI (widget), forniscono una maggiore garanzia che le parti dell'interfaccia utente funzionino correttamente
- **Integration Test:** *Massima fiducia*. Testando grandi porzioni dell'App o l'intera App, offrono la massima sicurezza che tutte le parti dell'App funzionino correttamente *insieme*



# Schemi comparativi sui test: Costi di manutenzione

- **Unit Test:** *Basso*. Poiché testando piccole parti del codice, i test unitari sono generalmente più facili da mantenere
- **Widget Test:** *Più alto*. I Widget Test richiedono più manutenzione rispetto agli Unit Test perché coinvolgono componenti UI che sono notoriamente più complesse
- **Integration Test:** *Massimo*. Gli Integration Test sono i più costosi da mantenere poiché testano ampie porzioni del sistema e possono rompersi facilmente quando l'App cambia



# Schemi comparativi sui test: Dipendenze

- **Unit Test:** *Poche*. Gli Unit Test hanno poche dipendenze perché si limitano a verificare il funzionamento di piccole unità di codice
- **Widget Test:** *Di più*. I Widget Test hanno più dipendenze rispetto agli Unit Test, poiché coinvolgono funzionalità più complesse o parti dell'interfaccia utente
- **Integration Test:** *Massime*. Gli Integration Test hanno il maggior numero di dipendenze poiché coinvolgono ampie porzioni di codice o l'intera App.



# Schemi comparativi sui test: Velocità di esecuzione

- **Unit Test:** *Veloce*. Gli Unit Test sono generalmente molto rapidi da eseguire.
- **Widget Test:** *Veloce*. Anche i Widget Test sono relativamente rapidi, sebbene un po' più lenti rispetto agli Unit Test
- **Integration Test:** *Lenta*. Gli Integration Test sono i più lenti da eseguire a causa della loro complessità e del numero di dipendenze coinvolte



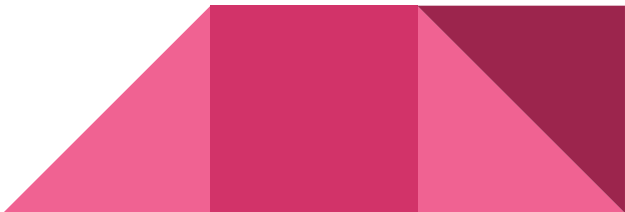


# Introduzione agli Unit Testing

# Aggiungere le dipendenze “test”

Il pacchetto “test” fornisce la funzionalità di base per scrivere test in Dart

```
nico@Mac-mini-di-Nico unittest % flutter pub add dev:test
Resolving dependencies...
Downloading packages...
+ _fe_analyzer_shared 67.0.0 (68.0.0 available)
+ analyzer 6.4.1 (6.5.0 available)
+ args 2.5.0
+ convert 3.1.1
+ coverage 1.8.0
+ crypto 3.0.3
+ file 7.0.0
...
nico@Mac-mini-di-Nico unittest %
```



## lib/counter.dart

```
class Counter {  
  int value = 0;  
  
  void increment() => value++;  
  
  void decrement() => value--;  
}
```

# test/counter\_test.dart

```
import 'package:unittest/counter.dart';  
import 'package:test/test.dart';  
  
void main() {  
  test('Il valore del counter dovrebbe essere incrementato', () {  
    final counter = Counter();  
  
    counter.increment();  
  
    expect(counter.value, 1);  
  });  
}
```

# test/test.dart

```
import 'package:unittest/counter.dart';
import 'package:test/test.dart';

void main() {
  group('Test start, incremento e decremento', () {
    test('Il valore dovrebbe essere 0', () {
      expect(Counter().value, 0);
    });

    test('Il valore del counter dovrebbe essere incrementato', () {
      final counter = Counter();

      counter.increment();

      expect(counter.value, 1);
    });

    test('Il valore del counter dovrebbe essere decrementato', () {
      final counter = Counter();

      counter.decrement();

      expect(counter.value, -1);
    });
  });
}
```

La funzione **group** permette di testare in contemporanea diversi metodi correlati eseguendo un solo test

# Esecuzione del test singolo

```
nico@Mac-mini-di-Nico unittest % flutter test test/counter_test.dart
```

```
00:01 +1: All tests passed!
```

```
nico@Mac-mini-di-Nico unittest %
```

# Esecuzione del test multiplo

```
nico@Mac-mini-di-Nico unittest % flutter test test/counter_test.dart
```

```
00:01 +1: All tests passed!
```

```
nico@Mac-mini-di-Nico unittest % flutter test test/test.dart
```

```
00:01 +3: All tests passed!
```

```
nico@Mac-mini-di-Nico unittest % flutter test --plain-name "Test start,  
incremento e decremento"
```

# Esercizio

Data la classe nella slide successiva, costruite un test.

Domanda per la classe: quali casistiche implementereste?



# student.dart

```
class Student {  
  String name;  
  int age;  
  List<int> grades;  
  
  Student({required this.name, required this.age, required this.grades});  
  
  double get averageGrade {  
    return grades.reduce((a, b) => a + b) / grades.length;  
  }  
  
  bool get isPassing {  
    return averageGrade >= 18;  
  }  
}  
  
// suggerimento, oggetto Student... Student(name: 'Nicola Test', age: 25, grades: [18, 24, 30]);
```

Cosa fa ".reduce"?

# Problema: e se lo Unit Test prevede un servizio terzo?

A volte, gli Unit Test possono dipendere da classi che recuperano dati da servizi web o database live.

Chiaramente questa condizione non è tra le più comode per diversi motivi:

- Chiamare servizi web o database rallenta l'esecuzione dei test.
- Un test che passa potrebbe iniziare a fallire se un servizio web o un database restituisce risultati inattesi. Questo è noto come "test instabile" ("flaky test").

È difficile testare tutti i possibili scenari di successo e fallimento utilizzando un servizio web o un database live.

Pertanto, anziché fare affidamento su un servizio web o database live, puoi "mockare" queste dipendenze. I mock permettono di emulare un servizio web o un database live e di restituire risultati specifici a seconda della situazione.

Quindi cominciamo a costruire **a mano** i nostri mock...



# <https://pub.dev/packages/mockito>



The screenshot shows the Mockito package page on pub.dev. The page title is "mockito 5.4.4". It indicates it was published 5 months ago and is compatible with Dart 3. The page includes tabs for SDK (Dart, Flutter), Platform (Android, iOS, Linux, MacOS, Web, Windows), and a "1.2K" like count. A "Readme" tab is selected, showing the package's purpose as a mock library for Dart. It mentions Mockito 5.0.0's support for Dart's null safety feature. A code snippet is provided for generating mock classes using the @GenerateNiceMocks annotation. The right sidebar shows statistics (1263 likes, 130 pub points, 99% popularity), publisher information (@dart.dev), metadata, repository (GitHub), documentation, license (Apache-2.0), and dependencies.

mockito 5.4.4

Published 5 months ago • @dart.dev (Dart 3 compatible)

SDK DART FLUTTER PLATFORM ANDROID IOS LINUX MACOS WEB WINDOWS 1.2K

Readme Changelog Example Installing Versions Scores

Try Dart CLI Rating pub 5.4.4 publisher: dart.dev

Mock library for Dart inspired by Mockito.

### Let's create mocks

Mockito 5.0.0 supports Dart's new null safety language feature in Dart 2.12, primarily with code generation.

To use Mockito's generated mock classes, add a `build_runner` dependency in your package's `pubspec.yaml` file, under `dev_dependencies`; something like `build_runner: ^1.11.0`.

For alternatives to the code generation API, see the [NULL\\_SAFETY\\_README](#).

Let's start with a Dart library, `cat.dart`:

```
import 'package:mockito/annotations.dart';
import 'package:mockito/mockito.dart';

@GenerateNiceMocks([MockSpec<Cat>()])
import 'cat.mocks.dart';

// Real class
class Cat {
  String sound() => "Meow";
  bool eatFood(String food, {bool? hungry}) => true;
  Future<void> chew() async => print("Chewing...");
  int walk(List<String> places) => 7;
  void sleep() {}
  void hunt(String place, String prey) {}
  int lives = 9;
}

void main() {
  // Create mock object.
  var cat = MockCat();
}
```

By annotating the import of a `.mocks.dart` library with `@GenerateNiceMocks`, you are directing Mockito's code generation to write a mock class for each "real" class listed, in a new library.

1263 130 99%  
LIKES PUB POINTS POPULARITY

Publisher  
@dart.dev

Metadata  
A mock framework inspired by Mockito with APIs for Fakes, Mocks, behavior verification, and stubbing.  
[Repository \(GitHub\)](#)  
[View/report issues](#)  
[Contributing](#)

Documentation  
[API reference](#)

License  
@ Apache-2.0 (LICENSE)

Dependencies  
analyzer, build, code\_builder, collection, dart\_style, matcher, meta, path, source\_gen, test, test\_api

More  
[Packages that depend on mockito](#)


Installiamo mockito con la seguente stringa:

```
flutter pub add http dev:mockito dev:build_runner
```

Ma cosa vuol dire quel *dev*: davanti al nome della libreria?

*dev*: indica le dipendenze di sviluppo, quei pacchetti necessari per lo sviluppo dell'applicazione, ma non per l'esecuzione dell'applicazione stessa. Questi pacchetti non sono inclusi nella build di distribuzione. Sono tipicamente utilizzati per i test, la generazione di codice e la documentazione. Nel nostro caso *http* è un package che dobbiamo usare anche in produzione ma abbiamo dipendenze specifiche di sviluppo come **mockito** per creare oggetti mock nei test, e **build\_runner** per la generazione di codice automatico.

Nel file **pubspec.yaml**, le dipendenze regolari sono elencate sotto **dependencies**, e le dipendenze di sviluppo sono elencate sotto **dev\_dependencies**.



# lib/main.dart

```
import 'dart:async';
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;

Future<Album> fetchAlbum() async {
  final response = await http
    .get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1'));

  if (response.statusCode == 200) {
    // If the server did return a 200 OK response,
    // then parse the JSON.
    return Album.fromJson(jsonDecode(response.body) as Map<String, dynamic>);
  } else {
    // If the server did not return a 200 OK response,
    // then throw an exception.
    throw Exception('Failed to load album');
  }
}
```

```
class Album {
  final int userId;
  final int id;
  final String title;
```

```
  const Album({
    required this.userId,
    required this.id,
```

Semplice acquisizione da json remoto

# lib/main.dart aggiornato

```
import 'dart:async';
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;

Future<Album> fetchAlbum(http.Client client) async {
  final response = await client
    .get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1'));

  if (response.statusCode == 200) {
    // If the server did return a 200 OK response,
    // then parse the JSON.
    return Album.fromJson(jsonDecode(response.body) as Map<String, dynamic>);
  } else {
    // If the server did not return a 200 OK response,
    // then throw an exception.
    throw Exception('Failed to load album');
  }
}

class Album {
  final int userId;
  final int id;
  final String title;

  const Album({
    required this.userId,
```

Fornisco un **http.Client** alla funzione. In questo modo possiamo scegliere di fornire l'http.Client corretto a seconda della situazione. Per i progetti Flutter viene fornito un http.IOCient., per le App web, viene fornito un http.BrowserClient. Per i test, viene fornito un mock http.Client.

Utilizza il client fornito per recuperare dati da internet, invece del **metodo statico http.get()**, che è difficile da mockare.

# test/fetch\_album\_test.dart

```
import 'package:http/http.dart' as http;
import 'package:unittest/main.dart';
import 'package:mockito/annotations.dart';

// Generate a MockClient using the Mockito package.
// Create new instances of this class in each test.
@GenerateMocks([http.Client])
void main() {}
```

Con il comando “dart run build\_runner build” creo dei mock da poter utilizzare nei test

# test/fetch\_album\_test.mocks.dart

```
/// See the documentation for Mockito's code generation for more information.
```

```
class MockClient extends _i1.Mock implements _i2.Client {
```

```
  MockClient() {  
    _i1.throwOnMissingStub(this);  
  }  
}
```

```
@override
```

```
_i3.Future<_i2.Response> head(  
  Uri? url, {  
    Map<String, String>? headers,  
  }) =>
```

```
  (super.noSuchMethod(  
    Invocation.method(  
      #head,  
      [url],  
      {#headers: headers},  
    ),  
    returnValue: _i3.Future<_i2.Response>.value(_FakeResponse_0(  
      this,  
      Invocation.method(  
        #head,  
        [url],  
        {#headers: headers},  
      ),  
    )),  
  ) as _i3.Future<_i2.Response>);
```

```
@override
```

```
_i3.Future<_i2.Response> get(  
  Uri? url, {  
    Map<String, String>? headers,  
  }) =>
```

```
  (super.noSuchMethod(  
    Invocation.method(  
      #get,  
      [url],  
      {#headers: headers},  
    ),  
    returnValue: _i3.Future<_i2.Response>.value(_FakeResponse_0(  
      this,  
      Invocation.method(  
        #get,  
        [url],  
      ),  
    )),  
  ) as _i3.Future<_i2.Response>);
```

Questo codice è autogenerato dal comando precedente...



# test/fetch\_album\_test.dart aggiornato

```
import 'package:flutter_test/flutter_test.dart';
import 'package:http/http.dart' as http;
import 'package:unittest/main.dart';
import 'package:mockito/annotations.dart';
import 'package:mockito/mockito.dart';

import 'fetch_album_test.mocks.dart';

// Generate a MockClient using the Mockito package.
// Create new instances of this class in each test.
@GenerateMocks([http.Client])
void main() {
  group('fetchAlbum', () {
    test('returns an Album if the http call completes successfully', () async {
      final client = MockClient();

      // Use Mockito to return a successful response when it calls the
      // provided http.Client.
      when(client)
        .get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1'))
        .thenAnswer((_) async => http.Response({'userId': 1, "id": 2, "title": "mock"}, 200));

      expect(await fetchAlbum(client), isA<Album>());
    });

    test('throws an exception if the http call completes with an error', () {
      final client = MockClient();

      // Use Mockito to return an unsuccessful response when it calls the
      // provided http.Client.
      when(client)
        .get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1'))
        .thenAnswer((_) async => http.Response('Not Found', 404));

      expect(fetchAlbum(client), throwsException);
    });
  });
}
```

Eseguo il test con **flutter test test/fetch\_album\_test.dart**

```
nico@Mac-mini-di-Nico unittest % flutter test test/fetch_album_test.dart
00:01 +2: All tests passed!
nico@Mac-mini-di-Nico unittest %
```

# Introduzione ai Widget Test

# Aggiungere anche per i Widget test le dipendenze “test”

Il pacchetto “test” fornisce la funzionalità di base per scrivere test in Dart

```
nico@Mac-mini-di-Nico widgettest % flutter pub add dev:test
```

```
Resolving dependencies...
```

```
Downloading packages...
```

```
+ _fe_analyzer_shared 67.0.0 (68.0.0 available)
```

```
+ analyzer 6.4.1 (6.5.0 available)
```

```
+ args 2.5.0
```

```
+ convert 3.1.1
```

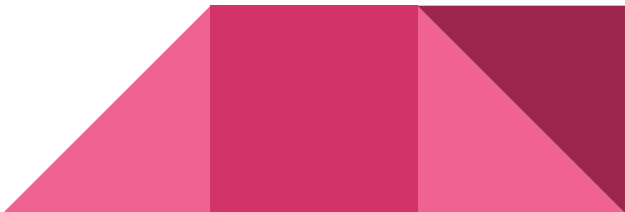
```
+ coverage 1.8.0
```

```
+ crypto 3.0.3
```

```
+ file 7.0.0
```

```
...
```

```
nico@Mac-mini-di-Nico widgettest %
```



# lib/main.dart

```
import 'package:flutter/material.dart';

void main() {
  runApp(const MyWidget(title: 'T', message: 'M'));
}

class MyWidget extends StatelessWidget {
  const MyWidget({
    super.key,
    required this.title,
    required this.message,
  });

  final String title;
  final String message;

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      home: Scaffold(
        appBar: AppBar(
          title: Text(title),
        ),
        body: Center(
          child: Text(message),
        ),
      ),
    );
  }
}
```

# test/widget\_test.dart

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

void main() {
  // Definisci un test. La funzione TestWidgets fornisce anche un WidgetTester
  // con cui lavorare. Il WidgetTester ti permette di costruire e interagire
  // con i widget nell'ambiente di test.
  testWidgets('MyWidget has a title and message', (tester) async {
    // Il Test va qui
  });
}
```

La funzione **testWidgets** permette di definire un test del widget e crea un **WidgetTester** con cui lavorare. (al momento è vuoto)

# test/widget\_test.dart

```
import 'package:flutter_test/flutter_test.dart';
import 'package:widgettest/main.dart';

void main() {
  // Definisci un test. La funzione TestWidgets fornisce anche un WidgetTester
  // con cui lavorare. Il WidgetTester ti permette di costruire e interagire
  // con i widget nell'ambiente di test.
  testWidgets('MyWidget has a title and message', (tester) async {
    await tester.pumpWidget(const MyWidget(title: 'T', message: 'M'));
  });
}
```

Costruiamo **MyWidget** all'interno dell'ambiente di test utilizzando il metodo **pumpWidget()** fornito da **WidgetTester**. Il metodo **pumpWidget** costruisce il widget fornito.

Il widget creato è un'istanza di **MyWidget** che visualizza "T" come titolo e "M" come messaggio.

# test/widget\_test.dart

```
import 'package:flutter_test/flutter_test.dart';
import 'package:widgettest/main.dart';

void main() {
  // Definisci un test. La funzione TestWidgets fornisce anche un WidgetTester
  // con cui lavorare. Il WidgetTester ti permette di costruire e interagire
  // con i widget nell'ambiente di test.
  testWidgets('MyWidget has a title and message', (tester) async {
    await tester.pumpWidget(const MyWidget(title: 'T', message: 'M'));
    // Creo dei finders per i widget che voglio testare
    final titleFinder = find.text('T');
    final messageFinder = find.text('M');

  });
}
```

Con un widget creato in ambiente di test, cerca nel tree dei widget il titolo e i widget di testo del messaggio utilizzando un **Finder** attraverso il metodo **find()** fornito dal pacchetto **flutter\_test**. Poiché si sta cercando un widget di tipo Text, usa il metodo `find.text()`.

# test/widget\_test.dart

```
import 'package:flutter_test/flutter_test.dart';
import 'package:widgettest/main.dart';

void main() {
  // Definisci un test. La funzione TestWidgets fornisce anche un WidgetTester
  // con cui lavorare. Il WidgetTester ti permette di costruire e interagire
  // con i widget nell'ambiente di test.
  testWidgets('MyWidget has a title and message', (tester) async {
    await tester.pumpWidget(const MyWidget(title: 'T', message: 'M'));
    // Creo dei finders per i widget che voglio testare
    final titleFinder = find.text('T');
    final messageFinder = find.text('M');
    expect(titleFinder, findsOneWidget);
    expect(messageFinder, findsOneWidget);
  });
}
```

Verifichiamo che i widget di testo del titolo e del messaggio appaiano sullo schermo utilizzando le costanti Matcher fornite da flutter\_test. Le classi Matcher sono una parte fondamentale del pacchetto di test e forniscono un modo comune per verificare che un determinato widget soddisfi le aspettative.



# Il risultato

```
nico@Mac-mini-di-Nico widgettest % flutter test test/widget_test.dart
```

```
00:04 +1: All tests passed!
```

```
nico@Mac-mini-di-Nico widgettest % flutter test test/widget_test.dart
```

```
00:01 +0: MyWidget has a title and message
```

```
══| EXCEPTION CAUGHT BY FLUTTER TEST FRAMEWORK |
```

The following TestFailure was thrown running a test:

Expected: exactly one matching candidate

Actual: \_TextWidgetFinder:<Found 0 widgets with text "X": []>



Which: means none were found but one was expected



# Altri Matcher che possiamo utilizzare

## Additional Matchers

In addition to `findsOneWidget`, `flutter_test` provides additional matchers for common cases.

`findsNothing`

Verifies that no widgets are found.

`findsWidgets`

Verifies that one or more widgets are found.

`findsNWidgets`

Verifies that a specific number of widgets are found.

`matchesGoldenFile`

Verifies that a widget's rendering matches a particular bitmap image ("golden file" testing).

<https://docs.flutter.dev/cookbook/testing/widget/finders>

The screenshot shows a web browser displaying the Flutter documentation page for 'Find widgets'. The page has a dark blue header with the Flutter logo and navigation links. A banner below the header announces Flutter 3.22 and Dart 3.4. A left sidebar contains a table of contents with categories like 'Accessibility & internationalization', 'Beyond UI', 'Platform integration', 'Packages & plugins', and 'Testing & debugging'. The main content area is titled 'Find widgets' and includes an introduction, a list of steps, and a code example. A right sidebar shows a 'Contents' section with links to specific topics. At the bottom, a Google cookie notice is visible.

Flutter

Multi-Platform Development Ecosystem Showcase Docs

Flutter 3.22 is live! Check out the Flutter 3.22 and Dart 3.4 announcement, the 3.22 technical blog post. Also check out what's new on the website.

## Find widgets

Cookbook > Testing > Widget > Find widgets

To locate widgets in a test environment, use the `Finder` classes. While it's possible to write your own `Finder` classes, it's generally more convenient to locate widgets using the tools provided by the `flutter_test` package.

During a `flutter run` session on a widget test, you can also interactively tap parts of the screen for the Flutter tool to print the suggested `Finder`.

This recipe looks at the `find` constant provided by the `flutter_test` package, and demonstrates how to work with some of the `Finders` it provides. For a full list of available finders, see the [CommonFinders documentation](#).

If you're unfamiliar with widget testing and the role of `Finder` classes, review the [Introduction to widget testing](#) recipe.

This recipe uses the following steps:

1. Find a `Text` widget.
2. Find a widget with a specific `Key`.
3. Find a specific widget instance.

### 1. Find a `Text` widget

In testing, you often need to find widgets that contain specific text. This is exactly what the `find.text()` method is for. It creates a `Finder` that searches for widgets that display a specific `String` of text.

```
testWidgets('finds a Text widget', (tester) async {  
  // Build an App with a Text widget that displays the letter 'H'.  
  await tester.pumpWidget(const MaterialApp(  
    home: Scaffold(  
      body: Text('H'),  
    ),  
  ));  
})
```

Contents

- 1. Find a `Text` widget
- 2. Find a widget with a specific `Key`
- 3. Find a specific widget instance

Summary

Complete example

Google uses cookies to deliver its services, to personalize ads, and to analyze traffic. You can adjust your privacy controls anytime in your [Google settings](#). [Learn more](#).

Okay

# Esercizio

Dato il widget nella slide successiva costruire un test

# main.dart

```
import 'package:flutter/material.dart';

void main() {
  // runApp(const MyWidget(title: 'T', message: 'M'));
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'My App',
      home: Scaffold(
        appBar: AppBar(
          title: const Text('My App Bar', key: Key('appBarTitle')),
        ),
        body: const Center(
          child: Text('Hello World'),
        ),
      ),
    );
  }
}
```

# Integration test

# Integration test

Come abbiamo visto all'inizio della nostra lezione, gli Unit Test e i Widget test servono a validare classi, widget e funzioni. Il loro scopo è quello di verificare che singoli “frammenti” di codice siano scritti bene, ma non sono in grado di valutare - ad esempio - se tutto il codice insieme funziona correttamente oppure se le performance di un'App sono buone o meno.

Per tutto questo ci sono gli Integration Test.

Per provarli, in maniera estremamente semplificata, cominciamo creando un nuovo progetto, lasciando nel main.dart il codice di default di Flutter.



# main.dart

```
...
class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Theme.of(context).colorScheme.inversePrimary,
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headlineMedium,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        key: const ValueKey('increment'),
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

La chiave **key** in Flutter viene utilizzata per identificare in modo univoco un widget all'interno dell'albero dei widget. In questo caso, **ValueKey('increment')** sta creando una chiave con il valore 'increment'.



# integration\_test/app\_test.dart

```
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';
import 'package:integrationtest/main.dart';
import 'package:integration_test/integration_test.dart';

void main() {
  IntegrationTestWidgetsFlutterBinding.ensureInitialized();
  group('end-to-end test', () {
    testWidgets('tap on the floating action button, verify counter',
      (tester) async {
        // Load app widget.
        await tester.pumpWidget(const MyApp());

        // Verify the counter starts at 0.
        expect(find.text('0'), findsOneWidget);

        // Finds the floating action button to tap on.
        final fab = find.byKey(const ValueKey('increment'));

        // Emulate a tap on the floating action button.
        await tester.tap(fab);

        // Trigger a frame.
        await tester.pumpAndSettle();

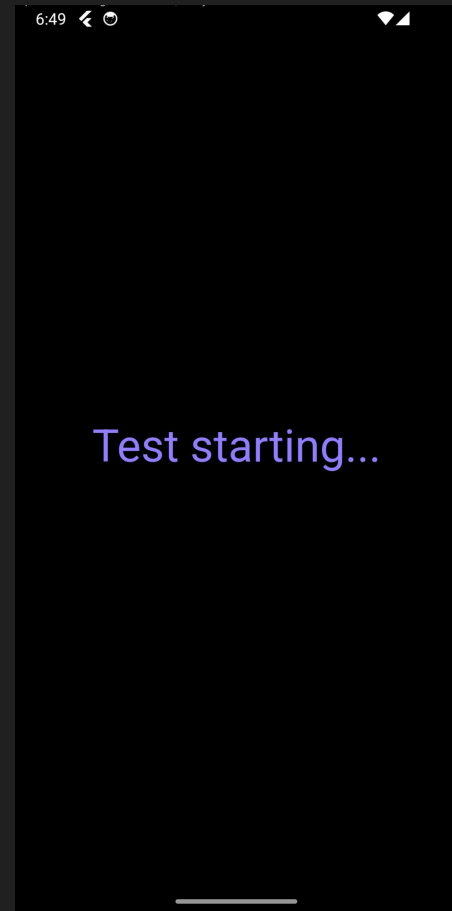
        // Verify the counter increments by 1.
        expect(find.text('1'), findsOneWidget);
      });
  });
}
```

È necessario installare il package `integration_test` con questo comando  
**`flutter pub add 'dev:integration_test:{"sdk":"flutter"}'`**

ATTENZIONE: gli integration test non vanno nella directory `test`, ma richiedono una specifica directory

# esecuzione del test

```
nico@Mac-mini-di-Nico integrationtest % flutter test integration_test/app_test.dart
00:00 +0: loading /Users/nico/flutter-proj/integrationtest/integration_test/app_test.dart
R00:09 +0: loading /Users/nico/flutter-proj/integrationtest/integration_test/app_test.dart
8,7s
✓ Built build/app/outputs/flutter-apk/app-debug.apk
00:10 +0: loading /Users/nico/flutter-proj/integrationtest/integration_test/app_test.dart
I00:14 +0: loading /Users/nico/flutter-proj/integrationtest/integration_test/app_test.dart
3,4s
00:20 +1: All tests passed!
nico@Mac-mini-di-Nico integrationtest %
```



# main.dart

```
class MyHomePage extends StatefulWidget {  
  const MyHomePage({super.key, required this.title});  
  
  final String title;  
  
  @override  
  State<MyHomePage> createState() => _MyHomePageState();  
}  
  
class _MyHomePageState extends State<MyHomePage> {  
  int _counter = 0;  
  
  void _incrementCounter() {  
    setState(() {  
      _counter++;  
      _counter++;  
    });  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        backgroundColor: Theme.of(context).colorScheme.inversePrimary,  
        title: Text(widget.title),  
      ),  
      body: Center(  
        child: Column(  
          mainAxisAlignment: MainAxisAlignment.center,
```

E se modificassi il codice del  
main.dart?

# integration\_test/app\_test.dart

```
00:21 +0: end-to-end test tap on the floating action button, verify counter
```

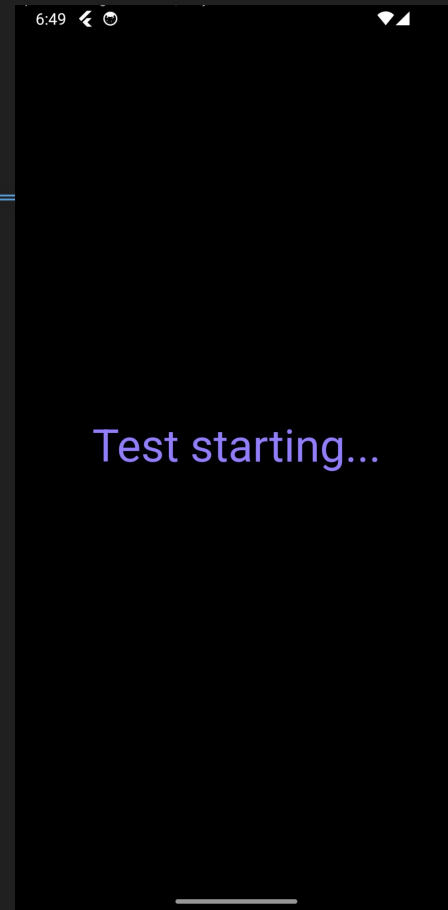
```
═══ EXCEPTION CAUGHT BY FLUTTER TEST FRAMEWORK ═══
```

```
The following TestFailure was thrown running a test:
```

```
Expected: exactly one matching candidate
```

```
  Actual: _TextWidgetFinder:<Found 0 widgets with text "1": []>
```

```
  Which: means none were found but one was expected
```



# Integration test - Performance

Quando si tratta di app mobili, le prestazioni sono fondamentali per l'esperienza dell'utente. Gli utenti si aspettano che le App scorrano in maniera fluida e che le animazioni siano senza scatti o fotogrammi saltati (jank).

Come possiamo garantire che la nostra App sia priva di jank su una vasta gamma di dispositivi?

Ci sono due opzioni:

- testare manualmente l'App su diversi dispositivi. Sebbene questo approccio possa funzionare per un'App di piccole dimensioni, diventa più oneroso man mano che l'app cresce
- eseguire un test di integrazione che svolge un compito specifico e registra una timeline delle prestazioni. Poi, esamina i risultati per determinare se una sezione specifica dell'App necessita di miglioramenti



# Creiamo un nuovo progetto partendo dal main.dart

```
import 'package:flutter/material.dart';

void main() {
  runApp(MyApp(
    items: List<String>.generate(10000, (i) => 'Item $i'),
  ));
}
```

```
class MyApp extends StatelessWidget {
  final List<String> items;

  const MyApp({super.key, required this.items});
```

```
  @override
  Widget build(BuildContext context) {
    const title = 'Long List';

    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: const Text(title),
        ),
        body: ListView.builder(
          key: const Key('long_list'),
          itemCount: items.length,
          itemBuilder: (context, index) {
            return ListTile(
              title: Text(
                items[index],
                key: Key('item_${index}_text'),
              ),
            );
          },
        ),
      ),
    );
  }
}
```

# test/widget\_test.dart

```
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';

import 'package:scrollwidget/main.dart';

void main() {
  testWidgets('finds a deep item in a long list', (tester) async {
    // Build our app and trigger a frame.
    await tester.pumpWidget(MyApp(
      items: List<String>.generate(10000, (i) => 'Item $i'),
    ));

    final listFinder = find.byType(Scrollable);
    final itemFinder = find.byKey(const ValueKey('item_50_text'));

    // Scroll until the item to be found appears.
    await tester.scrollUntilVisible(
      itemFinder,
      500.0,
      scrollable: listFinder,
    );

    // Verify that the item contains the correct text.
    expect(itemFinder, findsOneWidget);
  });
}
```

**await tester.pumpWidget(MyApp(...));** :  
Crea un'istanza del widget MyApp e lo rende la radice dell'albero dei widget. Il widget MyApp viene costruito con una lista di 10000 stringhe.

**final listFinder = find.byType(Scrollable);** : Crea un Finder che trova widget del tipo Scrollable.

**final itemFinder = find.byKey(const ValueKey('item\_50\_text'));** : Crea un Finder che trova un widget con la chiave ValueKey('item\_50\_text').

**await tester.scrollUntilVisible(...);**  
:Scorre il widget Scrollable fino a quando il widget con la chiave ValueKey('item\_50\_text') diventa visibile.

**expect(itemFinder, findsOneWidget);** :  
Verifica che il Finder trovi esattamente un widget. Se trova più widget o nessun widget, il test fallisce.