ENSE 375 - Software Testing and Validation - Laboratory

Lab 6: Widget Testing with Web / Mobile

University of Regina

Faculty of Engineering and Applied Science - Software Systems Engineering

Lab Instructor: Trevor Douglas

Introduction

Flutter provides a robust testing framework that allows developers to automate and validate various aspects of their app's behavior, including unit testing, widget testing, and integration testing. Testing in Flutter not only helps catch bugs early in the development cycle but also promotes code quality and facilitates a smoother user experience.

In this lab you will be introduced to Unit and Widget testing.

Background

Procedure

For the pre-lab, we are going to recreate the BMI Calculator but this time we are going to include automation testing.

Let's start by setting up the project again:

Paste this as your new main file...

▶ expand main.dart

in `main.dart`

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

void main() {
   runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
    );
  }
}
```

Create `lib/bmi.dart` and paste in the following:

▶ expand bmi.dart

in `bmi.dart`

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

class BMI {

    double calculateBMI(double height, double weight) {
        double heightInCM = height;
        double weightInKg = weight;
        double heightInM = heightInCM / 100;
        double heightSquared = heightInM * heightInM;
        double result = weightInKg / heightSquared;
        return result;
    }


}
```

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 variety of conditions. External dependencies of the unit under test are generally mocked out. Unit tests generally don't read from or write to disk, render to screen, or receive user actions from outside the process running the test.

All the test files in a Flutter app, except for integration tests, are placed in the test directory.

Create a new test file

First, you'll test the calculateBMI() method in the BMI class to verify that your bmi calculation is correct. By convention, the directory structure in the test directory mimics that in the lib directory and the Dart files have the same name with _test appended.

Create a bmi_test.dart file.

▶ Details
expand bmi_test.dart

in `bmi_test.dart`

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

void main() {
  group('Testing BMI', () {
    var bmi = BMI();

    test('Testing the BMI calculation', () {
      double bmiCalc = bmi.calculateBMI(175, 90);
      expect('29.39', bmiCalc.toStringAsFixed(2));
    });
  });
}
```

In our application it is primarily Widgets, so we should learn some Widget testing.

Widget Tests

Flutter widget tests are a type of test used to verify the visual appearance and behavior of individual Flutter widgets. These tests simulate user interactions and ensure that the widgets render correctly based on different states and inputs. Their primary purpose is to verify the correctness of Flutter widgets in terms of their appearance, layout, and interaction behaviors.

Tools and Libraries:

flutter_test Package: This Flutter package provides utilities and classes necessary for writing and executing widget tests.

WidgetTester Class: Central to widget testing, it allows for the creation of widget instances, interaction with widgets (such as tapping buttons or entering text), and inspecting widget properties during tests.

Common Use Cases

Stateful Widget Testing: Verifying the correct initialization and updating of UI elements based on state changes within a stateful widget.

Interaction Testing: Testing user interactions such as button taps, text input, scrolling behavior, etc., to ensure correct UI responses.

Layout Verification: Checking that widgets render correctly according to specified layout constraints and responsive design principles.

Testing Methodology

Setup and Teardown: Widget tests typically involve setting up the widget under test (pumpWidget), performing actions on it (tester.tap, tester.enterText, etc.), and then verifying expected outcomes (expect assertions).

Benefits of Widget Tests

Early Detection of UI Bugs: Catching UI-related issues early in development reduces the likelihood of them reaching users.

Enhanced Code Confidence: Ensures that widgets behave as expected across different states and interactions.

Facilitates Refactoring: Allows developers to refactor UI code confidently, knowing that tests will detect regressions.

Supports Rapid Iteration: Enables quick feedback on UI changes, aiding in iterative development and design refinement.

Build our applicaiton

Ok, we have to get our application to the point of last lab. Copy the following code:

▶ Details
expand bmi.dart

in `bmi.dart`

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

class BMI extends StatefulWidget {
  const BMI({super.key});

  @override
  State<BMI> createState() => _BMIState();
}

class _BMIState extends State<BMI> {
  final TextEditingController _heightController =
TextEditingController();
  final TextEditingController _weightController =
TextEditingController();

  double? _result;

  var _bmiVal;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('BMI Calculator'),
        centerTitle: true,
      ),
      body: Container(
        padding: EdgeInsets.symmetric(horizontal: 10.0),
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            TextField(
              controller: _heightController,
              keyboardType: TextInputType.number,
              decoration: InputDecoration(
                labelText: 'height in cm',
              ),
            ),
            SizedBox(height: 50),
            TextField(
              controller: _weightController,
              keyboardType: TextInputType.number,
              decoration: InputDecoration(
                labelText: 'weight in kg',
              ),
            ),
            SizedBox(height: 50),
            OutlinedButton(
              onPressed: calculateBMI,
              child: Text(
                "Calculate",
              ),
            ),
            SizedBox(height: 50),
            Text('Result'),
            Text(
              _result == null
                  ? "Enter Value"
                  : "${_result!.toStringAsFixed(2)}",
              style: TextStyle(
                color: Colors.redAccent,
                fontSize: 19.4,
                fontWeight: FontWeight.w500,
              ),
            ),
          ],
        ),
      ),
    );
  }

  void calculateBMI() {
    double heightInCM = double.parse(_heightController.text);
    double weightInKg = double.parse(_weightController.text);
    double heightInM = heightInCM / 100;
    double heightSquared = heightInM * heightInM;
    _result = weightInKg / heightSquared;
    setState(() {});
  }
}
```

Don't forget to put the home: parameter back in main.dart either!

▶ Details
expand main.dart

in `main.dart`

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

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: BMI(),
    );
  }
}
```

Create our tests

Now we are ready to test our 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.

The widget testing framework provides finders to find widgets, for example **text()**, **byType()**, and **byIcon()**. The framework also provides matchers to verify the results.

The api is found here

Lets verify the title of our application.

▶ Details
expand bmi_test.dart

in `bmi_test.dart`

```dart


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

import '../lib/bmi.dart';
```

```
void main() {
  group('Testing BMI', () {
    testWidgets('BMI page should have a certain Title', (tester) async {
      await tester.pumpWidget(MaterialApp(home: BMI()));

      final titleFinder = find.text('BMI Calculator');

      expect(titleFinder, findsOneWidget);
    });
  });
}
```

We have numerous text fields to verify also:

▶ Details
expand bmi_test.dart

in `bmi_test.dart`

```
    testWidgets('BMI page should height text', (tester) async {
      await tester.pumpWidget(const MaterialApp(home: BMI()));

      final titleFinder = find.text('height in cm');

      expect(titleFinder, findsOneWidget);
    });
```

The same method can be used to find all our **TextField** text.

Simulate User Interaction

Many widgets not only display information, but also respond to user interaction. This includes buttons that can be tapped, and **TextField** for entering text.

To test these interactions, you need a way to simulate them in the test environment. For this purpose, use the WidgetTester library.

The **WidgetTester** provides methods for entering text, tapping, and dragging.

- **enterText()**
- **tap()**
- **drag()**

In many cases, user interactions update the state of the app. In the test environment, Flutter doesn't automatically rebuild widgets when the state changes. To ensure that the widget tree is rebuilt after simulating a user interaction, call the **pump()** or **pumpAndSettle()** methods provided by the **WidgetTester**.

First we have to find the correct **TextField**

▶ Details
expand the bmi_test.dart

in `bmi_test.dart`

```
    //Find the Height field
      final heightField = find.ancestor(
        of: find.text('height in cm'),
        matching: find.byType(TextField),
      );

      //Enter a value
      await tester.enterText(heightField, "175");
      expect(find.text('175'), findsOneWidget);
```

Now enter a value of 90 into the weight field.

Since we have an **OutlinedButton** find it using **byType()** and give it a **tap()**.

Done forget to rebuild the **Widget** tree by calling the **pump()** method.

▶ Details
expand the bmi_test.dart

in `bmi_test.dart`

```
    //tap the button
    await tester.tap(find.byType(OutlinedButton));
    //Rebuild the Widget
    await tester.pump();
    //Check the result
    expect(find.text('29.39'), findsOneWidget);
```

Assignment

Just like in last lab, your requirements have changed and you must implement the Phase 1 and Phase 2 modifications. Now test your new Widgets with automated tests in your `bmi_test.dart` file as per the previous lab requirements.

Submission

Nothing to submit because you are using your GitHub repository.

References

Flutter:

Flutter Testing Code Lab

CommonFinders class

Testing Flutter apps

Dart:

S. Ford, The Dart Language: When Java and C# Aren't Sharp Enough, 2019

BMI Tutorial:

N. Singh, Flutter BMI Calculator App, 2020