Team Member Names	Pratik Pujari (2020300054) and Shivalik Pandita (2020300049)
Experiment No	9

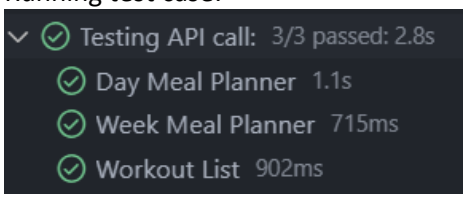
AIM:	To perform Unit testing
THEORY:	**What Is Unit Testing?** A unit test is a way of testing a unit - the smallest piece of code that can be logically isolated in a system. In most programming languages, that is a function, a subroutine, a method or property. The isolated part of the definition is important. In his book "Working Effectively with Legacy Code", author Michael Feathers states that such tests are not unit tests when they rely on external systems: “If it talks to the database, it talks across the network, it touches the file system, it requires system configuration, or it can't be run at the same time as any other test." Modern versions of unit testing can be found in frameworks like JUnit, or testing tools like TestComplete. Look a little further and you will find SUnit, the mother of all unit testing frameworks created by Kent Beck, and a reference in chapter 5 of The Art of Software Testing . Before that, it's mostly a mystery. I asked Jerry Weinberg about his experiences with unit testing -- "We did unit testing in 1956. As far as I knew, it was always done, as long as there were computers". Regardless of when and where unit testing began, one thing is for sure. Unit testing is here to stay. Let's look at some more practical aspects of unit testing. **What Do Unit Tests Look Like?** A unit can be almost anything you want it to be -- a line of code, a method, or a class. Generally though, smaller is better. Smaller tests give you a much more granular view of how your code is performing. There is also the practical aspect that when you test very small units, your tests can be run fast; like a thousand tests in a second fast. Consider this sample code: def divider (a, b) return a/b end Using Ruby, those small tests might look something like this: class smallTest < MiniTest::Unit::testCase def tiny_test @a=9 @b=3 assert_equal(3, divider(a, b)) end end This example is overly simple, but it gives you an idea of what I mean by small. Small tests also have the benefit of making it harder to cross systems -- from code into a database, or 3rd party system. Strictly speaking, there isn't anything wrong with crossing systems, but there are consequences like gradually slowing your tests. A few years ago I worked for a company where this crept into the test set, eventually we had thousands of tests, set up and tear down scrips for the database, and also a test suite that took hours to run. Start Unit Testing Now With a TestComplete Free Trial Download NowLearn More **Who Should Create The Unit Test Then?**

	In his book, Real Time Business Systems, Robert V. Head says "Frequently, unit testing is considered part of the programming phase, with the person that wrote the program...unit testing". That isn't because programmers hold the secret sauce to unit testing, it's because it makes sense. The programmer that wrote the prod code will likely know how to access the parts that can be tested easily and how to mock objects that can't be accessed otherwise. It's a time trade off. Other times, someone will come in after the fact and write tests to help create safe guards while they refactor or further develop that area of the code base. **What Can I Do With Them?** Hammers are great tools and can help you with lots of different jobs -- opening car windows or turning off alarm clocks. But, there are especially well suited to putting nails through hard surfaces. Unit tests are similar. They can do lots of different things, they should probably only do a few. **Test Driven Development** Test Driven Development, or TDD, is a code design technique where the programmer writes a test before any production code, and then writes the code that will make that test pass. The idea is that with a tiny bit of assurance from that initial test, the programmer can feel free to refactor and refactor some more to get the cleanest code they know how to write. The idea is simple, but like most simple things, the execution is hard. TDD requires a completely different mind set from what most people are used to and the tenacity to deal with a learning curve that may slow you down at first. **Checking Your Work** TDD isn't new, but at this point it is still mostly for the go getters. The rest of us are checking our work. Writing unit tests after you have written the production code may be a more traditional way of doing it, but it is no less useful. It's also something you're familiar with if you have been in a math class any time in the past ten years. After your work is checked and it is clear that the code is doing what you think it is doing, the value of the unit tests change a little bit. Tests that can be easily run with every build of your product act as change detection notifying you when code changes in unexpected ways. **Code Documentation** Code documentation is a drag, and it shows, mostly by how little code documentation gets written. Unit testing can make the documentation burden a little easier by encouraging better coding practices and also leaving behind pieces of code that describe what your product is doing. Rather than having to constantly feed the documentation beast with code change, you'll be updating a system of checks that is working for you. **Danger Zone** There are a few uses of unit testing that you will want to avoid when possible. Creating integration tests that cross system borders and touch databases or 3rd party systems can be done, but this quickly results in a test suite that takes longer and longer to run with each added test. There are plenty of test frameworks out there that specialize in higher level testing. If you want to test larger pieces of your product at a time, you might want to investigate those other frameworks. One other risky area is end to end tests. These usually require careful ordering, dependencies on other tests, and careful set up to get your system in a special 'test ready' state. Much like integration testing, there are plenty of tools to choose from made just for this purpose.

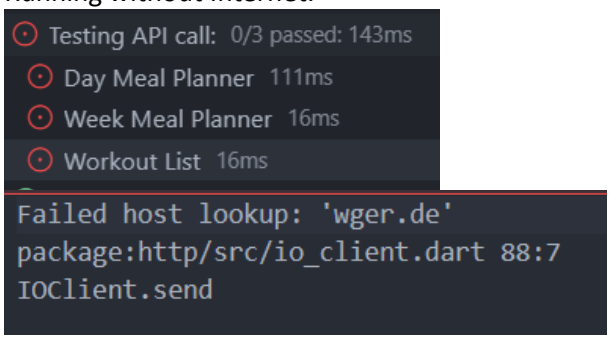
	You definitely can do these things with unit frameworks, but it might quickly become more work than it is worth. **Common Problems:** The most common problems we have with unit testing usually aren't technical problems. People have a hard time adapting to new ways of working after spending time in an environment where unit testing amounts to loading up the latest build and seeing whether it starts or not can be difficult. Test infected groups are a cultural phenomenon. I have had the most success in changing how testing is done by finding one person that is interested and invested in staying up to date. That person can be your champion to help build a case for the usefulness of unit testing and help to spread the idea through the dev organization through her successes. (http://dilbert.com/strip/2011-03-24) There is also a myth that says if testing is good, then more testing is better. Smart testing is good and will help to create a valuable, stable product. But smart testing doesn't always end in an impressive number of tests. Smart unit testing delivers relevant information about your software quickly and often. We've covered some of the basics of unit testing, and given you some ways to talk about them with your team. The next step is to start your own, preferably small, tests.

TEST CASES:	API Testing
	1) Day meal Planner API
	2) Week meal Planner API
	3) Workout API
	
	Run | Debug
	`test(' Day Meal Planner', () async {`
	`  //api call`
	`  final url = Uri.parse(MealPlanner.baseUrl + MealPlanner.generateMealPlan);`
	`  final response = await http.get(url, headers: MealPlanner.headers);`
	`  MealTemplateDay mealTemplate =`
	`  MealTemplateDay.fromJson(jsonDecode(response.body));`
	`  // logs.i(mealTemplate);`
	`  expect(`
	`  mealTemplate,`
	`  isA<MealTemplateDay>(),`
	`  reason: 'The response should be a MealTemplateDay object',`
	`  );`
	`});`
	
	Run | Debug
	`test(' Week Meal Planner', () async {`
	`  //api call`
	`  final url = Uri.parse(MealPlanner.baseUrl + MealPlanner.generateMealPlan);`
	`  final response = await http.get(url, headers: MealPlanner.headers);`
	`  MealTemplateWeek mealTemplate =`
	`  MealTemplateWeek.fromJson(jsonDecode(response.body));`
	`  // logs.i(mealTemplate);`
	`  expect(`
	`  mealTemplate,`
	`  isA<MealTemplateWeek>(),`
	`  reason: 'The response should be a MealTemplateWeek object',`
	`  );`
	`});`

Running test case:



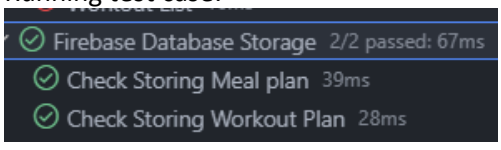
Running without Internet:



Firebase Storage Testing

```dart
Run | Debug
test('Check Storing Meal plan', () async {
  mockCollectionReference = fakeFirebaseFirestore.collection('meals');
  await mockCollectionReference.add(mealTemplate.toJson());
  QuerySnapshot<Object?> querySnapshot =
      await mockCollectionReference.get();
  //get the data from the collection
  List<QueryDocumentSnapshot<Object?>> documents = querySnapshot.docs;
  //get the first document
  QueryDocumentSnapshot<Object?> document = documents.first;
  //get the data from the document
  final Map<String, dynamic> data = document.data() as Map<String, dynamic>;
  //check if the data is the same as the one we added
  expect(data, mealTemplate.toJson());
});

Run | Debug
test('Check Storing Workout Plan', () async {
  mockCollectionReference = fakeFirebaseFirestore.collection('workouts');
  await mockCollectionReference.add(workoutTemplate.toJson());
  QuerySnapshot<Object?> querySnapshot =
      await mockCollectionReference.get();
  //get the data from the collection
  List<QueryDocumentSnapshot<Object?>> documents = querySnapshot.docs;
  //get the first document
  QueryDocumentSnapshot<Object?> document = documents.first;
  //get the data from the document
  final Map<String, dynamic> data = document.data() as Map<String, dynamic>;
  //check if the data is the same as the one we added
  expect(data, workoutTemplate.toJson());
});
```

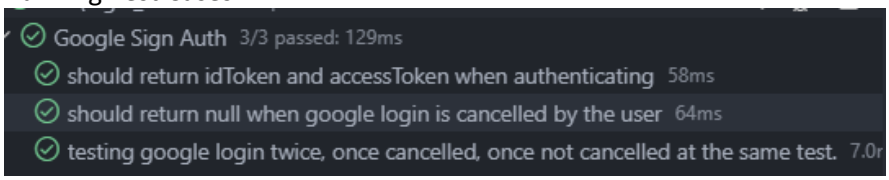
Running test case:



Google Sign In Testing

```
Run | Debug
test('should return idToken and accessToken when authenticating', () async {
  final signInAccount = await googleSignIn.signIn();
  final signInAuthentication = await signInAccount!.authentication;
  expect(signInAuthentication, isNotNull);
  expect(googleSignIn.currentUser, isNotNull);
  expect(signInAuthentication.accessToken, isNotNull);
  expect(signInAuthentication.idToken, isNotNull);
});

Run | Debug
test('should return null when google login is cancelled by the user',
    () async {
  googleSignIn.setIsCancelled(true);
  final signInAccount = await googleSignIn.signIn();
  expect(signInAccount, isNull);
});
Run | Debug
test(
    'testing google login twice, once cancelled, once not cancelled at the same test.',
    () async {
  googleSignIn.setIsCancelled(true);
  final signInAccount = await googleSignIn.signIn();
  expect(signInAccount, isNull);
  googleSignIn.setIsCancelled(false);
  final signInAccountSecondAttempt = await googleSignIn.signIn();
  expect(signInAccountSecondAttempt, isNotNull);
});
```

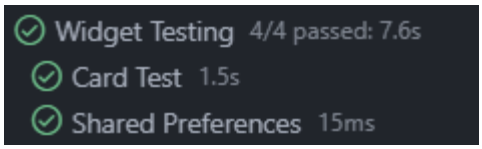
Running Test Cases



Widget and Local Storage Testing

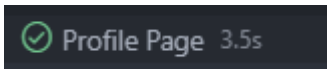
```
Run | Debug
testWidgets('Card Test', (WidgetTester tester) async {
  await tester.pumpWidget(MaterialApp(
    home: Scaffold(
      body: modernCard(
        const Text('Card'),
      ),
    ), // Scaffold
  )); // MaterialApp

  expect(find.text('Card'), findsOneWidget);
});

Run | Debug
test('Shared Preferences', () async {
  SharedPreferences.setMockInitialValues({});
  final prefs = await SharedPreferences.getInstance();
  prefs.setString('name', 'John');
  expect(prefs.getString('name'), 'John');
});
```

Running Test Case



Profile Page Testing

```
Run | Debug
testWidgets('Profile Page', (WidgetTester tester) async {
  await tester.pumpWidget(const MaterialApp(
    home: ProfileSetup(),
  )); // MaterialApp
  await sharedPrefs.clear();
  // enter name, height weight and age in text fields
  //get the text field that says name
  final nameField = find.byKey(const Key('name'));
  await tester.enterText(nameField, 'John');
  final heightField = find.byKey(const Key('height'));
  await tester.enterText(heightField, '170');
  final weightField = find.byKey(const Key('weight'));
  await tester.enterText(weightField, '60');
  final ageField = find.byKey(const Key('age'));
  await tester.enterText(ageField, '20');

  await tester.tap(find.byKey(const Key('save')));
  await tester.pump();

  // verify that shared prefs has been updated
  LocalUser user = await sharedPrefs.getUserDetails();
  // check if the name is John and the height is 170
  expect(user.name, 'John');
  expect(user.height, 170);
  expect(user.weight, 60);
  expect(user.age, 20);
});
```



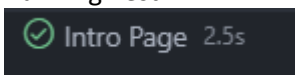
When given an incorrect input

```
final heightField = find.byKey(const Key('height'));
await tester.enterText(heightField, '0');
```

Intro Page Testing

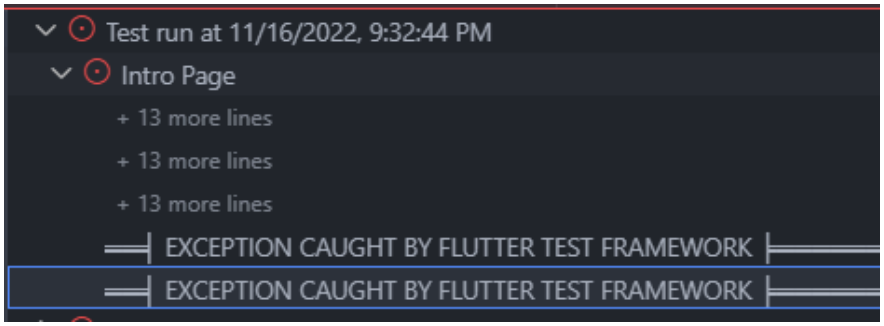
```
Run | Debug
testWidgets('Intro Page', (WidgetTester tester) async {
  await tester.pumpWidget(GetMaterialApp(
    routes: testRoutes,
    initialRoute: '/',
  ));
  await sharedPrefs.clear();
  //tap on the next button
  await tester.tap(find.byKey(const Key('next')));
  await tester.pump();
  await tester.tap(find.byKey(const Key('next')));
  await tester.pump();
  await tester.tap(find.byKey(const Key('next')));
  await tester.pump();
  //wait for one sec
  await tester.pump(const Duration(seconds: 1));
  await tester.tap(find.byKey(const Key('done')));
  await tester.pump();

  // verify that shared prefs has been updated
  bool firstEntry = await sharedPrefs.getBool('firstEntry');
  expect(firstEntry, true);
});
```

Running Test



Failed Test Case:

```
The following TestFailure was thrown
running a test:
Expected: <true>
  Actual: <false>
```



TEST CASE TABLE:

Sr No.	Test Description	Expected Result	Result Obtained	Status
1	Check if the Day meal Planner API Works	The API should result object corresponding to the result	Correct API Result	PASS
2.	Check if the Week meal Planner API works	The API should return list of meals that suffice for a week	API results having list of meals that have 7 days of meals	PASS
3.	Check if the meal plans data is stored in Cloud Firestore	The function should store the data under the collection 'meals'	Data is stored on the cloud firestore	PASS
4.	Check if the workout data is stored in Cloud Firestore	The function should take the workouts and convert them to store in cloud storage	Data isn't stored on the cloud firestore due to data being null	FAIL
5	Google Sign in	The function should sign in and return the access token and other detail	Access Token and other details are retrieve after the google sign in	PASS
6.	Google Sign In Cancelled	The function should cancel mid sign in	The result return is null	FAIL
7.	Profile Page	User enters the details like name, age, weight, height etc	The local storage confirms that the user details have been saved after clicking save	PASS
8.	Intro Page	After four page navigation, the local storage variable should change	User doesn't navigate after the third screen	FAIL
9.	Local Storage	The functions saves and checks if there exists in the local storage	The data is stored in local storage	PASS
10.	Testing the Widget	The function tests if the widget is displayed correctly	The widget displayed correctly	PASS

CONCLUSION:

From this experiment we were able to understand the importance of unit testing and how it affects the delivery of the project. We were also able to use in built **Flutter unit Testing** in order to perform unit testing tests, which made us aware of the different functionalities of the software.