Oppia

Write Frontend Tests

Google Summer of Code 2021

Radesh Kumar

About You

Why are you interested in working with Oppia, and on your chosen project?

My foremost reason for starting contributing to open source is GSoC. Back in 2020, I really wanted to participate in GSoC, and Indeed I've selected Oppia as an organization because I was pretty good at angular and Web Development. Then I realized that most of the Frontend had been written in angular js. And the architecture was in hybrid mode, which bogged me down.

Fast forward to the present. I've managed to analyze and break down the whole architecture into simple components. I came out of my comfort zone and picked up angular migration as a starter project. This gave me proper exposure to understand how both angular and angular js are initialized under the hood.

I've attended the "Oppia's 2021 Strategy Meeting", The way that Wala and Himanshu showcased how Oppia is impacting the lives of students all over the globe which inspired me and pushed my limits to contribute more and more. It has changed my entire perspective on open source and motivated me to be a better person.

Chosen Project

The reason for choosing the "Write Frontend Tests" project is, it involves two important things.
1.Angular Migration (**optional**)
2.Testing

According to me, the prominent thing right now is angular migration. We are not currently using the complete functionalities of angular. We lack vital features like Angular-CLI, reactive pages. We are also in a hybrid mode, which means we have to build both angular and angular js, increasing the compilation time.

My most favorite thing about Oppia is its Test-Driven-Environment. All these tests (FE, BE, E2E, Lint) will ensure that no new error or bug will be introduced into the system. Since Oppia is one of the most prominent open-source organizations, it needs rigorous testing and 100% coverage.

Being part of both the **Angular Migration core** team and the **Automated QA** team gives me an advantage over this project.

Community Bonding

Before even starting my contributions Pulkit Aggarwal, Srijan Reddy Vasa helped and made me understand how open source works and gave me a heads up about the Oppia code base structure.

My first PR was reviewed by Sandeep Dubey I really like the way he critically reviews each PR and maintains the codebase clean and well structured. He helped me in merging my initial PR's and to write a clean code.

After a couple of my PR's being merged, I was invited as a collaborator to the Oppia, which pleased me.
I've selected the Angular migration project. Savitha Jayasankar and Deepam Banerjee are the onboarding mentors. They helped and gave me directions to solve some starter issues.

I also wanted to explore other parts of the Oppia codebase to understand each page's workflow. The Automated QA team seemed the best fit for that. KEVIN THOMAS introduced me to A0RA CA1P (Chris). From then on, I've been a member of both teams and attended weekly meetings, and solved a few issues from both projects.

April 8th was a very special day for me, Because Sean sent me an invitation to become a member of the Oppia Team.

I would like to thank all members and maintainers of Oppia for helping me in each phase. Especially my mentors Nithesh Hariharan and Chris, without these guys, I wouldn’t be solving any of the issues.

Prior experience

1. Milkie Way, California
   Software Engineering Intern
   1. Project Announce:
      With the help of **Angular** and Python programming languages, we have created an effective Web App, Which consists of several functionalities to make announcements all over the globe based on the user's requests.
      It will showcase the announcements in the neighboring area. Experience letter

   2. Project Point: We have developed a custom address which is known as Point. With the help of a point, the user can share his/her address very easily. We also implemented a QR scan mechanism for scanning custom point codes on paper or postcards.

2. WissionTalks, Hyderabad
   Full Stack Developer
   1. Project Entrepreneur.online: This project mainly focuses on creating a medium for the communication of ideas, projects, obstacles faced by various Entrepreneurs. Experience letter

3. IEEE - Evolve X
   Full Stack Developer
   1. Project AgriWorld: We designed a WebApp which gives comprehensive information about Weather forecast, soil condition, new varieties of crops, plant protection and Nutrient deficiency.
   2. We have used **Angular** for the frontend, Node js for the backend and MongoDB as Database.
   3. Our Evolve-X project won third rank while competing against 800 people all over India. Certificate of Appreciation

My other work experiences and projects can be found from here and here.

My contributions to Oppia

Since I've been contributing to both the Angular **Migration core** team and the **Automated QA** team. I'm quite comfortable with the codebase and workflow in general.

1. My PR's
   - #12056 Migrated Learner's Dashboard Completely
     (This PR will fixes couple of **critical blocking** bugs and few UI bugs)
     (In this PR I've written unit tests for **12** files which consist of **495** line hits)
   - #11463, #11446 Refactored Backend Api Service for Admin Page Completely
     (And have written unit tests to fully cover all the files)
   - #11899 Migrated story-update.service (And have written unit tests to fully cover the same file)
   - #12110 Migrated promo-bar-directive (And have written unit tests to fully cover the same file)

   All other PR's of mine can be found here.
2. I've opened a few issues. Which consists of both critical blocking bugs and minor UI bugs.
   All the issues opened by me can be found here.
3. I've also participated in two testing releases.
4. I've been actively helping fellow contributors and solving some of the issues faced by them. And also reviewing PR's.
5. I'm currently maintaining Migrate directives/controllers to Angular components this issue. Which involves picking up beginner-friendly issues for contributors, actively updating the issue list, and assigning files.
   (We [Angular Migration Team] have a deadline that migration should be completed before the start of GSoC. This will boost up the process.)

Contact info and timezone(s)

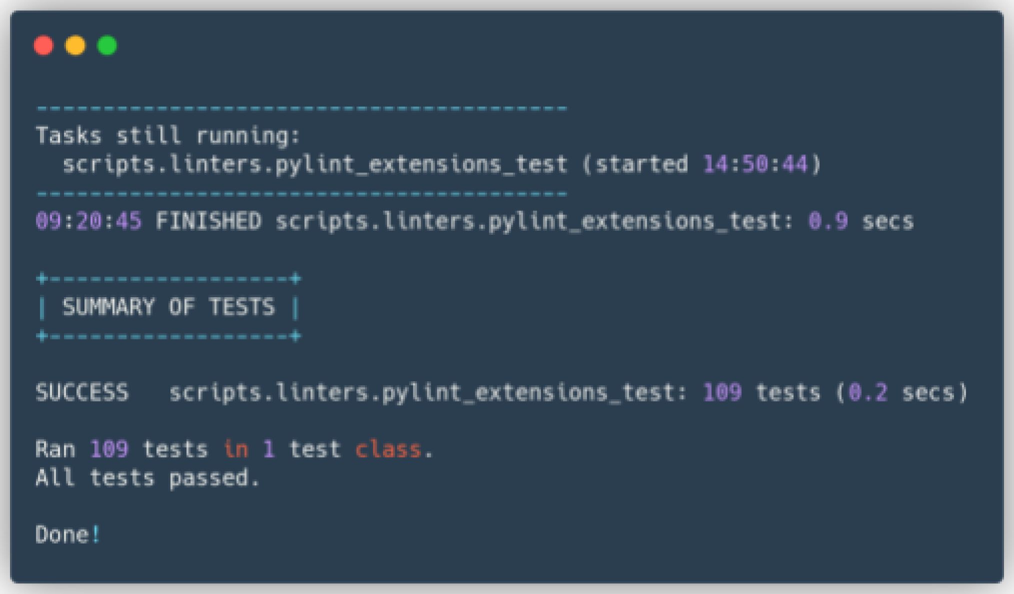
Email and Hangout: imradesh@gmail.com
Phone: +91 8639679152
Discord: Radesh#6840
Timezone: Indian Standard Time (+5:30 GMT)
Preferred Method of communication: Hangouts, WhatsApp, Discord.

Time commitment

- I'll be working up to 25 hours/ week, which sums up 250 hours for 10 weeks.
- Weekdays: 4-5 hours/day
- Weekends: 2-3 hours/day. If any unexpected issues arise or if mentors request changes. (Weekends would be best for reviewing for the mentors too.)

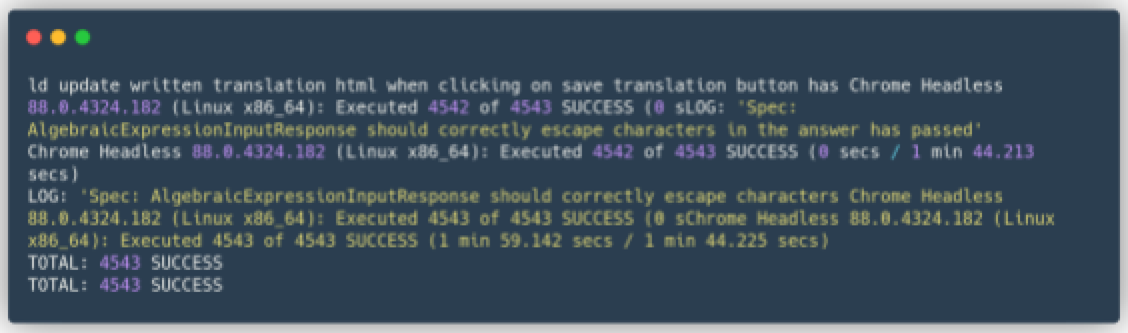
Essential Prerequisites

- I am able to run a single backend test target on my machine.



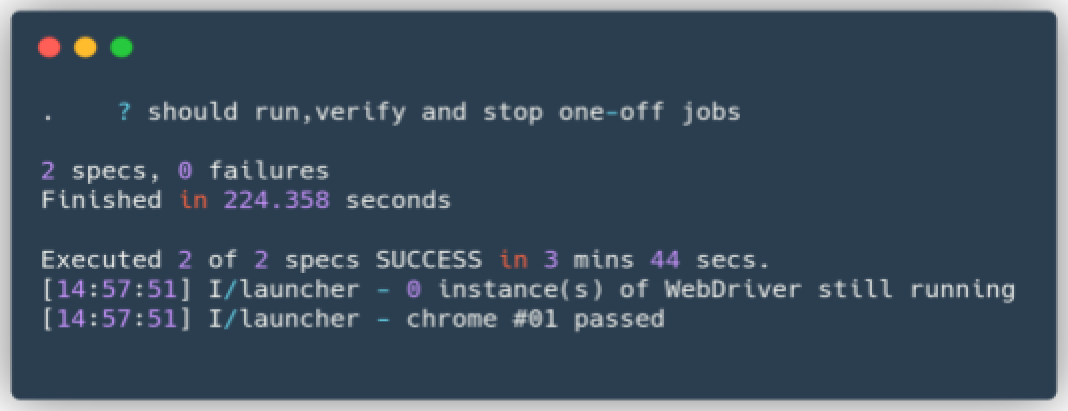
```
----------------------------------------
Tasks still running:
  scripts.linters.pylint_extensions_test (started 14:50:44)
----------------------------------------
09:20:45 FINISHED scripts.linters.pylint_extensions_test: 0.9 secs

+------------------+
| SUMMARY OF TESTS |
+------------------+

SUCCESS   scripts.linters.pylint_extensions_test: 109 tests (0.2 secs)

Ran 109 tests in 1 test class.
All tests passed.

Done!
```

- I am able to run all the frontend tests at once on my machine.



```
ld update written translation html when clicking on save translation button has Chrome Headless
88.0.4324.182 (Linux x86_64): Executed 4542 of 4543 SUCCESS (0 sLOG: 'Spec:
AlgebraicExpressionInputResponse should correctly escape characters in the answer has passed'
Chrome Headless 88.0.4324.182 (Linux x86_64): Executed 4542 of 4543 SUCCESS (0 secs / 1 min 44.213
secs)
LOG: 'Spec: AlgebraicExpressionInputResponse should correctly escape characters Chrome Headless
88.0.4324.182 (Linux x86_64): Executed 4543 of 4543 SUCCESS (0 sChrome Headless 88.0.4324.182 (Linux
x86_64): Executed 4543 of 4543 SUCCESS (1 min 59.142 secs / 1 min 44.225 secs)
TOTAL: 4543 SUCCESS
TOTAL: 4543 SUCCESS
```

- I am able to run one suite of e2e tests on my machine.



```
.    ? should run,verify and stop one-off jobs

2 specs, 0 failures
Finished in 224.358 seconds

Executed 2 of 2 specs SUCCESS in 3 mins 44 secs.
[14:57:51] I/launcher - 0 instance(s) of WebDriver still running
[14:57:51] I/launcher - chrome #01 passed
```

Other summer obligations

- **Time Constraint**: The only time constraint I will be having is my semester exams during the first two weeks of July. We have 3 semesters this year. Typically we will be having only 2/year. Due to Covid19, one of the semester exams was postponed.
- To be on the safer side, If mentors allow me to contribute 2 weeks prior to the starting of the official GSoC (i.e., during community bonding), It would be beneficial. Otherwise, I'm willing to start from the official starting date of GSoC itself.

Communication channels

- I will be communicating with my mentor through Hangouts.
- In addition to that I'll be maintaining a spreadsheet for updating my progress throughout the program.

Application to multiple orgs

- I am applying only to Oppia.

Project Details

Product Design

What are Unit tests?

Unit testing is a process that involves testing the smallest piece of code by logically isolating the system. The codebase is broken down into individual modules. Modules are further broken down into functions, methods, attributes, and properties. These are commonly known as units.

The main purpose behind this is to check that all the individual parts are working as intended. The entire system will only be able to work well if the individual units are working well.
We can broadly divide unit tests into two types.

1.**State Based Unit Testing**:
The process of verifying the system whether the unit test produces correct results or that its resulting state is correct.

2.**Interaction Based Unit Testing**:
The process of verifying the system whether it is properly invoking certain methods or functions accurately.

Why Unit tests?

Evading New Bugs: Since Oppia's codebase is considerably huge and consists of a large community, many people tend to make code changes. In case of any new bugs due to the latest code changes are prevented from being entered into the system.

Documentation: Unit tests serve as code documentation. Since we are checking each path's functionality, It would be easier for a developer to understand how actual code works by looking at the test file.

Properties of Unit tests

Modularity: It is a testing technique that emphasizes separating the program's functionality into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.

Readability: The intent of a unit test should be clear and concise. A good unit test tells about the behavioral aspect of our application, so it should be easy to understand.

Writability: We generally write lots of unit tests to cover different cases and aspects of the system’s behavior, so it should be easy to code all of those test routines without immense effort. Since we are repeating the same procedure.

Reliability: Unit tests should fail only if there’s a bug in the system under test. But programmers often run into an issue when their tests fail, even when no bugs were introduced. These are known as flakes.

These situations are indicative of a design flaw. Good unit tests should be reproducible and independent from external factors such as the environment or running order.
I’ve found a couple of Frontend test flakes in the code base and opened issues for the same.
#12316, #12142

Isolation: Unit Test cases should be completely independent. In case of any enhancements or changes in dependencies, unit test cases should not be affected.
We should not access the network resources, databases, file system, etc., to eliminate the influence of external factors.
We should not access components and services directly. Instead, we should use mocks and spies.

Limitations of Unit Tests

- Unit tests can't be expected to catch every error in a program. It is not possible to evaluate all execution paths even in the most trivial programs.
- Unit tests by its very nature focuses on a unit of code. Hence it can't catch integration errors or broad system level errors.
- Unit tests can't test the workflow of components in the UI, as we could do in E2E tests.

How To Unit Test?

Unit tests can be confined into three main steps (AAA).

1.Arrange:

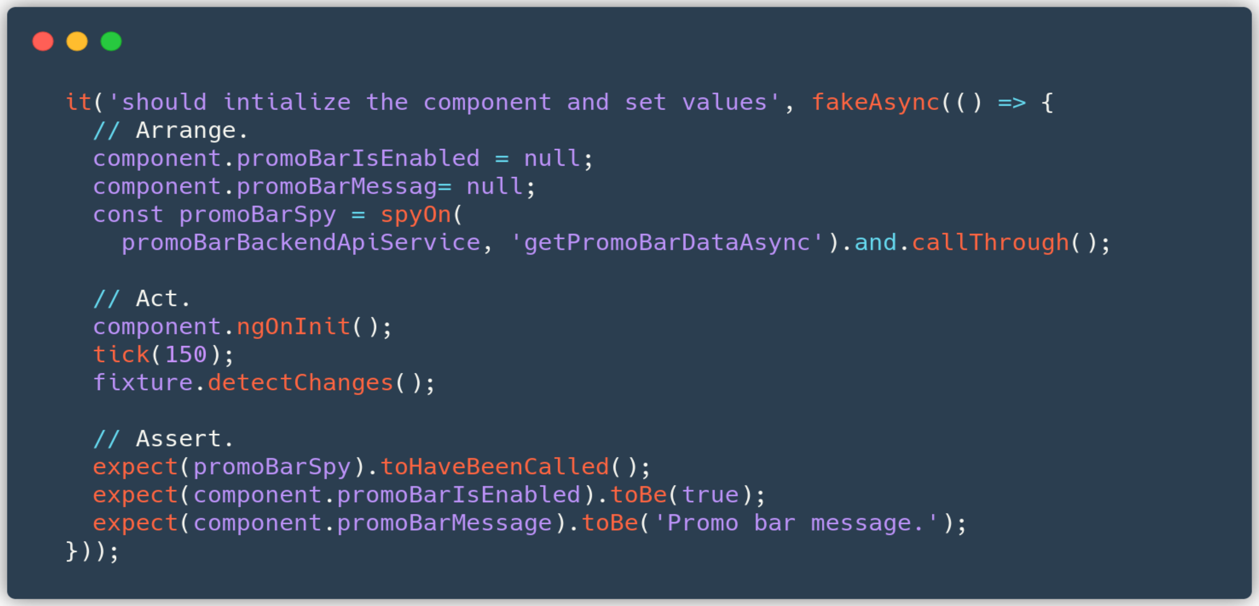
In this phase, we will be initializing values to the variables, creating mocks/stub components, and creating spies for services.

2.Act:

In this phase, we will be triggering functions, promises, observables, and life cycle hooks.

3.Assert :

In this phase, we will be asserting the states of objects or values of variables and check whether the functions are called correctly.



```
it('should intialize the component and set values', fakeAsync(() => {
  // Arrange.
  component.promoBarIsEnabled = null;
  component.promoBarMessag= null;
  const promoBarSpy = spyOn(
    promoBarBackendApiService, 'getPromoBarDataAsync').and.callThrough();

  // Act.
  component.ngOnInit();
  tick(150);
  fixture.detectChanges();

  // Assert.
  expect(promoBarSpy).toHaveBeenCalled();
  expect(component.promoBarIsEnabled).toBe(true);
  expect(component.promoBarMessage).toBe('Promo bar message.');
}));
```

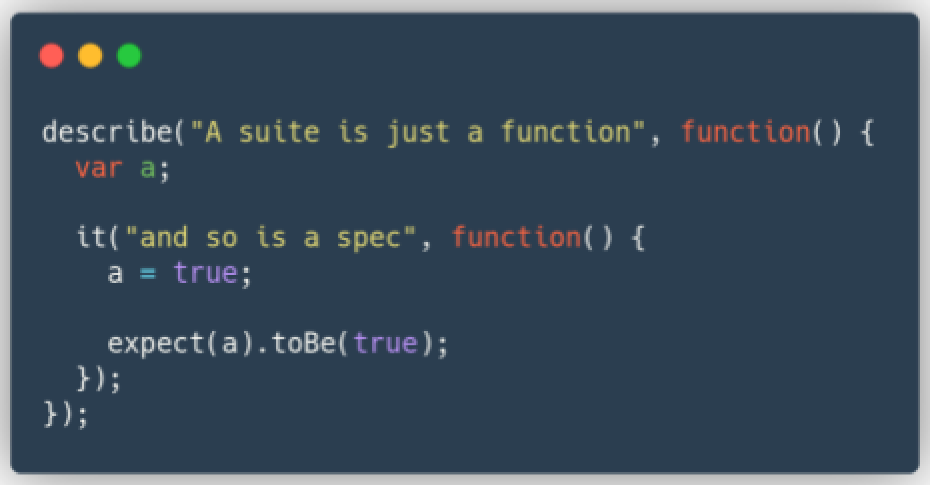
Components of Testing (Jasmine)

Test Suite :

- A Test Suite is a collection of test cases. In automated testing, it can mean a collection of test scripts. In a test suite, the test cases/scripts are organized in a logical order.
- For example, the test case for registration will precede the test case for login.
- In jasmine, test suites are mentioned as **describe.**

Test Cases:

- Test cases are used to test every possible path/result of a specific function.
- In jasmine they are represented as **it.**



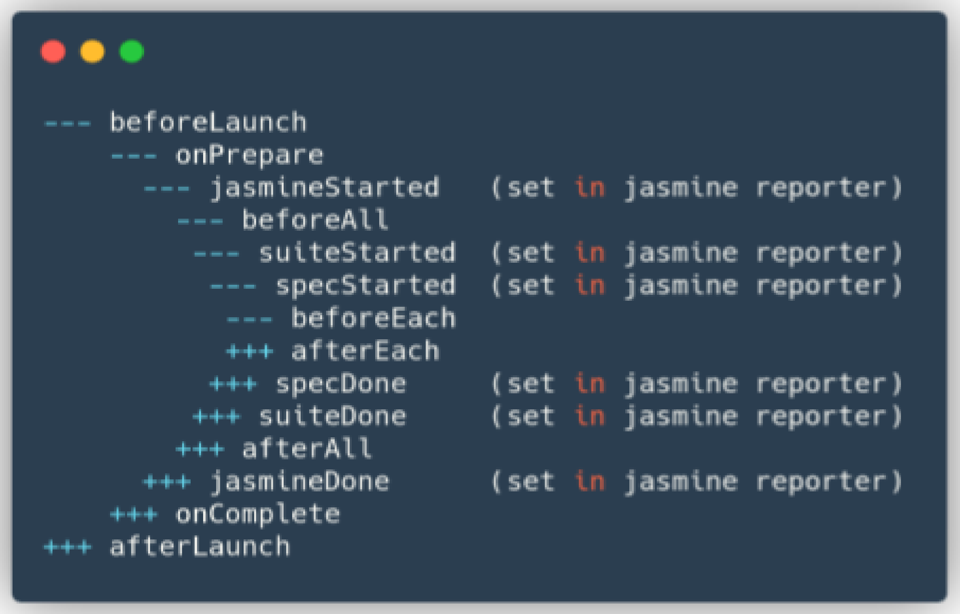
```
describe("A suite is just a function", function() {
  var a;

  it("and so is a spec", function() {
    a = true;

    expect(a).toBe(true);
  });
});
```

Fig - Sample Test Suite

●

Hooks :

- With the help of jasmine hooks, we can trigger any function / or change the variables.
- These are mainly used in the **Setup** and **Teardown** process.
- Although there are many hooks, we mainly use beforeEach and afterEach, these are triggered before each test case and after each case respectively.



```
--- beforeLaunch
    --- onPrepare
      --- jasmineStarted   (set in jasmine reporter)
        --- beforeAll
         --- suiteStarted  (set in jasmine reporter)
          --- specStarted  (set in jasmine reporter)
           --- beforeEach
           +++ afterEach
          +++ specDone     (set in jasmine reporter)
         +++ suiteDone     (set in jasmine reporter)
        +++ afterAll
      +++ jasmineDone      (set in jasmine reporter)
    +++ onComplete
+++ afterLaunch
```

Fig - Jasmine Hooks in Chronological order

Expectation :

- These are used in the assertion phase of a test case.
- We have different matcher functions, each for a unique purpose.
- Some of them are as follows.

MATCHER	PURPOSE
toBe()	Passed if the actual value is of the same type and value as that of the expected value. It compares with === operator
toEqual()	Works for simple literals and variables.Should work for objects too
toMatch()	To check whether a value matches a string or a regular expression
toBeDefined()	To ensure that a property or a value is defined
toBeUndefined()	To ensure that a property or a value is undefined
toBeNull()	To ensure that a property or a value is null.
toBeTruthy()	To ensure that a property or a value is `true`
ToBeFalsy()	To ensure that a property or a value is `false`
toContain()	To check whether a string or array contains a substring or an item.
toBeLessThan()	For mathematical comparisons of less than

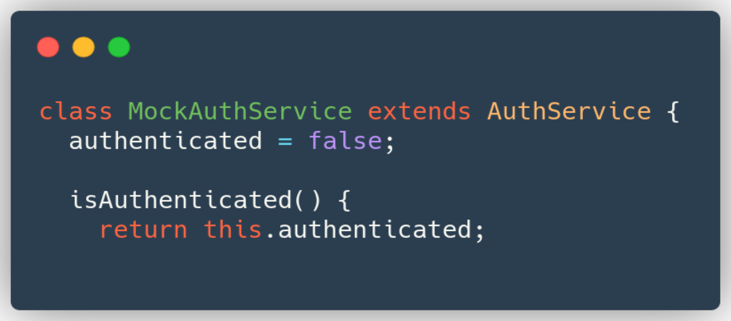
toBeGreaterThan()	For mathematical comparisons of greater than
toBeCloseTo()	For precision math comparison
toThrow()	For testing if a function throws an exception
toThrowError()	For testing a *specific* thrown exception

Mocks vs Spies:

- One of the primary aims of unit testing is to isolate a method or component that you want to test.
- We should **never** call the original functions of service/components, Since they have dependencies over other services/components which will sabotage the main principle of unit testing (isolating).
- Here, Mocks and spies come into play.
- Mocks and Spies are very much interchangeable.
- It really depends on the use case/ application for choosing one specific method over another method.
- We can also use a mixture of both methods in our tests.

Mocks:

- Mocks work by implementing the proxy pattern.
- When you create a mock object, it creates a proxy object that takes the place of the real object.
- We can then define what methods are called and their returned values from within our test method.



```
class MockAuthService extends AuthService {
  authenticated = false;

  isAuthenticated() {
    return this.authenticated;
```

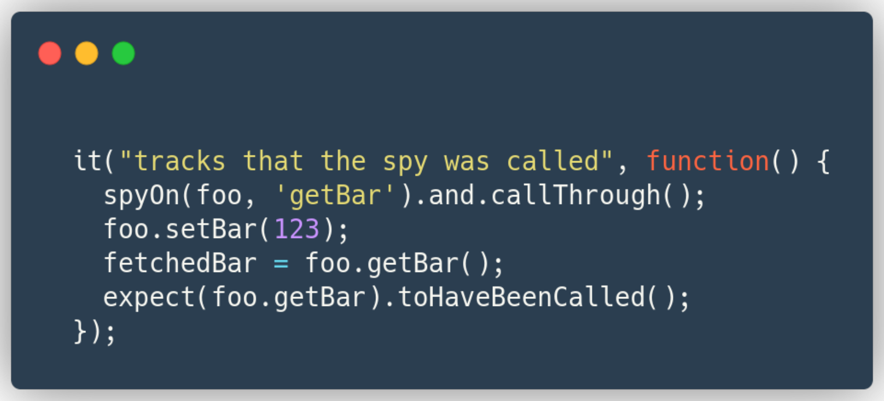
Spies:

- A spy can stub any function and tracks calls to it and all arguments.
- A spy only exists in the describe or it block in which it is defined, and will be removed after each spec.
- We can spy either or function of objects methods and properties.

SpyOn:

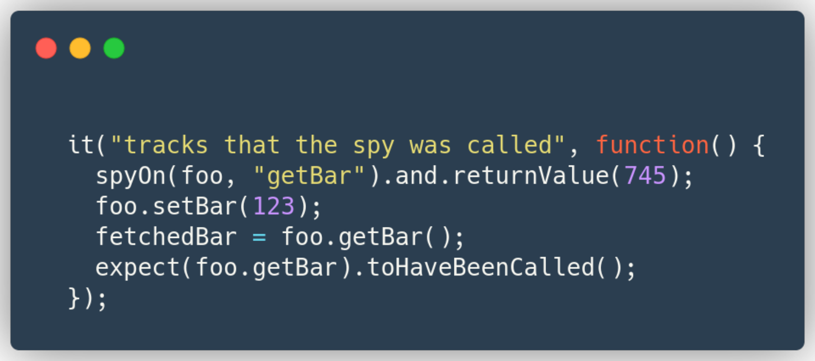
- With the help of spyOn we can easily spy on any function.
- There are mainly three types.

1. **spyOn.and.callthrough():**
   By chaining the spy with and.callThrough, the spy will still track all calls to it but in addition it will delegate to the actual implementation.



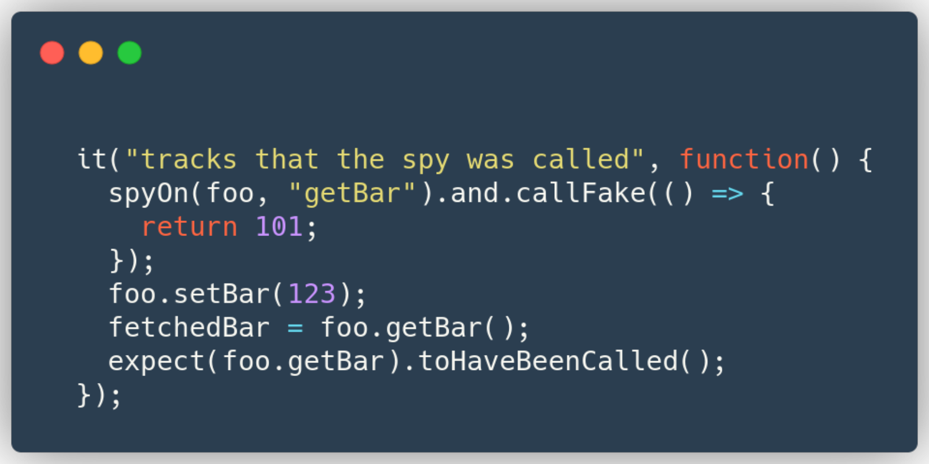
```
it("tracks that the spy was called", function() {
  spyOn(foo, 'getBar').and.callThrough();
  foo.setBar(123);
  fetchedBar = foo.getBar();
  expect(foo.getBar).toHaveBeenCalled();
});
```

2. **spyOn.and.returnValue():**
   By chaining the spy with and.returnValue, all calls to the function will return a specific value.



```
it("tracks that the spy was called", function() {
  spyOn(foo, "getBar").and.returnValue(745);
  foo.setBar(123);
  fetchedBar = foo.getBar();
  expect(foo.getBar).toHaveBeenCalled();
});
```

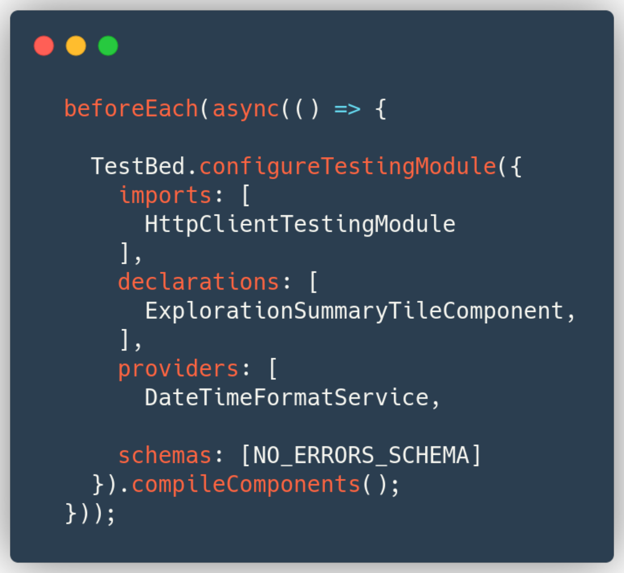
3. **spyOn.and.callFake():**
By chaining the spy with and.callFake, all calls to the spy will delegate to the supplied anonymous function.



```
it("tracks that the spy was called", function() {
  spyOn(foo, "getBar").and.callFake(() => {
    return 101;
  });
  foo.setBar(123);
  fetchedBar = foo.getBar();
  expect(foo.getBar).toHaveBeenCalled();
});
```

Angular Test Bed:

- The Angular Test Bed (ATB) is a higher level Angular Only testing framework that allows us to quickly test behaviours that depend on the Angular Framework.
- We still write our tests in Jasmine and run using Karma, but we now have a slightly better way to create components, handle dependency injection, test asynchronous behaviour and interact with our application.
- Angular TestBed mocks the behaviour of ngModules and provides us all the attributes related to ngModules.
- Example: imports, declarations, providers etc..



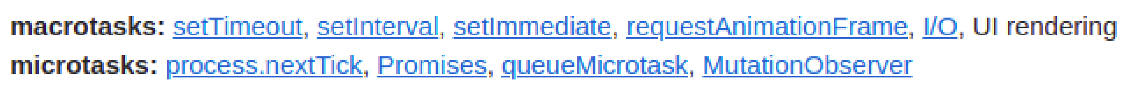
```
beforeEach(async(() => {

  TestBed.configureTestingModule({
    imports: [
      HttpClientTestingModule
    ],
    declarations: [
      ExplorationSummaryTileComponent,
    ],
    providers: [
      DateTimeFormatService,

    schemas: [NO_ERRORS_SCHEMA]
  }).compileComponents();
}));
```

Testing Services:

- Firstly, we need to import the service file which will be tested and also all depending files (involves services, models, and constants).
- Then we should set up and initialize the testbed.
- Inject all required dependencies into the testbed (services).
- Initialize Setup and Teardown with the help of beforeEach and afterEach if required.
- Then we should write test cases for all possible paths.
- We should never call original methods or functions in the test file. Instead, we should use mocks and spies. For example, if a service is dependent on another service, in order to isolate unit tests we should never call the original dependent service in our spec file (We can either mock/spy on them).

Testing Asynchronous functions :

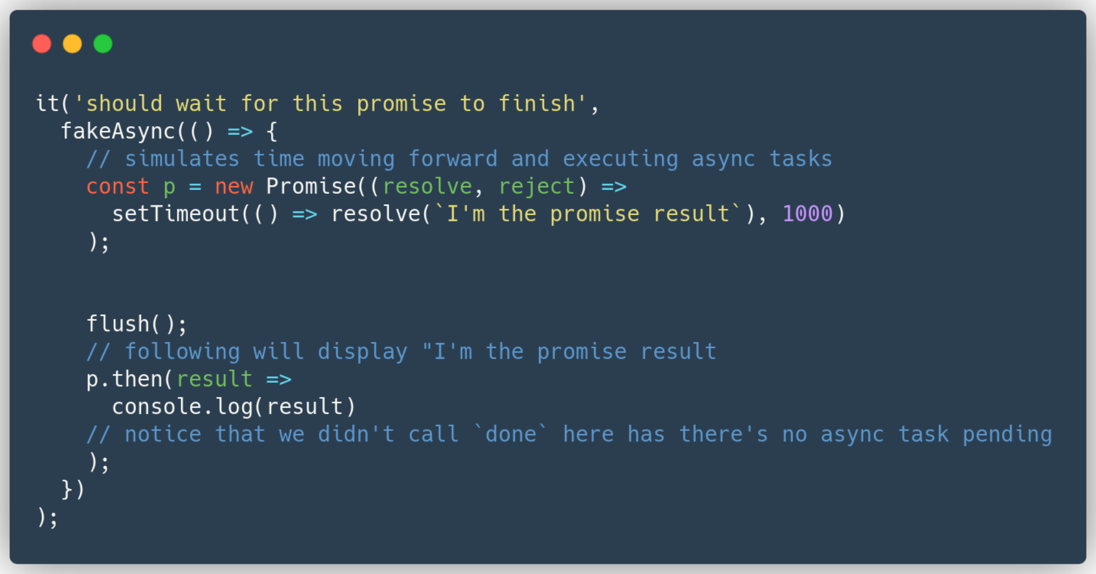
- Since most of the services consist of HTTP requests and promises, we must be aware of how to test in such scenarios.
- We can broadly divide asynchronous tasks into two categories.
    1. **Micro Task**
    2. **Macro Task**



macrotasks: setTimeout, setInterval, setImmediate, requestAnimationFrame, I/O, UI rendering
microtasks: process.nextTick, Promises, queueMicrotask, MutationObserver

- There is no big difference between micro and macro tasks. Both are implemented using a queue.
- But, Micro tasks have relatively higher priority than macro tasks.
- For testing asynchronous code, we mainly use 3 functions
    1. **flushMicrotasks ( )**
    2. **tick ( )**
    3. **flush ( )**
- flushMicrotask( ) will make sure that all remaining micro tasks in the queue are resolved,
- tick( ) and flush( ) will wait to make sure that all remaining micro tasks and macro tasks in the queue are resolved.
- This will force the test spec to run in a synchronous block.
- Such that we can assert the expectations after all micro tasks and macro tasks are resolved

.



```
it('should wait for this promise to finish',
  fakeAsync(() => {
    // simulates time moving forward and executing async tasks
    const p = new Promise((resolve, reject) =>
      setTimeout(() => resolve(`I'm the promise result`), 1000)
    );


    flush();
    // following will display "I'm the promise result
    p.then(result =>
      console.log(result)
    // notice that we didn't call `done` here has there's no async task pending
    );
  })
);
```

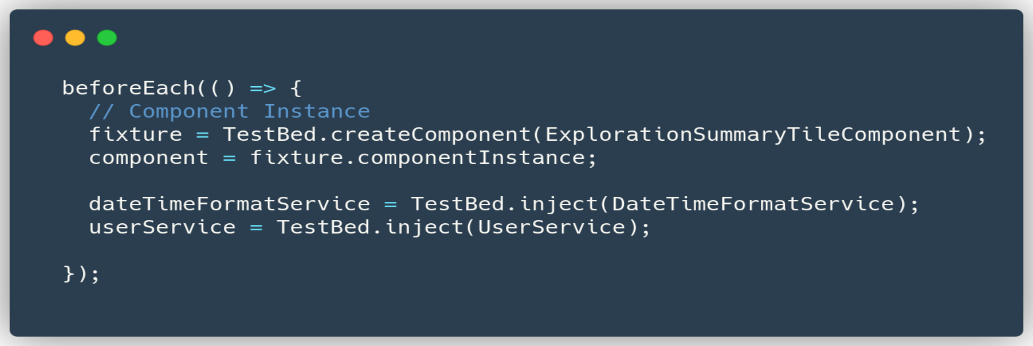
Fig - Testing Asynchronous Functions

Testing Components:

- Firstly, we need to import the service file which will be tested and also all depending files (involves services, models, constants, pipes, and other angular components).
- Then we should set up and initialize the testbed.
- Inject all required dependencies into the testbed (Service, pipes, component stubs).
- Initialize Setup and Teardown with the help of beforeEach and afterEach if required.
- Then we should write test cases for all possible paths.
- We should never call original methods or functions in the test file. Instead, we should use mocks and spies.

Testing Component States :

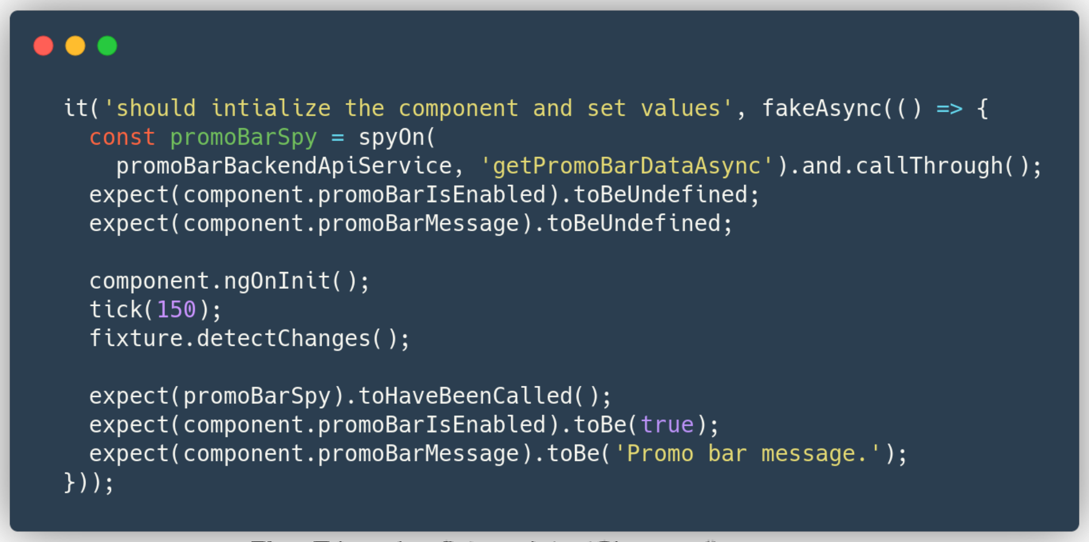
- Testing Component states is a bit trickier because zone.js constantly detects changes and updates the component in angular applications.
- We can't use a traditional approach to test in these situations.
- We can approach this issue in 3 different ways.
    1. **fixture.detectChanges( )**
    2. **fixture.whenStable( )**
    3. **ComponentFixtureAutoDetect**
- In order to use the fixture, we must create the component using a testbed.



```
beforeEach(() => {
  // Component Instance
  fixture = TestBed.createComponent(ExplorationSummaryTileComponent);
  component = fixture.componentInstance;

  dateTimeFormatService = TestBed.inject(DateTimeFormatService);
  userService = TestBed.inject(UserService);

});
```

1.fixture.detectChanges( )

By using **fixture.detectChanges( )** we can manually trigger the change detection in the testbed itself. (This is similar to $scope.$apply() we have in AngularJS)



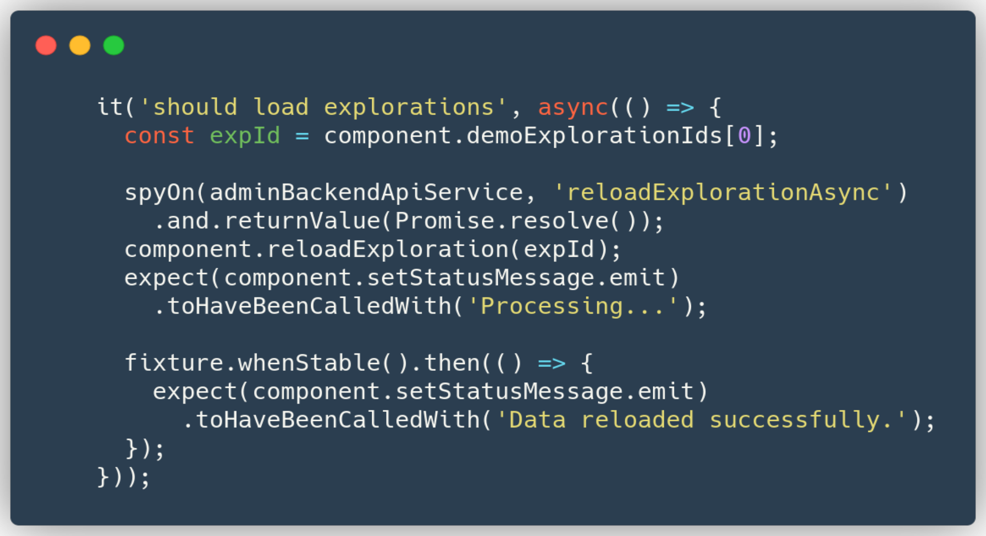
```
it('should intialize the component and set values', fakeAsync(() => {
  const promoBarSpy = spyOn(
    promoBarBackendApiService, 'getPromoBarDataAsync').and.callThrough();
  expect(component.promoBarIsEnabled).toBeUndefined;
  expect(component.promoBarMessage).toBeUndefined;

  component.ngOnInit();
  tick(150);
  fixture.detectChanges();

  expect(promoBarSpy).toHaveBeenCalled();
  expect(component.promoBarIsEnabled).toBe(true);
  expect(component.promoBarMessage).toBe('Promo bar message.');
}));
```

Fig - Triggering fixture.detectChanges()

- This method is used mostly in synchronous functions.

2.fixture.whenStable( )

- In order to test change detection in asynchronous operations, we should use **fixture.whenStable( ).**
- **Note:** whenStable() does nothing if you test it without async or fakeAsync.
- What whenStable() does is to wait for all tasks in the test NgZone to complete.When you don't test with async the NgZone does not get created at all and whenStable() just returns immediately.

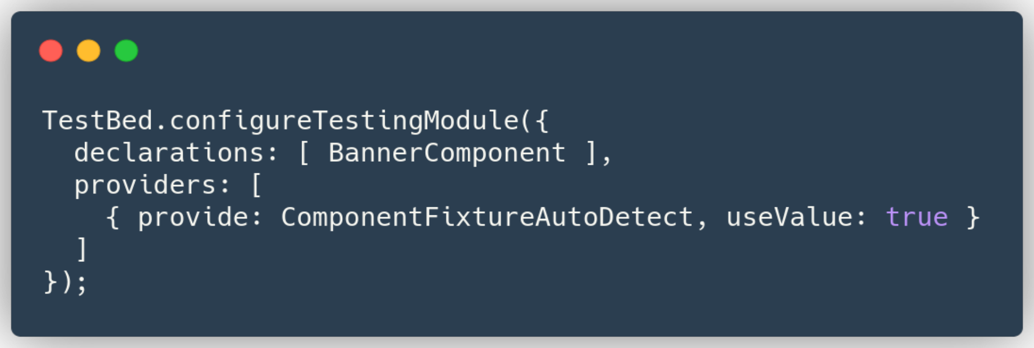


```
it('should load explorations', async(() => {
  const expId = component.demoExplorationIds[0];

  spyOn(adminBackendApiService, 'reloadExplorationAsync')
    .and.returnValue(Promise.resolve());
  component.reloadExploration(expId);
  expect(component.setStatusMessage.emit)
    .toHaveBeenCalledWith('Processing...');

  fixture.whenStable().then(() => {
    expect(component.setStatusMessage.emit)
      .toHaveBeenCalledWith('Data reloaded successfully.');
  });
}));
```

Fig - Triggering fixture.whenStable( )

3.ComponentFixtureAutoDetect

- We can detect changes automatically by configuring the TestBed with the ComponentFixtureAutoDetect provider.



```
TestBed.configureTestingModule({
  declarations: [ BannerComponent ],
  providers: [
    { provide: ComponentFixtureAutoDetect, useValue: true }
  ]
});
```

Fig - initializing ComonentFixtureAutoDetect

Aim Of the Project

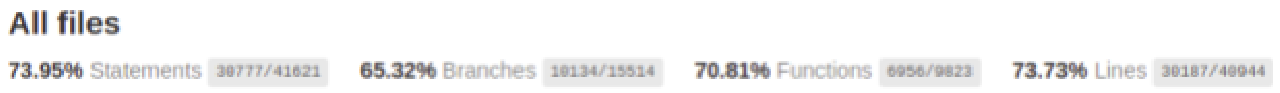
Oppia’s frontend codebase can be broadly classified into four main categories.

1. Components
2. DIrectives
3. Services
4. Others

Students (3members) are expected to fully cover components, directives, and services. **Excluding others** category as per Idea List

Line counts as of **April 8, 2021**:

	Total Lines	Covered Lines	Uncovered Lines
Components	8977	8058	919
Directives	10245	2591	7654
Service	11917	10540	1377
Others	9805	8998	807
Total	40944	30187	10757

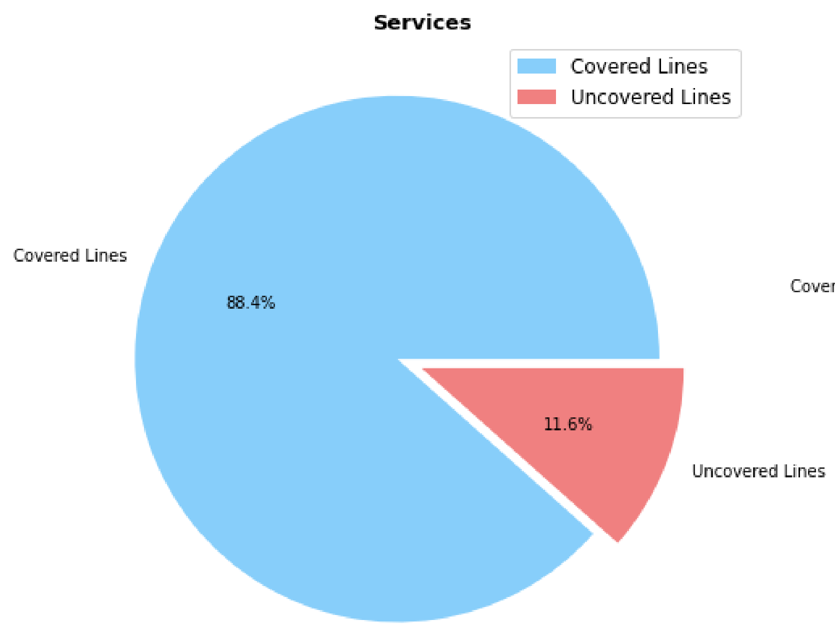


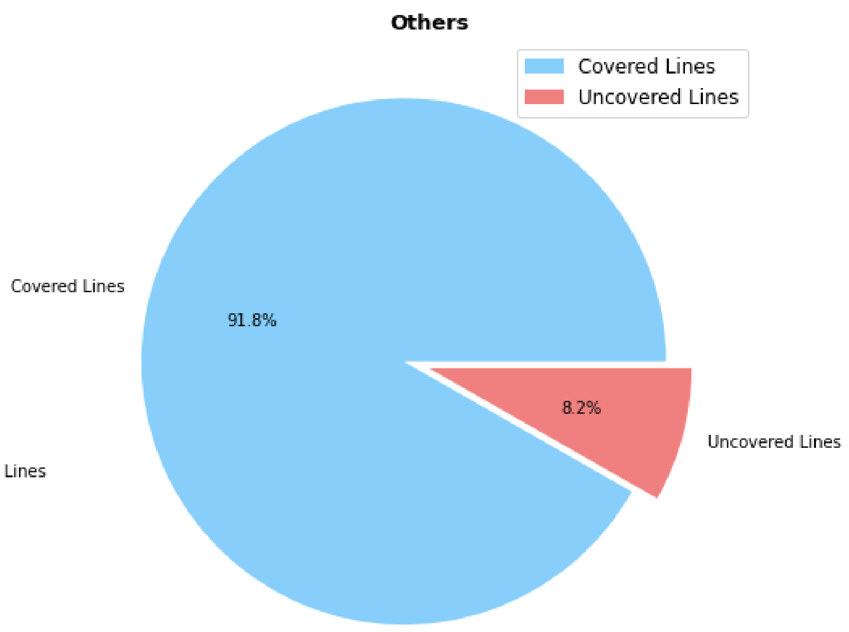
All files

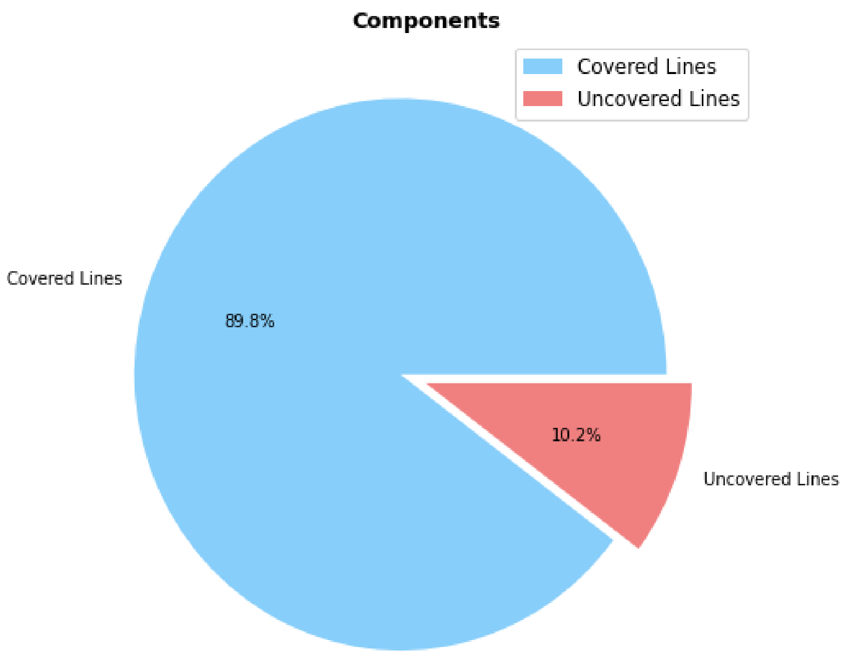
73.95% Statements 30777/41621 **65.32%** Branches 10134/15514 **70.81%** Functions 6956/9823 **73.73%** Lines 30187/40944

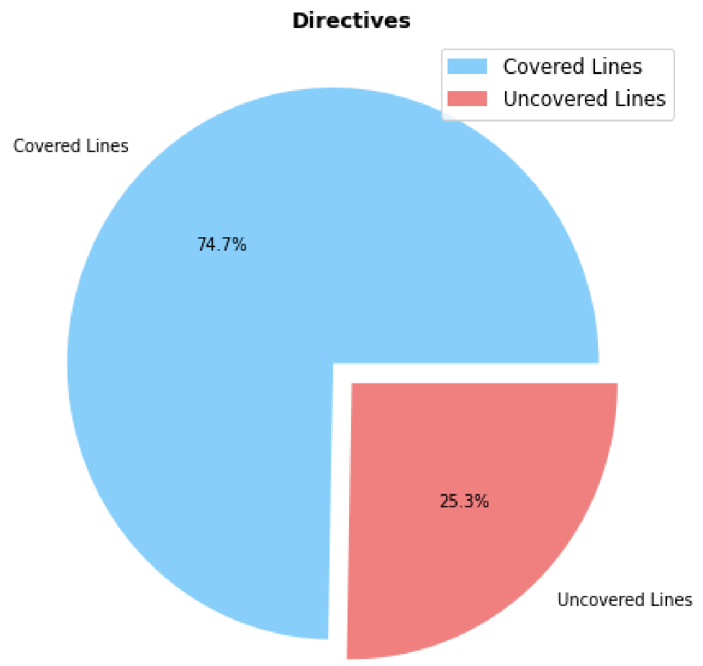
Note:

- **Lines** mentioned in the table refer to line hits which are generated by **Istanbul** (Karma coverage Reporter) **Not lines of code**.
- All line hits are calculated using the lcov.info file produced by **istanbul**.









Expected Goal

- By excluding others category (807), we should thoroughly test and should achieve 100% coverage in components, directives and service
- From **9950** Lines, 3 students are equally distributed. Which means **3316** lines for each student.
- Students are expected to migrate AngularJS directive to AngularJS components.

My Goal

- My primary focus for this project is only unit testing. Although I'll add angular migration in the appendix section in the proposal as an optional entity.
- It is very unclear for mentors to sort and assign files for migration/unit tests, As they are still work in progress (WIP) and changing constantly.(Files will be sorted in the Community Bonding Period.)

Best Case Scenario

- Migration will be fully completed prior to the starting date of GSoC.
- I will only be writing only unit tests.

Worst Case Scenario

- Migration would be partially done (At most 60% of directives).
- I will be migrating the remaining directives along with unit tests to fully cover them.

Technical Design

Architectural Overview

Most of the frontend code is available in either core/templates or extensions
core/templates: frontend code

- base-components: Core components, which can be initialized only once.
- components: Consists of Reusable components.
- css: Consists of all styling options(includes custom styles and Material css).
- directives: Directives that are not associated with Reusable components.
- domain: Consists of models and object factories (acts as a logical layer for frontend).
- pages: Consists of all the pages for the web app.
- services: Consists of shared services for the web app.

extensions: extensions where can be implemented by the developers

- interactions: Consists of components that allow the learner to submit an answer.
- objects: Objects that Oppia recognizes.
- rich_text_components: Consists of components for providing custom widgets.
- visualizations: Pictorial representations of functionalities.

File counts as of **April 8, 2021**:

	Total Files	Fully Covered	Partially Covered	Uncovered
Components	205	163	42	0
Directives	203	15	187	1
Service	333	262	71	0
Others	388	348	35	5
Total	1129	788	335	6

List of Services

We have a total of 333 services. In which, 262 services have 100% coverage. Partially covered services are 71. And finally uncovered files are 0.

Partially covered services:

- Services which have uncovered lines > 1.
- Link

Fully covered services:

- Services which have uncovered lines = 0.
- Link

List of Directives

We have a total of 203 directives. In which, 15 directives have 100% coverage. Partially covered directives are 188. And finally uncovered files are 1.

Partially covered Directives:

- Directives which have uncovered lines > 1.
- Link

Fully covered Directives:

- Directives which have uncovered lines = 0.
- Link

Uncovered Directives:

- Directives which have covered lines = 0.
- Link

List of Components

We have a total of 205 components. In which, 163 components have 100% coverage. Partially covered components are 42. And finally uncovered files are 0.

Partially covered Components:

- Componentes which have uncovered lines > 1.
- Link

Fully covered Components:

- Componentes which have uncovered lines = 0.
- Link

Note:

- In the previous section, I've used a spreadsheet which consists of all the frontend files in the Oppia.
- It is a comprehensive list with 4 sections (components, directives, services, and others).
- Each section has 3 subsections (partially covered, fully covered, uncovered).
- It is sorted with respect to uncovered lines in descending order.
- I've written scripts to sort these files which are derived from the **lcov.info** file which can be found here.
- Simply copy-pasting all files here (in the proposal) is just redundant and won't do any good.
- Instead, I've created a spreadsheet that will help mentors to **track** student's (3 people) progress simultaneously.
- Each student can update the sheet after completing the assigned tasks.
- Otherwise, mentors need to check in with each student. It's just time consuming and not efficient.

Implementation Approach

- Considering the fact that, for this project, files are distributed among the students (3 people) **only after getting selected**, I'm evaluating files very broadly.
- We have a total of 9950 lines from 301 files.
- So each student will get around 3300 lines and 100 files to test.
- Considering we have a time period of 10 weeks, I'm distributing 330 lines / 10 files for a week.

	Easy	**Medium**	**Hard**
Complexity	Upto 100 lines	Upto 250 lines	At Least 250 lines
Time	1-4 files/ day	1-2 files/ 2 days	1 file/2-3 days

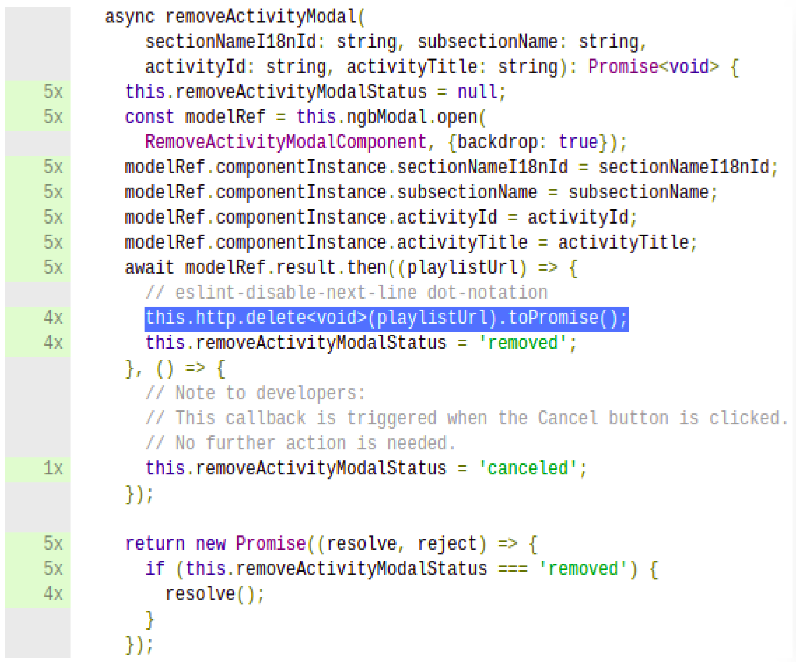
` **Note:** In the above list files are considered to be tested and migrated **(optional)**.

Testing Approach

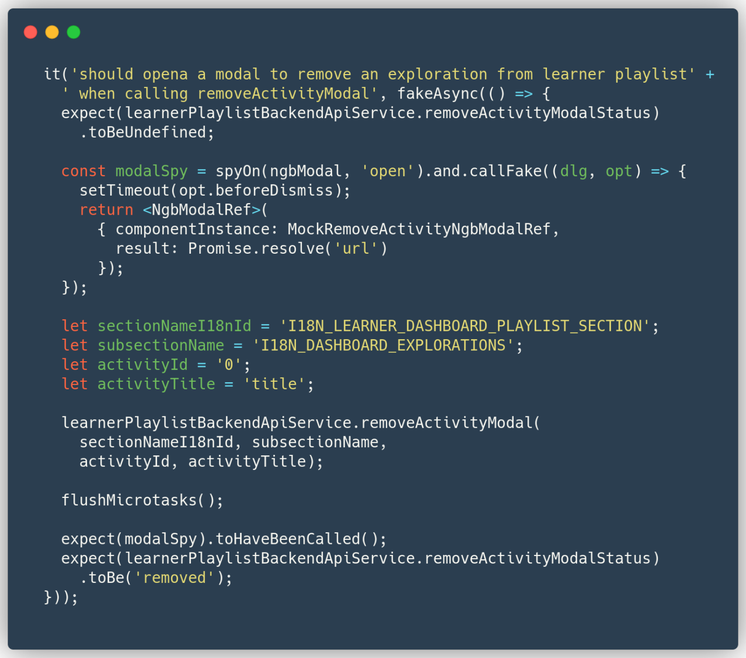
Some Tricky Scenarios

1.Hitting Lines vs Actually Testing Lines

- The end goal of the unit testing is **not** just hitting each line (100% coverage). Instead, we should test each line and each path which helps us to make our software more robust and less error prone
- Although we didn't test a specific line in a test sometimes, it will be covered in the Karma coverage report.



```
      async removeActivityModal(
          sectionNameI18nId: string, subsectionName: string,
          activityId: string, activityTitle: string): Promise<void> {
5x      this.removeActivityModalStatus = null;
5x      const modelRef = this.ngbModal.open(
          RemoveActivityModalComponent, {backdrop: true});
5x      modelRef.componentInstance.sectionNameI18nId = sectionNameI18nId;
5x      modelRef.componentInstance.subsectionName = subsectionName;
5x      modelRef.componentInstance.activityId = activityId;
5x      modelRef.componentInstance.activityTitle = activityTitle;
5x      await modelRef.result.then((playlistUrl) => {
          // eslint-disable-next-line dot-notation
4x        this.http.delete<void>(playlistUrl).toPromise();
4x        this.removeActivityModalStatus = 'removed';
        }, () => {
          // Note to developers:
          // This callback is triggered when the Cancel button is clicked.
          // No further action is needed.
1x        this.removeActivityModalStatus = 'canceled';
        });

5x      return new Promise((resolve, reject) => {
5x        if (this.removeActivityModalStatus === 'removed') {
4x          resolve();
          }
        });
```

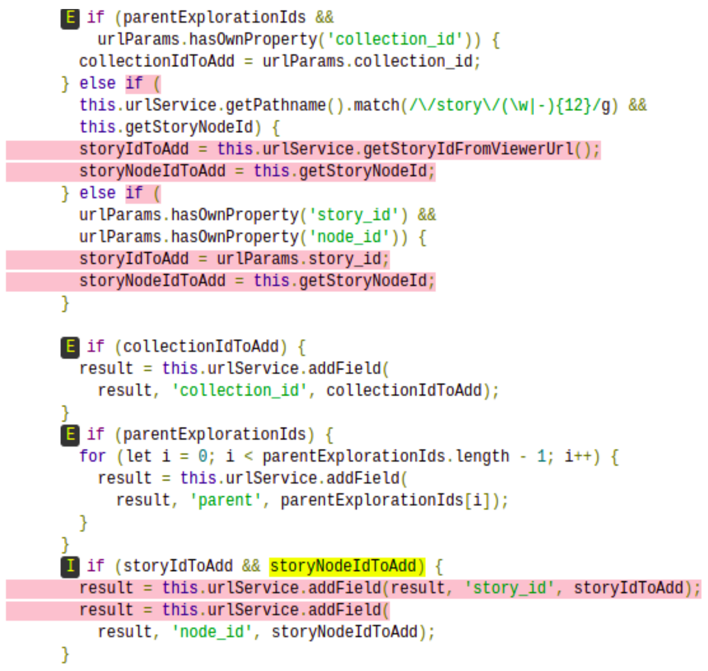


```
it('should opena a modal to remove an exploration from learner playlist' +
  ' when calling removeActivityModal', fakeAsync(() => {
  expect(learnerPlaylistBackendApiService.removeActivityModalStatus)
    .toBeUndefined;

  const modalSpy = spyOn(ngbModal, 'open').and.callFake((dlg, opt) => {
    setTimeout(opt.beforeDismiss);
    return <NgbModalRef>(
      { componentInstance: MockRemoveActivityNgbModalRef,
        result: Promise.resolve('url')
      });
  });

  let sectionNameI18nId = 'I18N_LEARNER_DASHBOARD_PLAYLIST_SECTION';
  let subsectionName = 'I18N_DASHBOARD_EXPLORATIONS';
  let activityId = '0';
  let activityTitle = 'title';

  learnerPlaylistBackendApiService.removeActivityModal(
    sectionNameI18nId, subsectionName,
    activityId, activityTitle);

  flushMicrotasks();

  expect(modalSpy).toHaveBeenCalled();
  expect(learnerPlaylistBackendApiService.removeActivityModalStatus)
    .toBe('removed');
}));
```

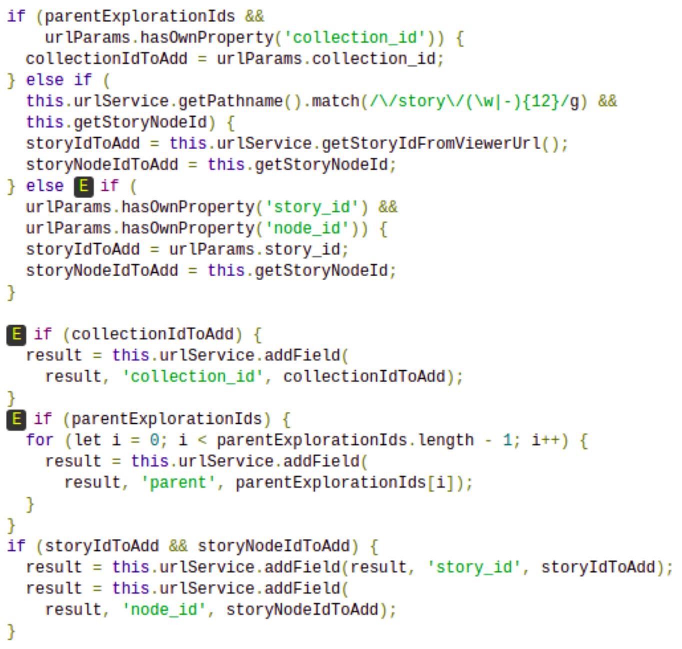
- Although I didn't test the HTTP request, the karma coverage report shows the line hit for that line.
- We should highly avoid these kinds of test cases in our spec file.
- Because, In future, the HTTP request may fail, but our test cases will pass which will introduce **flakes** (non-deterministic results) into our system.
- We can make this specific test case more robust by adding a HTTP check.

2.Unreachable Code

- Unreachable code is the most common issue in unit testing.
- It will occur when we don't test every path in a function or branch.
- As you can see below, there are a lot of if-else conditions and nested if-else. A single test case can't satisfy every if-else state, such that resulting in an unreachable test.
- So, we simply divide this function into multiple test cases to test each path individually.
- We should not test multiple conditions in a single test because if one condition fails, every other condition will also fail even though they are correct.



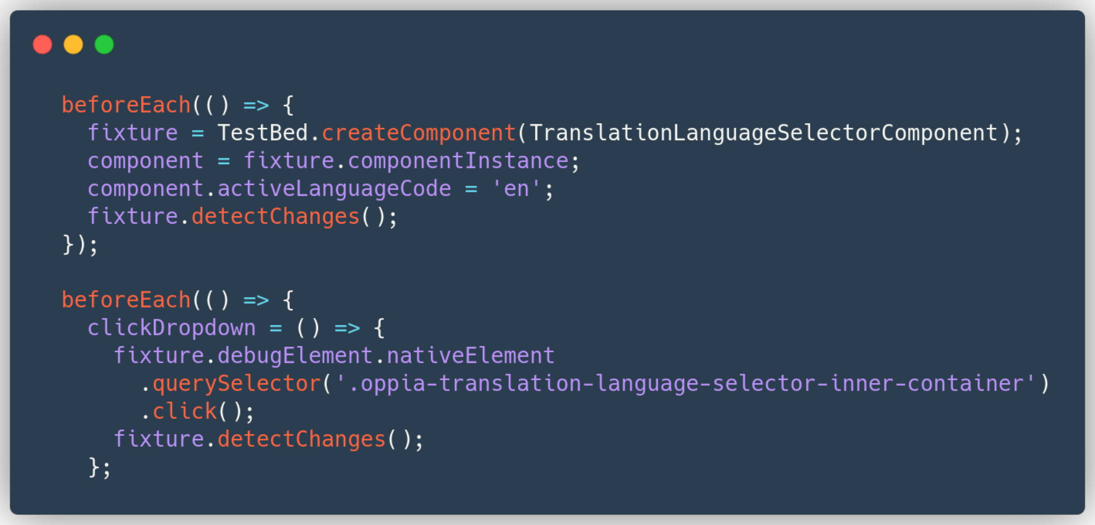
```
if (parentExplorationIds &&
    urlParams.hasOwnProperty('collection_id')) {
  collectionIdToAdd = urlParams.collection_id;
} else if (
  this.urlService.getPathname().match(/\/story\/(\w|-){12}/g) &&
  this.getStoryNodeId) {
  storyIdToAdd = this.urlService.getStoryIdFromViewerUrl();
  storyNodeIdToAdd = this.getStoryNodeId;
} else if (
  urlParams.hasOwnProperty('story_id') &&
  urlParams.hasOwnProperty('node_id')) {
  storyIdToAdd = urlParams.story_id;
  storyNodeIdToAdd = this.getStoryNodeId;
}

if (collectionIdToAdd) {
  result = this.urlService.addField(
    result, 'collection_id', collectionIdToAdd);
}
if (parentExplorationIds) {
  for (let i = 0; i < parentExplorationIds.length - 1; i++) {
    result = this.urlService.addField(
      result, 'parent', parentExplorationIds[i]);
  }
}
if (storyIdToAdd && storyNodeIdToAdd) {
  result = this.urlService.addField(result, 'story_id', storyIdToAdd);
  result = this.urlService.addField(
    result, 'node_id', storyNodeIdToAdd);
}
```



```
if (parentExplorationIds &&
    urlParams.hasOwnProperty('collection_id')) {
  collectionIdToAdd = urlParams.collection_id;
} else if (
  this.urlService.getPathname().match(/\/story\/(\w|-){12}/g) &&
  this.getStoryNodeId) {
  storyIdToAdd = this.urlService.getStoryIdFromViewerUrl();
  storyNodeIdToAdd = this.getStoryNodeId;
} else if (
  urlParams.hasOwnProperty('story_id') &&
  urlParams.hasOwnProperty('node_id')) {
  storyIdToAdd = urlParams.story_id;
  storyNodeIdToAdd = this.getStoryNodeId;
}

if (collectionIdToAdd) {
  result = this.urlService.addField(
    result, 'collection_id', collectionIdToAdd);
}
if (parentExplorationIds) {
  for (let i = 0; i < parentExplorationIds.length - 1; i++) {
    result = this.urlService.addField(
      result, 'parent', parentExplorationIds[i]);
  }
}
if (storyIdToAdd && storyNodeIdToAdd) {
  result = this.urlService.addField(result, 'story_id', storyIdToAdd);
  result = this.urlService.addField(
    result, 'node_id', storyNodeIdToAdd);
}
```

3.Testing UI Elements

- If you are wondering about testing UI through unit tests. Yes, we can test UI with the help of **fixture.debugElement** which is offered by Angular Testbed.
- We are currently relying on E2E tests entirely for UI tests. Since Oppia is a huge codebase, only critical paths have E2E tests available as of now. There are many flakes in them right now (We [Automated QA team] are prioritizing them to be resolved first).
- Instead of totally depending on E2E tests, we can include some UI tests in Unit tests.
- This will help us to find bugs quickly rather than waiting for all CI checks to run.
- These tests may not find every bug related UI because these are pretty much isolated. But, something is better than nothing.
- Currently **Oppia** has only **5 files** which use UI tests.
- For using this feature, we should create the component with the help of Testbed.
- The fixture will give us an instance of the component class.
- We can use it by concatenating with **debugElement**, which can grab any dom element with the help of **id** or **class** name.
- For Example, In case of modal, when we click any button it should open a modal. There is not much data handling happening there. We are triggering a click and the modal is displayed on the browser.We can write UI tests for such situations where data is not passing through the application but the UI is changing.

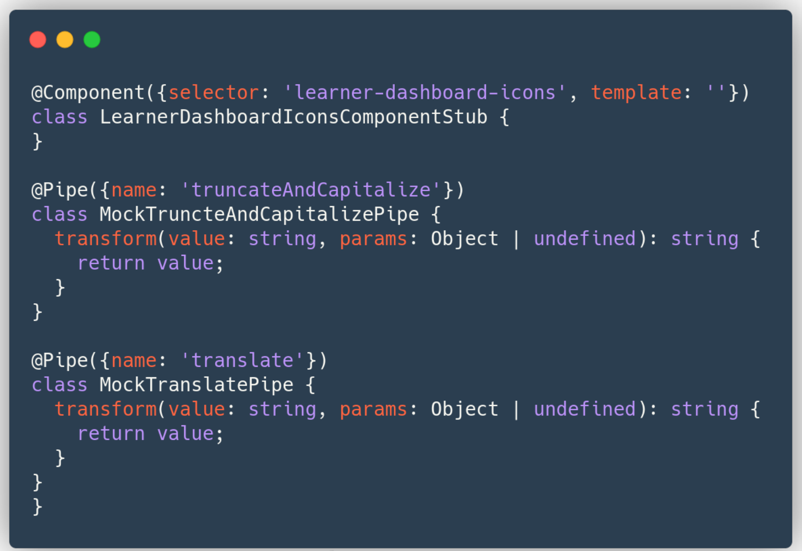


```
beforeEach(() => {
  fixture = TestBed.createComponent(TranslationLanguageSelectorComponent);
  component = fixture.componentInstance;
  component.activeLanguageCode = 'en';
  fixture.detectChanges();
});

beforeEach(() => {
  clickDropdown = () => {
    fixture.debugElement.nativeElement
      .querySelector('.oppia-translation-language-selector-inner-container')
      .click();
    fixture.detectChanges();
  };
```

Fig - Testing UI Elements

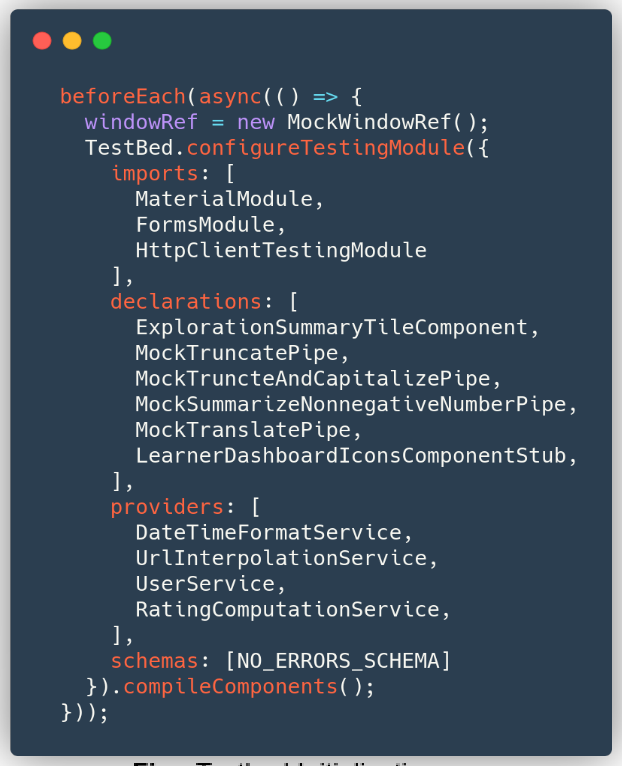
4.Template Parse Errors

- These are the most common errors. These occur when the Testbed is not initialized correctly.
- There are 4 main reasons behind this error (pipes, components, third party imports, schemas).
- We should check whether our template uses pipe, child components, and any third party components.
- In the case of pipe and components, we should use mock/stub components.
- For third party libraries, we should simply add them to imports in the Testbed.
- Note that we have included MaterialModule in imports (We have used <mat-card>...</mat-card> in the template which is a third party component).
- And we should finally add [NO_ERRORS_SCHEMA] in the schema section in the Testbed.



```
@Component({selector: 'learner-dashboard-icons', template: ''})
class LearnerDashboardIconsComponentStub {
}

@Pipe({name: 'truncateAndCapitalize'})
class MockTruncteAndCapitalizePipe {
  transform(value: string, params: Object | undefined): string {
    return value;
  }
}

@Pipe({name: 'translate'})
class MockTranslatePipe {
  transform(value: string, params: Object | undefined): string {
    return value;
  }
}
}
```

Fig - Mocking Components and Pipes

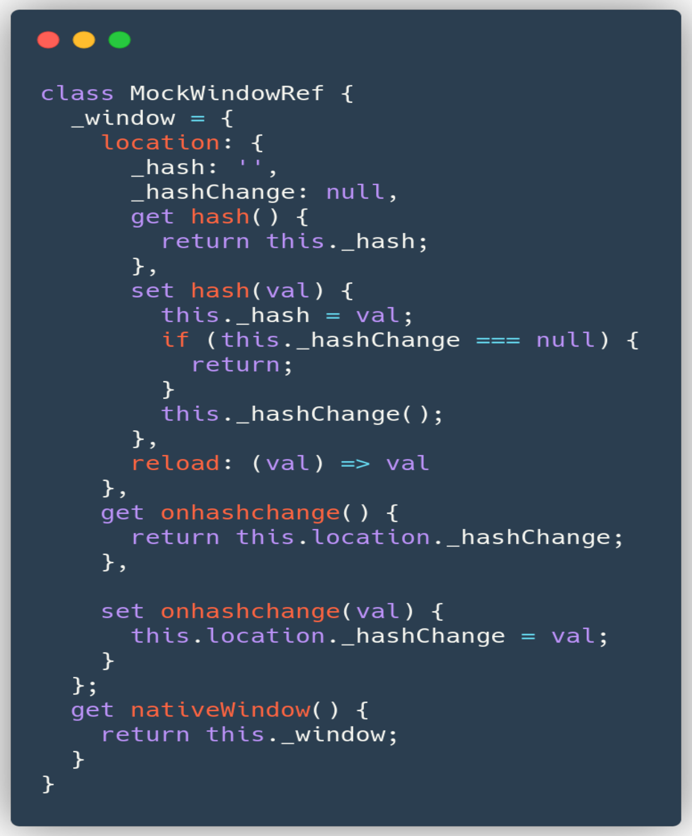


```
beforeEach(async(() => {
  windowRef = new MockWindowRef();
  TestBed.configureTestingModule({
    imports: [
      MaterialModule,
      FormsModule,
      HttpClientTestingModule
    ],
    declarations: [
      ExplorationSummaryTileComponent,
      MockTruncatePipe,
      MockTruncteAndCapitalizePipe,
      MockSummarizeNonnegativeNumberPipe,
      MockTranslatePipe,
      LearnerDashboardIconsComponentStub,
    ],
    providers: [
      DateTimeFormatService,
      UrlInterpolationService,
      UserService,
      RatingComputationService,
    ],
    schemas: [NO_ERRORS_SCHEMA]
  }).compileComponents();
}));
```

Fig - Testbed Initialization

5.Mocking WindowRef

- We should Mock window objects, because changing location.href causes the full page to reload.
- Page reloads raise an error in karma.

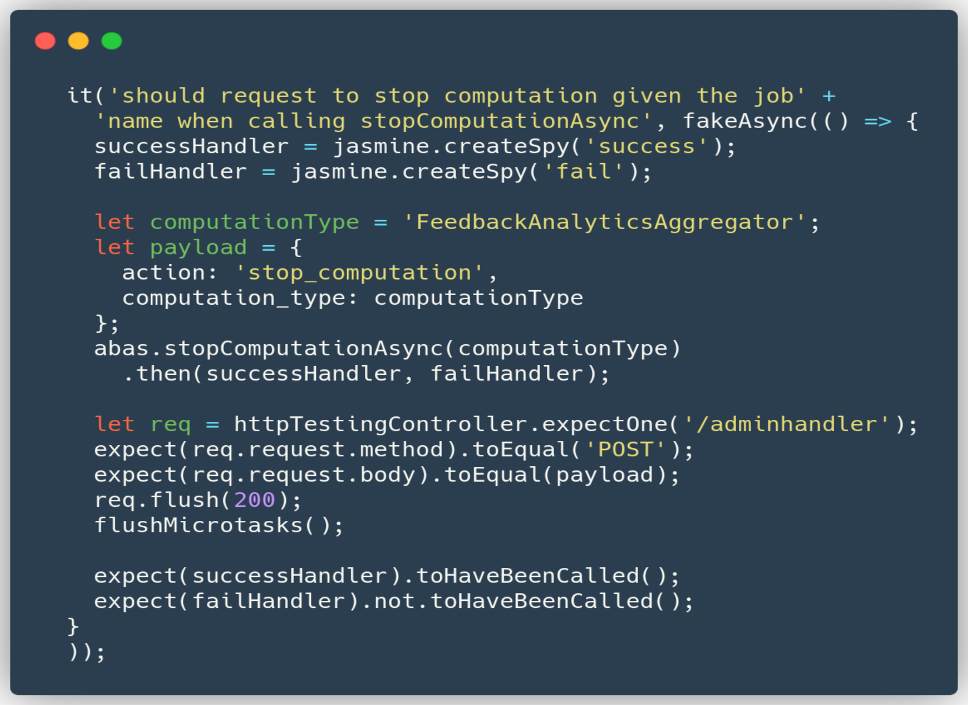


```
class MockWindowRef {
  _window = {
    location: {
      _hash: '',
      _hashChange: null,
      get hash() {
        return this._hash;
      },
      set hash(val) {
        this._hash = val;
        if (this._hashChange === null) {
          return;
        }
        this._hashChange();
      },
      reload: (val) => val
    },
    get onhashchange() {
      return this.location._hashChange;
    },

    set onhashchange(val) {
      this.location._hashChange = val;
    }
  };
  get nativeWindow() {
    return this._window;
  }
}
```

Fig - Mocking WindowRef

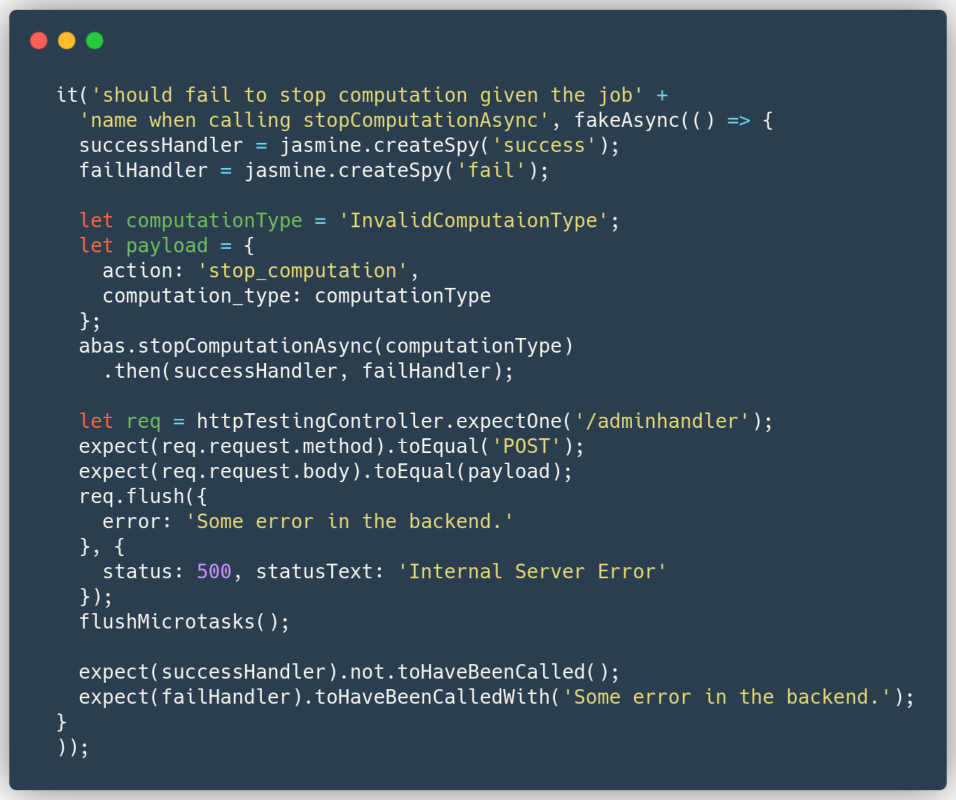
6.Testing Promises / HTTP

- Since Promises / HTTP requests consist of two call back functions, one will be triggered when the promise is resolved, and the other will be triggered when the promise is rejected.
- It is essential to test both the cases. Even Though Karma coverage will be satisfied with one case.
- We can achieve this by using two main components (spy objects, httpTestingController)
- We will be creating two spy objects (successHandler, failHandler).
- With the help of httpTestingController will create an instance for particular request (Ex: ‘POST’)
- By concatenating .flush( ) to the instance we can easily mock the HTTP request, Instead actually sending one to the server.



```
it('should request to stop computation given the job' +
  'name when calling stopComputationAsync', fakeAsync(() => {
  successHandler = jasmine.createSpy('success');
  failHandler = jasmine.createSpy('fail');

  let computationType = 'FeedbackAnalyticsAggregator';
  let payload = {
    action: 'stop_computation',
    computation_type: computationType
  };
  abas.stopComputationAsync(computationType)
    .then(successHandler, failHandler);

  let req = httpTestingController.expectOne('/adminhandler');
  expect(req.request.method).toEqual('POST');
  expect(req.request.body).toEqual(payload);
  req.flush(200);
  flushMicrotasks();

  expect(successHandler).toHaveBeenCalled();
  expect(failHandler).not.toHaveBeenCalled();
}
));
```

Fig - Http Successful Request



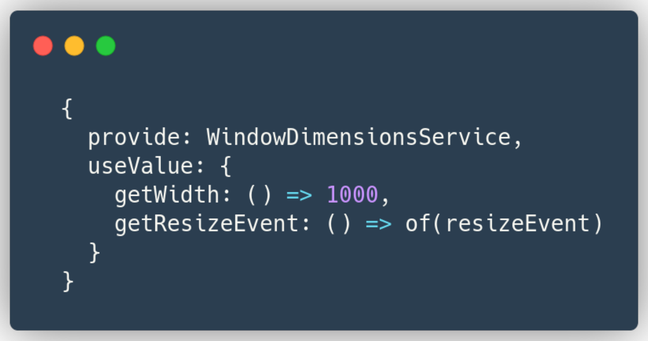
```
it('should fail to stop computation given the job' +
  'name when calling stopComputationAsync', fakeAsync(() => {
  successHandler = jasmine.createSpy('success');
  failHandler = jasmine.createSpy('fail');

  let computationType = 'InvalidComputaionType';
  let payload = {
    action: 'stop_computation',
    computation_type: computationType
  };
  abas.stopComputationAsync(computationType)
    .then(successHandler, failHandler);

  let req = httpTestingController.expectOne('/adminhandler');
  expect(req.request.method).toEqual('POST');
  expect(req.request.body).toEqual(payload);
  req.flush({
    error: 'Some error in the backend.'
  }, {
    status: 500, statusText: 'Internal Server Error'
  });
  flushMicrotasks();

  expect(successHandler).not.toHaveBeenCalled();
  expect(failHandler).toHaveBeenCalledWith('Some error in the backend.');
}
));
```

Fig - Http Failure Case

- Note that we have used the status codes (200, 500) in req.flush(...) In order to reproduce success call back and failure callback.

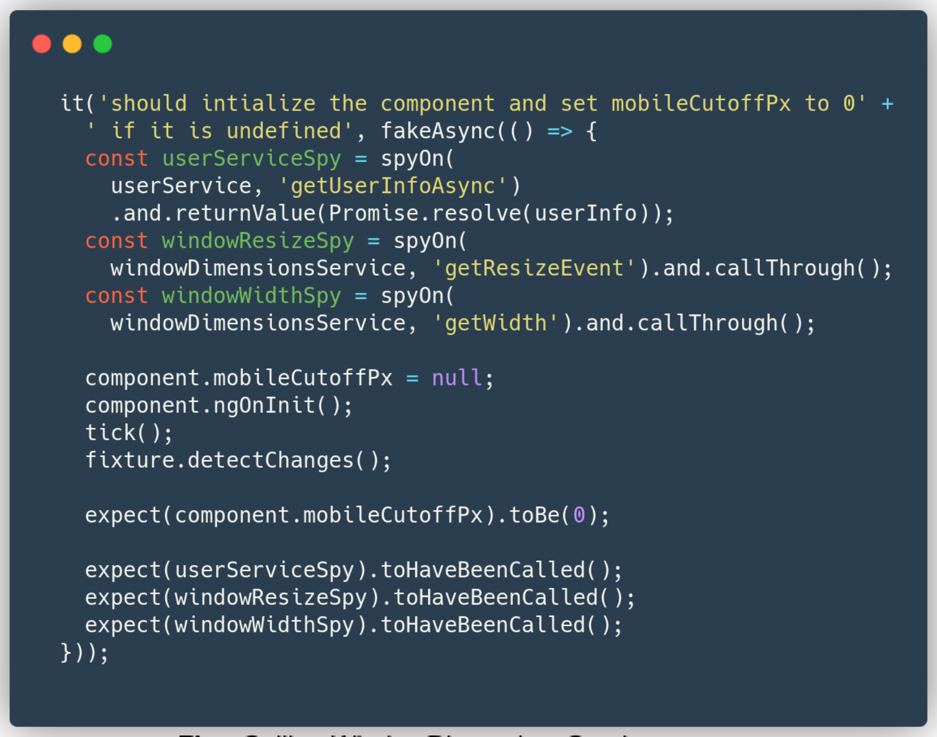
7.Testing Observables

- Observables can be considered as an asynchronous data stream of elements.
- These are used throughout the angular applications.
- Observables are mainly used to perform event handling, asynchronous programming, and handling multiple values, making them hard to test.
- When I was migrating and testing the **exploration-summary-tile.component**, It has dependency on WindowDimensionService.
- WindowDimensionService consists of method **getResizeEvent** which returns an observable.
- Inorder to test this method, I've mocked WindowDimensionService in the providers array as shown below.
- Firstly we should
  `import { of } from 'rxjs';`
- And then we should create a new resize event as follow
  `let resizeEvent = new Event('resize');`
- This will help us to mock observable behaviour.



```
{
  provide: WindowDimensionsService,
  useValue: {
    getWidth: () => 1000,
    getResizeEvent: () => of(resizeEvent)
  }
}
```

Fig - Mocking WindowDimensionsService

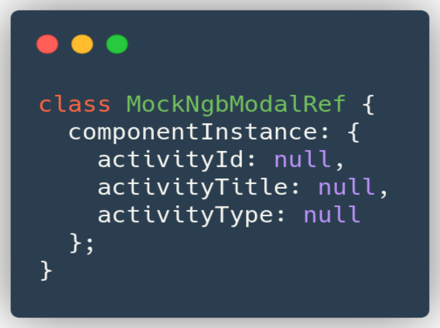


```
it('should intialize the component and set mobileCutoffPx to 0' +
  ' if it is undefined', fakeAsync(() => {
  const userServiceSpy = spyOn(
    userService, 'getUserInfoAsync')
    .and.returnValue(Promise.resolve(userInfo));
  const windowResizeSpy = spyOn(
    windowDimensionsService, 'getResizeEvent').and.callThrough();
  const windowWidthSpy = spyOn(
    windowDimensionsService, 'getWidth').and.callThrough();

  component.mobileCutoffPx = null;
  component.ngOnInit();
  tick();
  fixture.detectChanges();

  expect(component.mobileCutoffPx).toBe(0);

  expect(userServiceSpy).toHaveBeenCalled();
  expect(windowResizeSpy).toHaveBeenCalled();
  expect(windowWidthSpy).toHaveBeenCalled();
}));
```

Fig - Calling WindowDimensionsService

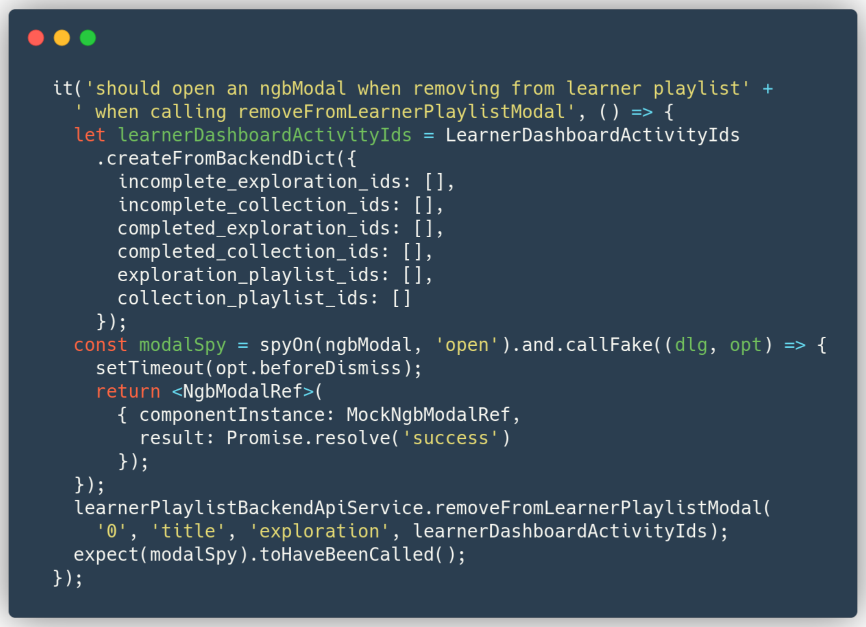
8.Testing NgbModal

- Testing NgbModal is challenging because of two reasons.
- It is a UI component, which is dependent on another angular component.
- Opening NgbModal(UI modal) will not return Promise. Instead, it will return NgbModal's Reference.
- In order to tackle this situation I've created a stub class which consists of all properties related to Modal's
  reference.



```
class MockNgbModalRef {
  componentInstance: {
    activityId: null,
    activityTitle: null,
    activityType: null
  };
}
```

Fig - Mocking NgbModalRef



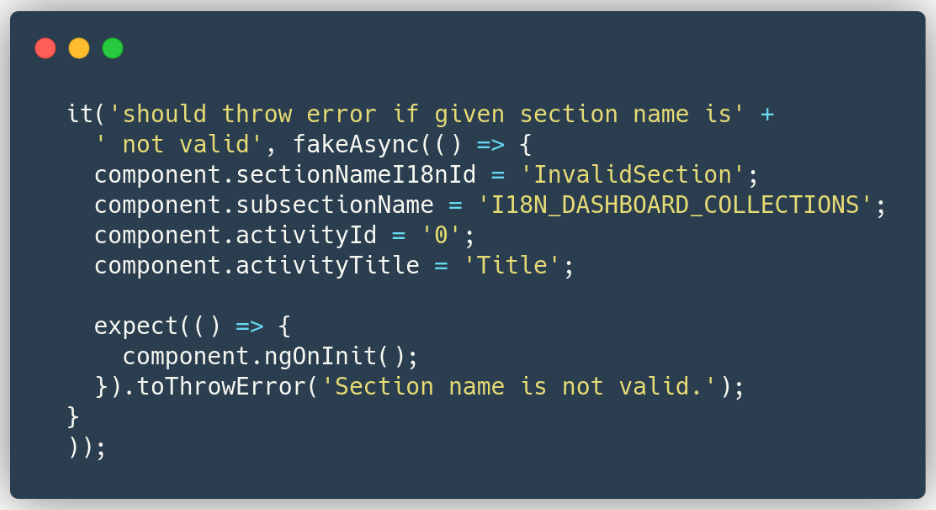
```
it('should open an ngbModal when removing from learner playlist' +
  ' when calling removeFromLearnerPlaylistModal', () => {
  let learnerDashboardActivityIds = LearnerDashboardActivityIds
    .createFromBackendDict({
      incomplete_exploration_ids: [],
      incomplete_collection_ids: [],
      completed_exploration_ids: [],
      completed_collection_ids: [],
      exploration_playlist_ids: [],
      collection_playlist_ids: []
    });
  const modalSpy = spyOn(ngbModal, 'open').and.callFake((dlg, opt) => {
    setTimeout(opt.beforeDismiss);
    return <NgbModalRef>(
      { componentInstance: MockNgbModalRef,
        result: Promise.resolve('success')
      });
  });
  learnerPlaylistBackendApiService.removeFromLearnerPlaylistModal(
    '0', 'title', 'exploration', learnerDashboardActivityIds);
  expect(modalSpy).toHaveBeenCalled();
});
```

Fig - Calling NgbModalRef

- I've used spyOn on the ngbModal (Service), we can not return a promise here, So I've used callFake to return customized output i.e, **<NgbModalRef>.**
- In the <NgbModalRef> we have another attribute named **result** which returns a promise.
- If it resolves then the modal will be opened, if it rejects then the modal is closed.
- I've implemented both the test cases. You can find the rest of the implementation here.

9.Test Cases for Error Handling

- We can't test the functions which throw errors in the traditional approach.
- Because, if we invoke a function like that, it will also throw an error in the test suite which results in a failure test case.
- Jasmin offers two methods for handling these types of cases.
- 1. expect(() => { throw new Error (...) }).toThrow(...);
- 2. expect(() => { ... }).toThrowError(...);
- In the below example ngOnInit throws an error on initialization. I've handle it using **.toThrowError** method



```
it('should throw error if given section name is' +
  ' not valid', fakeAsync(() => {
  component.sectionNameI18nId = 'InvalidSection';
  component.subsectionName = 'I18N_DASHBOARD_COLLECTIONS';
  component.activityId = '0';
  component.activityTitle = 'Title';

  expect(() => {
    component.ngOnInit();
  }).toThrowError('Section name is not valid.');
}
));
```

Fig - Testing Error Handling

What are Flakes?

- A flaky test is a test that both passes and fails periodically without any code changes.
- Flaky tests are quite costly since they often require developers to retrigger entire builds on CI.
- I've found a couple of flakes in FE test #12316, #12142.
- Flakes need high priority because, If they are not resolved, it will slow down the development workflow, which is very annoying.

Causes for Flakes

- Each test should be truly **isolated**, which means we should not use any dependent service / pipes in our test suite directly
- We should test **single unit at a time,** Each test case should have only one purpose, if we try to test multiple conditions in our test case it will result in flakiness.
- A test case must have proper **Setup** and **Teardown,** This will ensure that all states which are necessary for the test case are created before each test case and also destroyed after each test case.
- Tests could fail nondeterministically when the processing takes longer than the timeout value. **Different environments** have different processing power. This might cause flakiness

Handling Flakes

- The first thing we should do is to collect as much information as possible.
- It will include running tests several times, logging error messages, checking the probability of flakiness etc.
- We should go through the commit history, find out which commit has recently changed file, from when it started flaking.
- Find the root cause of the flake.
- Fixing the flake.

Milestones

Since, Files are assigned to the students only after getting selected(Specifically for this project). I'm dedicating time for the files very broadly.

Note:

- Milestones are further refined in the community bonding period after files are assigned to us.
- Below **lines** represent **line hits** which are calculated in istanbul (Karma coverage report), **Not lines of code**.

Milestone 1

Key Objective: Complete half of the assigned files (1650 lines)
Time: June 7 - July 11

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
1.1	Test 330 lines		7/06/2021	12/06/2021
1.2	Test 330 lines		13/06/2021	18/06/2021
1.3	Test 330 lines		19/06/2021	24/06/2021
1.4	Test 330 lines		25/06/2021	01/07/2021
1.5	Test 330 lines		02/07/2021	07/07/2021
1.6	Buffer Period for Milestone 1		08/07/2021	11/07/2021

Milestone 2

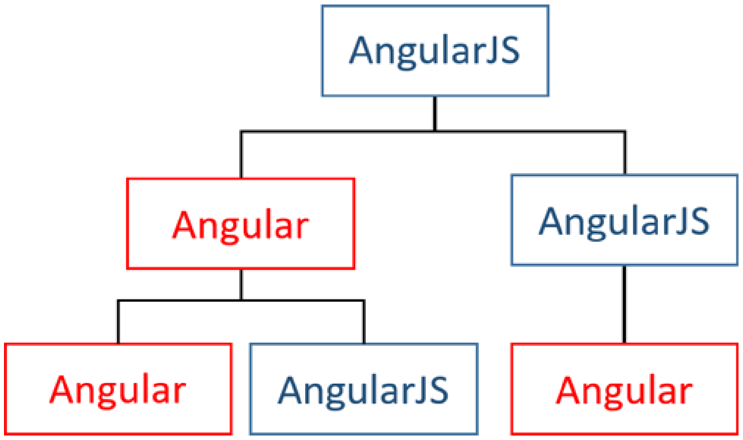
Key Objective: Complete other half of the assigned files (1650 lines)
Time: July 12 - August 15

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
2.1	Test 330 lines		12/07/2021	17/07/2021
2.2	Test 330 lines		18/07/2021	23/07/2021
2.3	Test 330 lines		24/07/2021	28/07/2021
2.4	Test 330 lines		29/07/2021	04/08/2021
2.5	Test 330 lines		05/08/2021	10/08/2021
2.6	Buffer Period for Milestone 2		11/08/2021	15/08/2021

Appendix (Optional Section)

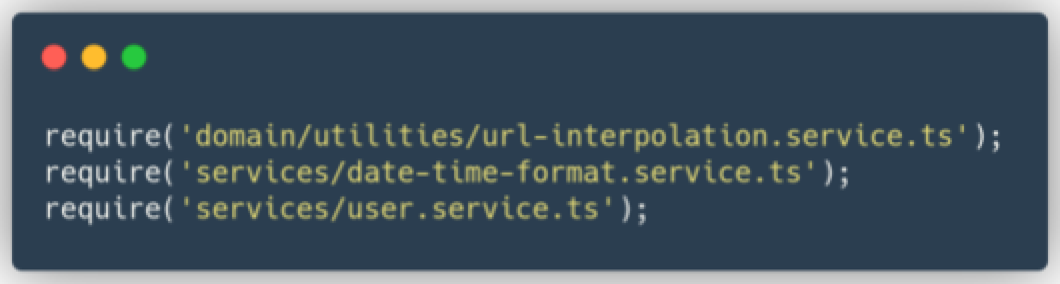
Migration

- Dependencies are the core and crux of migration architectures.
- We can not directly migrate the parent class without migrating its child classes.
- All components/services should be migrated from lower topological order to higher.
- I usually think of migration as playing an adventure game. In order to defeat the boss, we must go through all mini-boss levels.
- We have two options when dealing with dependency issues for synchronization in between higher-level components (angular) and lower-level directives (angular js).
- We can either upgrade the lower-level directives(angular js).
- Or we can downgrade the higher-level component(angular).



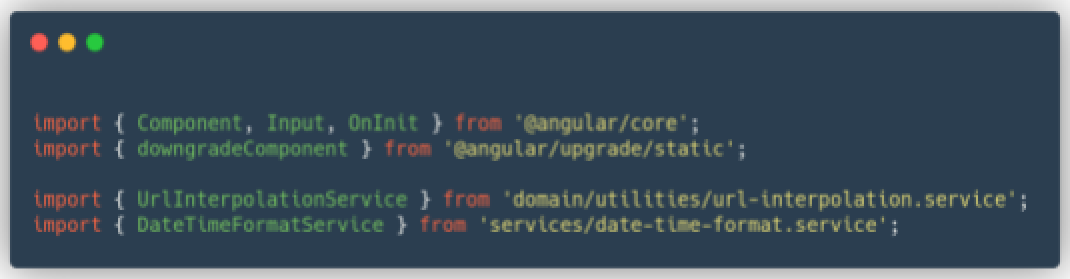
Migrating Directives/Controllers

- The foremost thing we should be doing is checking whether the directive/controller depends on any other directive/service.
- If so, we should check whether the directive/service is migrated or upgraded.
- We should change the names of the corresponding files by removing directive/controller and adding *component.ts
- Changing directive declaration to component declaration.
- Changing ES5 (require) import system to ES6 (import--from).



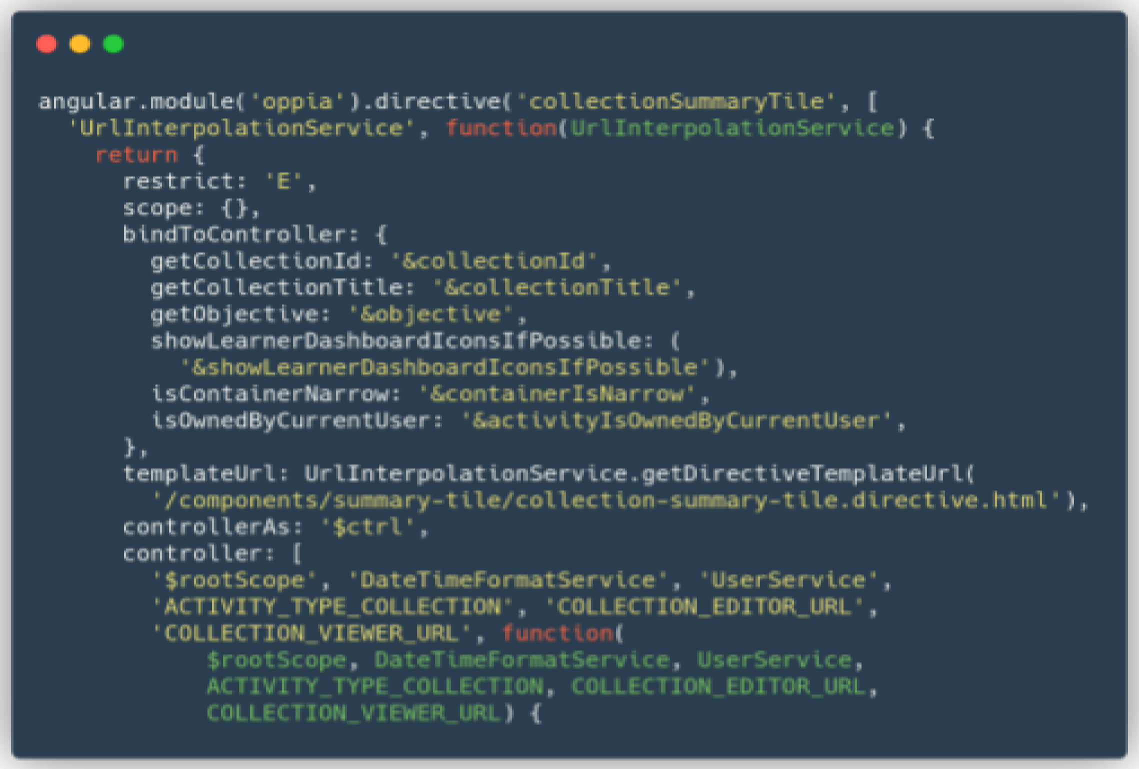
```
require('domain/utilities/url-interpolation.service.ts');
require('services/date-time-format.service.ts');
require('services/user.service.ts');
```

Fig - ES5 imports



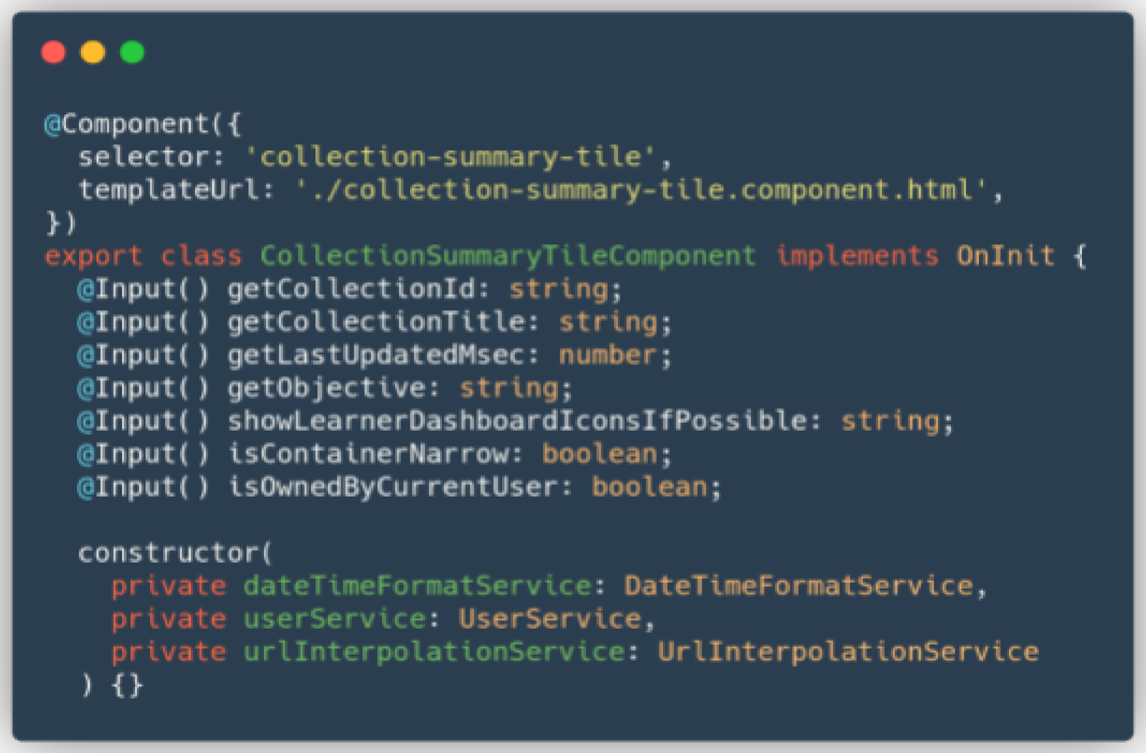
```
import { Component, Input, OnInit } from '@angular/core';
import { downgradeComponent } from '@angular/upgrade/static';

import { UrlInterpolationService } from 'domain/utilities/url-interpolation.service';
import { DateTimeFormatService } from 'services/date-time-format.service';
```

Fig - ES6 imports



```
angular.module('oppia').directive('collectionSummaryTile', [
  'UrlInterpolationService', function(UrlInterpolationService) {
    return {
      restrict: 'E',
      scope: {},
      bindToController: {
        getCollectionId: '&collectionId',
        getCollectionTitle: '&collectionTitle',
        getObjective: '&objective',
        showLearnerDashboardIconsIfPossible: (
          '&showLearnerDashboardIconsIfPossible'),
        isContainerNarrow: '&containerIsNarrow',
        isOwnedByCurrentUser: '&activityIsOwnedByCurrentUser',
      },
      templateUrl: UrlInterpolationService.getDirectiveTemplateUrl(
        '/components/summary-tile/collection-summary-tile.directive.html'),
      controllerAs: '$ctrl',
      controller: [
        '$rootScope', 'DateTimeFormatService', 'UserService',
        'ACTIVITY_TYPE_COLLECTION', 'COLLECTION_EDITOR_URL',
        'COLLECTION_VIEWER_URL', function(
            $rootScope, DateTimeFormatService, UserService,
            ACTIVITY_TYPE_COLLECTION, COLLECTION_EDITOR_URL,
            COLLECTION_VIEWER_URL) {
```

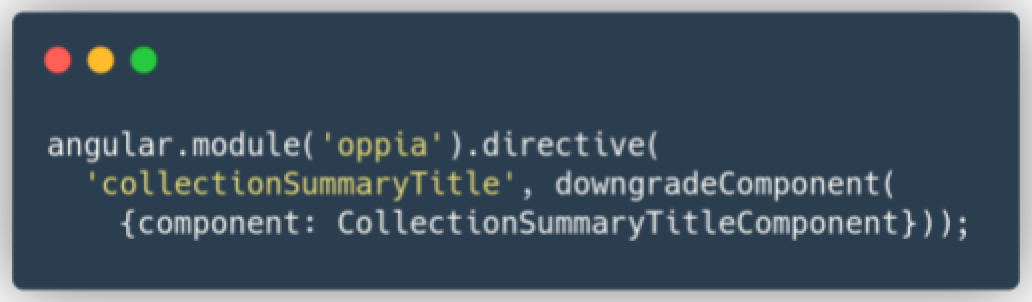
Fig - Angular JS Directive Declaration



```
@Component({
  selector: 'collection-summary-tile',
  templateUrl: './collection-summary-tile.component.html',
})
export class CollectionSummaryTileComponent implements OnInit {
  @Input() getCollectionId: string;
  @Input() getCollectionTitle: string;
  @Input() getLastUpdatedMsec: number;
  @Input() getObjective: string;
  @Input() showLearnerDashboardIconsIfPossible: string;
  @Input() isContainerNarrow: boolean;
  @Input() isOwnedByCurrentUser: boolean;

  constructor(
    private dateTimeFormatService: DateTimeFormatService,
    private userService: UserService,
    private urlInterpolationService: UrlInterpolationService
  ) {}
```

Fig - Angular Component Declaration

- All services that are used by the component should be injected into the constructor. It is also known as dependency injection.
- Replacing bindToControllers with @input and @output.
- After the above change, we should change the directive bindings in every file(templates) where the current component is used.
- If it is an Angular JS template, change layout-align-type=layoutAlign → [layout-align-type]=layoutAlign (Kebab Case).
- If it is an Angular template, change layout-align-type=layoutAlign --> [layoutAlignType]=layoutAlign (Camel Case).
- Removing all `$ctrl` and `$scope` attributes and replacing them with `this`.
- Finally, downgrading the component so that it can also be used in the AngularJS ecosystem.

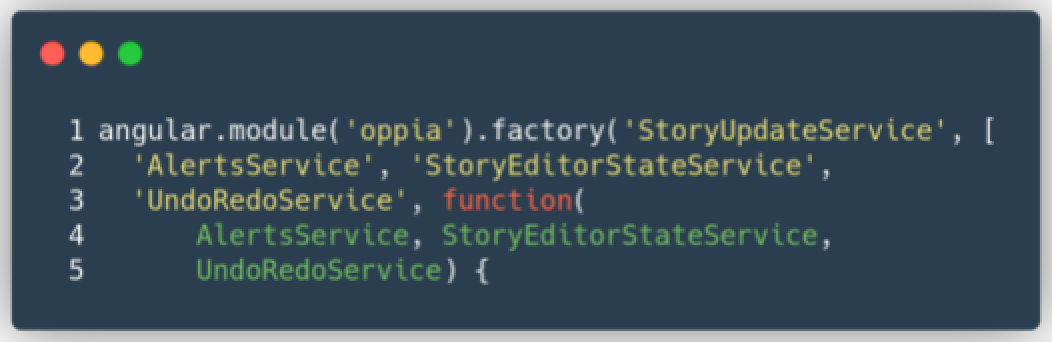


```
angular.module('oppia').directive(
  'collectionSummaryTitle', downgradeComponent(
    {component: CollectionSummaryTitleComponent}));
```

Fig - Downgraded Component

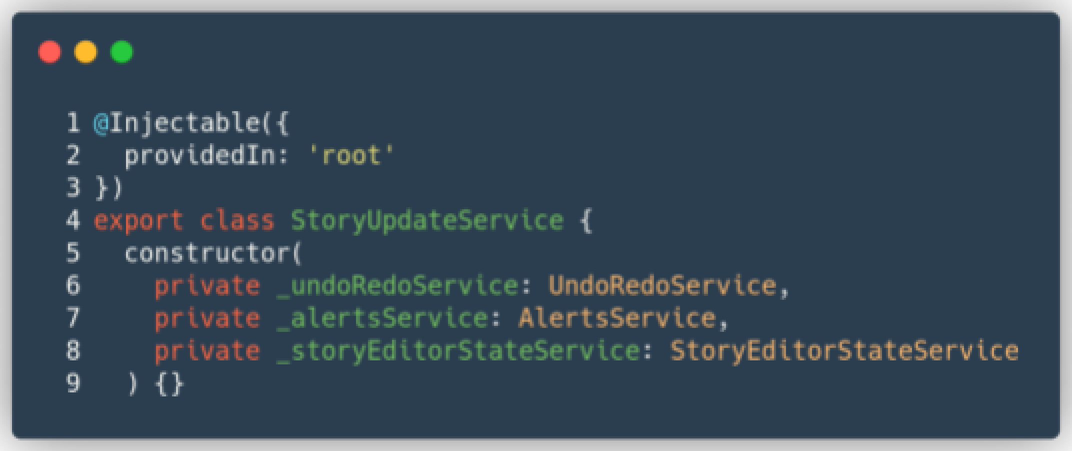
Migrating Services

- Migrating service is relatively easy when compared to migrating directives.
- The primary thing we should be doing is checking whether the service depends on any other service.
- If so, we should check whether the depending service is migrated or upgraded.
- We should change the names of the corresponding files by adding *backend-api.service.ts
- Changing AngularJS factory declaration to Angular service declaration.
- Changing ES5 (require) import system to ES6 (import--from).



```
angular.module('oppia').factory('StoryUpdateService', [
  'AlertsService', 'StoryEditorStateService',
  'UndoRedoService', function(
      AlertsService, StoryEditorStateService,
      UndoRedoService) {
```

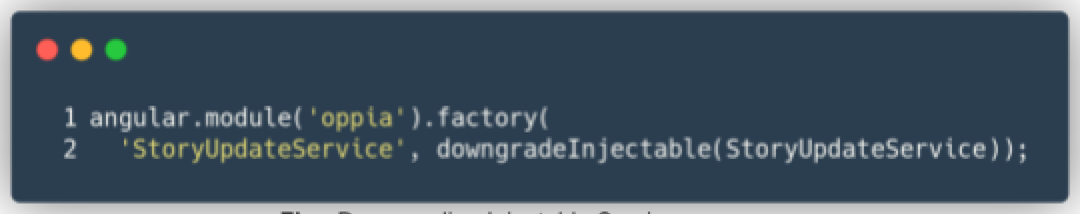
Fig - Angular JS factory



```
@Injectable({
  providedIn: 'root'
})
export class StoryUpdateService {
  constructor(
    private _undoRedoService: UndoRedoService,
    private _alertsService: AlertsService,
    private _storyEditorStateService: StoryEditorStateService
  ) {}
```

Fig - Angular Injectable Service

- @injectable enables angular to use dependency injection. It will provide access to the entire application for using the corresponding service.
- AngularJS $http will return a promise, whereas Angular's HttpClient (Injectable Service) will return an observable. In order to replicate the same behaviour we will use .toPromise() method which will convert an observable to promise.
- Finally, downgrading the service so that it can also be used in the AngularJS ecosystem.



```
angular.module('oppia').factory(
  'StoryUpdateService', downgradeInjectable(StoryUpdateService));
```

Fig - Downgrading Injectable Service

Some Common Errors

Change Detection:

- After Migrating, we might be facing issues regarding e2e test case failures.
- The most common reason for that is change detection.
- Angular's change detection is fully automated. It is controlled by Zone.js
- Whereas in AngularJS, we have to explicitly mention where change detection is required.
- When AngularJs directives are depending on Angular's components, these types of errors will occur.
- In order to solve those errors, we must use $rootscope.$apply() in case of normal functions (Synchronous). $rootScope.$applyAsync() in case of Promises (Asynchronous).

View Encapsulation:

- Angular uses View Encapsulation, whereas, in angular JS, we have to explicitly Encapsulate CSS style from the root component (oppia.css) from the rest of the files in the codebase.
- This has already been implemented. The issue will occur after migration,
- When we use manual encapsulation in Angular components.

```
learner-dashboard-page .oppia-explorations-select {
.oppia-explorations-select {
  margin-bottom: 7px;
}
```

- For Instance, in the above example we should remove `learner-dashboard-page` after completion of migration

Material Css:

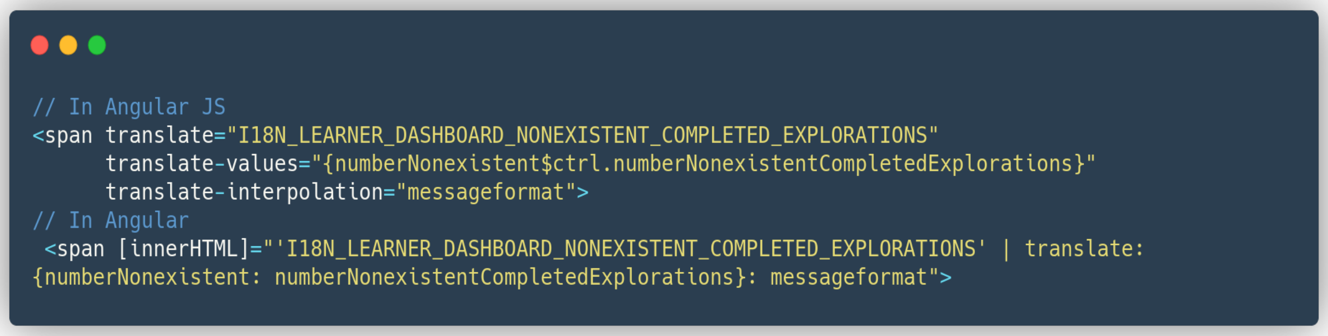
- Since most of our codebase third-party style sheets like bootstrap and material designs, most of the features from Angular JS are not supported by Angular (deprecated).
- After Migrating instead of AngularJS Material we should use Angular's material. Otherwise we will get template parse errors.

- For Instance, in the below example we should replace <md-card --- > with <mat-card --- >

```
    ng-class="{'oppia-learner-dashboard-mobile-mode': $ctrl.checkMobileView()}"
    ng-if="!$ctrl.noActivity">
 <md-card layout="row"
         ng-style="$ctrl.checkMobileView() ? {'margin-bottom': '10px'} : {'margin-bottom': '40px'}">
    [ngClass]="{'oppia-learner-dashboard-mobile-mode': checkMobileView()}"
    *ngIf="!noActivity">
 <mat-card class="oppia-flex-row"
         [ngStyle]="checkMobileView() ? {'margin-bottom': '10px'} : {'margin-bottom': '40px'}">
```

Translations:

- How to migrate translations are mentioned on the wiki page up to some extent.
- But tricky cases like nested translations and message format in translations are not mentioned.
- I've figured out a way to implement them. And refactored the code base in order to perform these translations. Here is the link to the corresponding PR



```
// In Angular JS
<span translate="I18N_LEARNER_DASHBOARD_NONEXISTENT_COMPLETED_EXPLORATIONS"
      translate-values="{numberNonexistent$ctrl.numberNonexistentCompletedExplorations}"
      translate-interpolation="messageformat">
// In Angular
 <span [innerHTML]="'I18N_LEARNER_DASHBOARD_NONEXISTENT_COMPLETED_EXPLORATIONS' | translate:
{numberNonexistent: numberNonexistentCompletedExplorations}: messageformat">
```

Filters vs Pipes:

- AngularJS uses filters, and Angular uses pipes in order to use map-reduce operations in the template (HTML) itself.
- The issue here is some of the filters from AngularJS are deprecated.
- In Angular, we don't have pipes for such filters. In that case, we should build custom pipes for replicating the behaviour of AngularJS filters.
- For example, we have an orderBy filter in AngularJS. But we don't have the exact same pipe in Angular.
- orderBy is used for sorting arrays/objects in ascending or descending order.
- I've built a custom pipe sortBy in order to replicate the behaviour of the orderBy filter here.

Note : I'm currently preparing **common errors/debugging doc**, In order to help the fellow contributors who are relatively new to the codebase. It is also helpful for mentors, they don't need to mention the solution for the same issue/error over and over again. This will boost the migration process which we have deadline to complete it before the starting date GSoC

Future Work

1. Angular Migration Team
    - Completing migration of angular (Object factories are left behind as they are not blocking anything else).
    - They should be converted into *model.ts
2. Automated QA Team
    - Writing tests for **others** sections as mentioned above (They are not in the scope of GSoC).
    - We can not rely on frontend tests completely. We should have E2E tests for each path in the application. (Currently entire application is not covered)
    - Resolve flakiness in E2E tests.