About You

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

To learn angular, I bought an online course. I searched for 'courses in angular', selected the top-rated one, and purchased it without the blink of an eye. Fast forward to 30th Jan 2021; the Oppia 2021 Strategy meet showed the stories of children from countries like Palestine, India etc. and how Oppia has helped them. Not just them, their parents too. The meeting made me realise how easy it was for me to learn anything I wanted to. I am privileged. But not everyone is...

250 million children in the world do not have basic literacy and numeracy; children in countries amidst conflicts drop out of school, girls are denied education. The right to education may exist, but the means certainly do not. This is where Oppia comes into play. Oppia aims to provide education to all underserved children across the world in an effective, accessible and engaging way.

I started contributing to Oppia back in September'20, and ever since, I have enjoyed contributing. I feel happy that even a tiny bug fix done by me plays a role in providing education to underserved children. And this is what keeps me motivated.

Project chosen - Automated QA Team - **Write Frontend Tests**.

Oppia uses AngularJs/Angular 2+ for the frontend and has a massive codebase. It is crucial to test this code in order to catch any bug before it reaches the 'develop' branch. Increasing the code coverage would also be helpful to other developers, as 100% coverage would reduce the chances of any bug popping out at a later stage. I find digging through the huge codebase and debugging very interesting. This project would be the best way for me to gain expertise in unit testing and Angular.

Prior experience

I have been into development for the past year, have learnt quite a few technologies and started contributing to open source in this time period. I am familiar with TypeScript and Python, and various front-end and backend frameworks like Angular/AngularJS, VueJS, Django Rest Framework. I have worked on a few college projects and personal projects which are visible on my GitHub profile.

I have been contributing to Oppia for the past 6 months, and have submitted a number of PRs, gaining familiarity with the codebase. I am currently working with the Learner and Creator Experience team.

PRs submitted:

1. #12189 - Migrated components and wrote unit tests.
2. #12216 - Added unit tests for components.
3. #11021 - Migrated and added unit tests for a service.
4. #12010 - Added e2e test for the student user journey.
5. #11701 - Added e2e test for topic and story viewer page.

List of all of my pull requests and issues created.

Contact info and timezone(s)

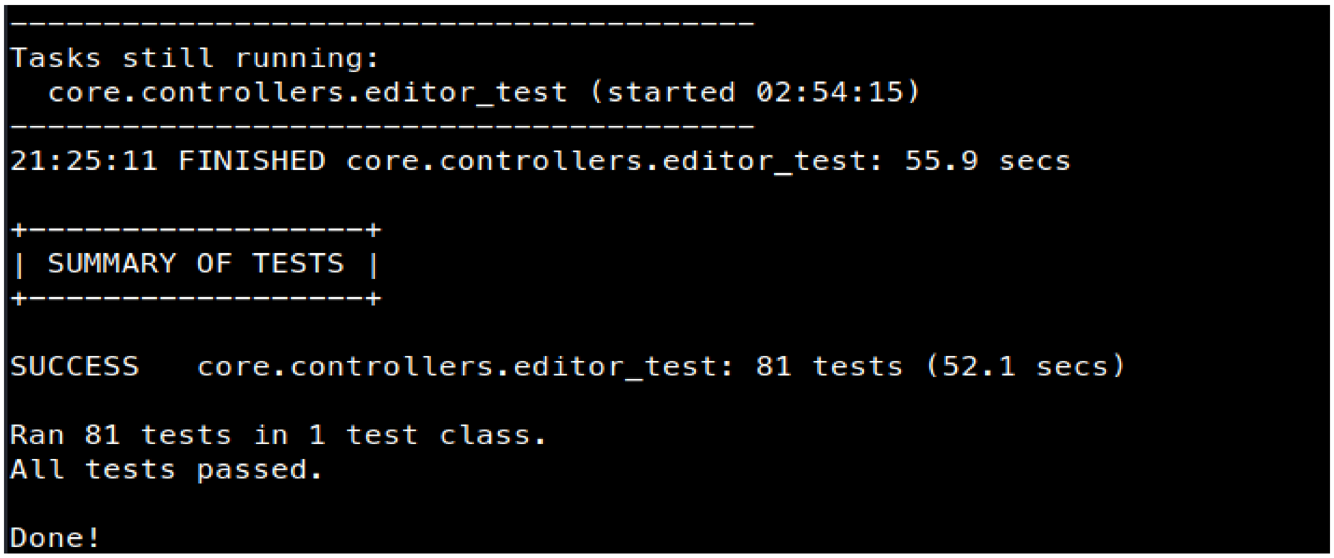
Email - aishwary.saxena.min19@iitbhu.ac.in
Phone - +91 8299329212
Timezone - Indian Standard Time (IST) (+5:30 GMT)
Preferred Method of communication - Hangouts, Email, WhatsApp

Time commitment

- I would be working on the GSoC project throughout the 10-week period, from 7th June to 16th August.
- I would commit at least 3-4 hrs per day during the coding period, or more if required.
- Currently, my college campus is closed, but it may so happen that the campus reopens during the summer to complete lab courses. If that is the case, I would increase the time before returning to the campus to 5-6 hrs per day and I would commit 2-3 hrs per day once I am on campus.

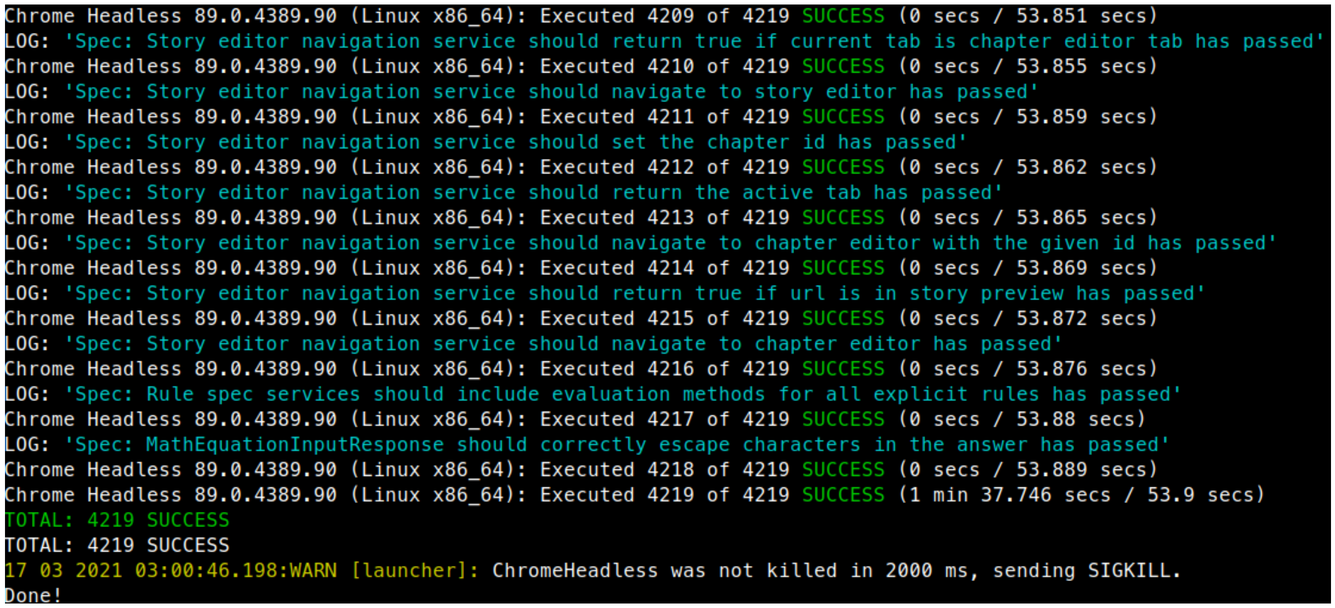
Essential Prerequisites

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



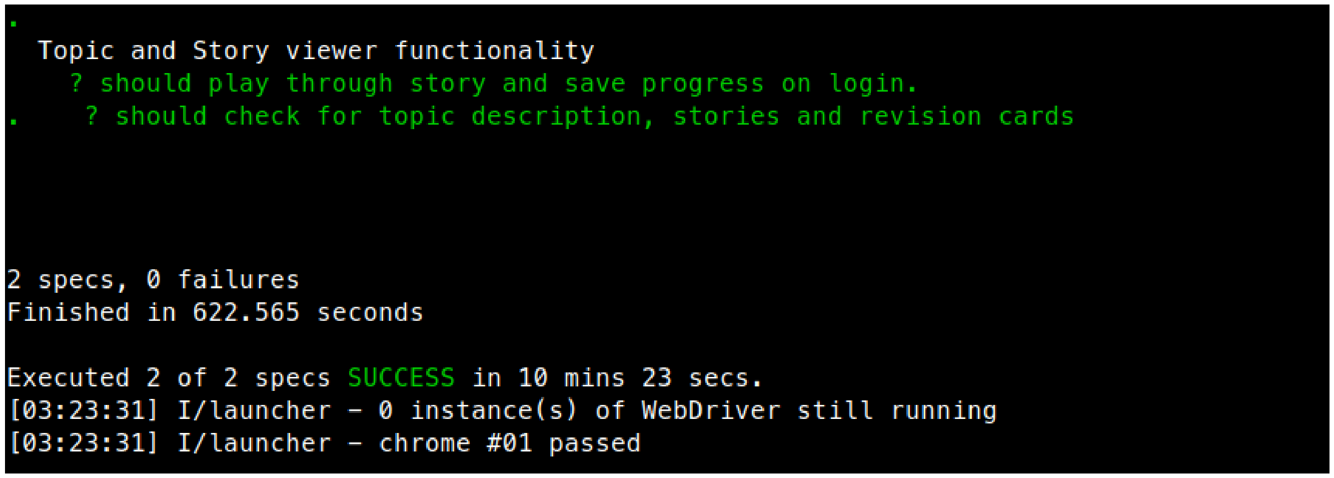
```
----------------------------------------
Tasks still running:
  core.controllers.editor_test (started 02:54:15)
----------------------------------------
21:25:11 FINISHED core.controllers.editor_test: 55.9 secs

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

SUCCESS   core.controllers.editor_test: 81 tests (52.1 secs)

Ran 81 tests in 1 test class.
All tests passed.

Done!
```

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



```
Chrome Headless 89.0.4389.90 (Linux x86_64): Executed 4209 of 4219 SUCCESS (0 secs / 53.851 secs)
LOG: 'Spec: Story editor navigation service should return true if current tab is chapter editor tab has passed'
Chrome Headless 89.0.4389.90 (Linux x86_64): Executed 4210 of 4219 SUCCESS (0 secs / 53.855 secs)
LOG: 'Spec: Story editor navigation service should navigate to story editor has passed'
Chrome Headless 89.0.4389.90 (Linux x86_64): Executed 4211 of 4219 SUCCESS (0 secs / 53.859 secs)
LOG: 'Spec: Story editor navigation service should set the chapter id has passed'
Chrome Headless 89.0.4389.90 (Linux x86_64): Executed 4212 of 4219 SUCCESS (0 secs / 53.862 secs)
LOG: 'Spec: Story editor navigation service should return the active tab has passed'
Chrome Headless 89.0.4389.90 (Linux x86_64): Executed 4213 of 4219 SUCCESS (0 secs / 53.865 secs)
LOG: 'Spec: Story editor navigation service should navigate to chapter editor with the given id has passed'
Chrome Headless 89.0.4389.90 (Linux x86_64): Executed 4214 of 4219 SUCCESS (0 secs / 53.869 secs)
LOG: 'Spec: Story editor navigation service should return true if url is in story preview has passed'
Chrome Headless 89.0.4389.90 (Linux x86_64): Executed 4215 of 4219 SUCCESS (0 secs / 53.872 secs)
LOG: 'Spec: Story editor navigation service should navigate to chapter editor has passed'
Chrome Headless 89.0.4389.90 (Linux x86_64): Executed 4216 of 4219 SUCCESS (0 secs / 53.876 secs)
LOG: 'Spec: Rule spec services should include evaluation methods for all explicit rules has passed'
Chrome Headless 89.0.4389.90 (Linux x86_64): Executed 4217 of 4219 SUCCESS (0 secs / 53.88 secs)
LOG: 'Spec: MathEquationInputResponse should correctly escape characters in the answer has passed'
Chrome Headless 89.0.4389.90 (Linux x86_64): Executed 4218 of 4219 SUCCESS (0 secs / 53.889 secs)
Chrome Headless 89.0.4389.90 (Linux x86_64): Executed 4219 of 4219 SUCCESS (1 min 37.746 secs / 53.9 secs)
TOTAL: 4219 SUCCESS
TOTAL: 4219 SUCCESS
17 03 2021 03:00:46.198:WARN [launcher]: ChromeHeadless was not killed in 2000 ms, sending SIGKILL.
Done!
```

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



```
.
  Topic and Story viewer functionality
    ? should play through story and save progress on login.
.     ? should check for topic description, stories and revision cards




2 specs, 0 failures
Finished in 622.565 seconds

Executed 2 of 2 specs SUCCESS in 10 mins 23 secs.
[03:23:31] I/launcher - 0 instance(s) of WebDriver still running
[03:23:31] I/launcher - chrome #01 passed
```

Other summer obligations

I have no other commitments during the summer. But I may have to go to my college campus a bit early, as mentioned above.

Communication channels

I would be communicating with my mentor through the common preferred mode of communication (Hangouts, Email, WhatsApp), daily or on a weekly basis, with quick updates and questions through the communication channel (Hangouts, WhatsApp etc.).

Application to multiple orgs

I would be applying only in Oppia.

Project Details

Product Design

Why do we need to test our code?

Providing education to underserved children in an interactive way is Oppia's goal. Anyone can create and share interactive learning activities using Oppia's website. These interactive lessons are known as explorations, and currently, Oppia has numerous explorations.

Hence, Oppia has a huge codebase. It is possible that some bug may slip in if we do not test the code properly. Testing the code rigorously ensures that the quality of code is maintained, saving time for developers and providing a better, bug-free experience to our users (i.e the students).

Unit tests are a must in any codebase because:

- Unit tests help us **validate** our work. We might think the code that we have written is correct, but it often happens that we have an unknown bug in the code, which can be caught by unit tests.
- For unit testing, we must break down our code into small parts. This helps to make the code **modular**, readable and understandable.
- Unit tests can also be seen as **code-documentation**, as they can help someone how a particular piece of code works.
- Sometimes it may happen that a bug slips into production, and this introduces regression which could take a large amount of time to get fixed. Unit tests help us to catch bugs at an early stage and **prevent regressions**.
- Unit tests can help us **refactor** codes without worrying about breaking anything.

What is a unit test?

Consider you are building a car, and you have to take the car for a test ride each time you want to check if the car seats are correctly fitted? That would be very cumbersome and would result in wastage of time and resources.

That is why the method of unit testing exists. In literal terms, unit testing means testing the smallest unit of code. It can be a function, a class, or even an if-statement.

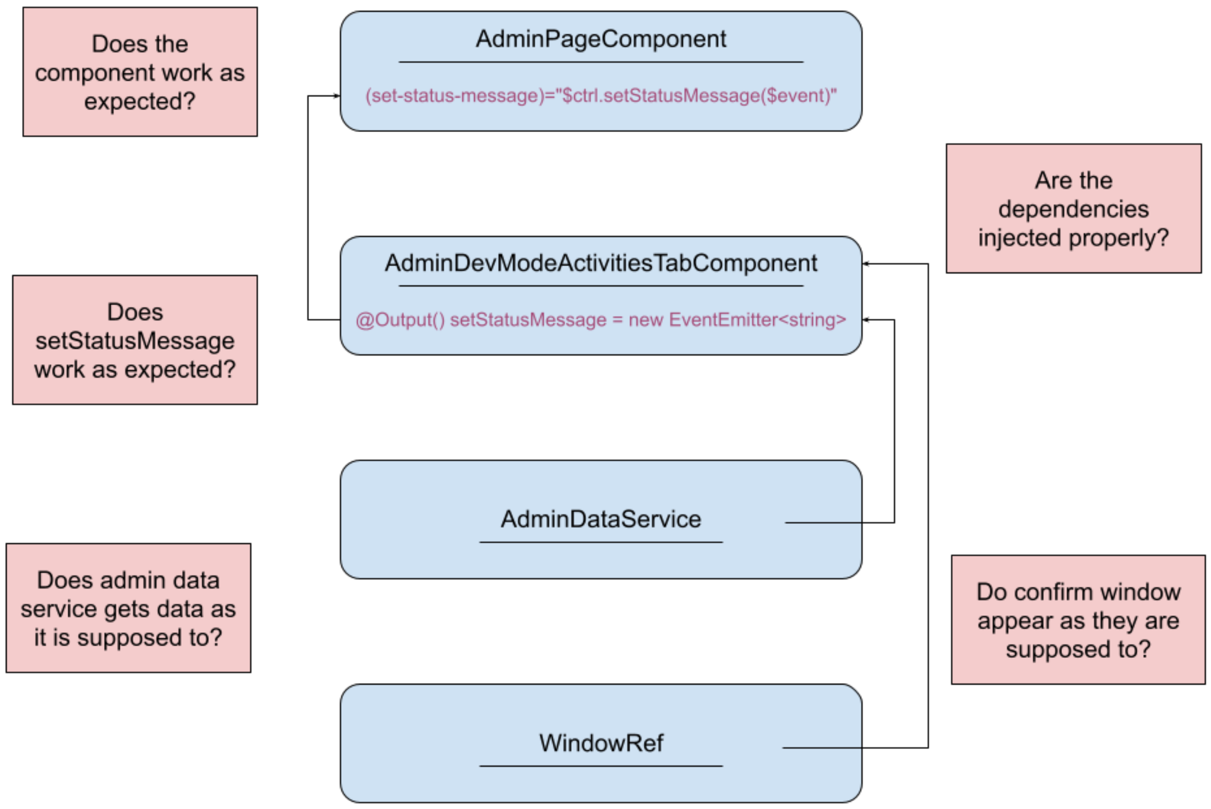

Does the component work as expected?
AdminPageComponent
(set-status-message)="$ctrl.setStatusMessage($event)"
Are the dependencies injected properly?
AdminDevModeActivitiesTabComponent
@Output() setStatusMessage = new EventEmitter<string>
Does setStatusMessage work as expected?
AdminDataService
Does admin data service gets data as it is supposed to?
Do confirm window appear as they are supposed to?
WindowRef

Fig *- Why do we need tests? An example component & some questions that unit tests answer.*

Problem

Oppia’s codebase consists of:

- Services
- Directives
- Components
- Other files

These files are written in both Angular and AngularJS, and the migration of these files is currently going on. In case of an AngularJS directive, we would need to migrate it to a component and then test the code.

Also, not 100% of these files have been covered. Some files do not have a test file, while some are not fully tested.

As of 18th March 2021, this the statistics of frontend test coverage:

File type	Lines	Lines covered	Lines to cover
Component	7680	7280	400
Directive	10854	2657	8197
Service	11689	10296	1393
Other	9828	9028	800
Total	40051	29261	10790
Total (without other files)	30370	20479	9990

**Table was generated using the functions in ‘check_frontend_test_coverage.py’*

Hence, there are **9990** lines to cover.

Goal

At the end of this GSoC project our goal is to achieve:

- 100% coverage of Component, Directive and Services.
- Ensure that all new files added have unit tests along with them.

Work Plan

- Completing unit testing of partially covered files.
- Adding new test files to components/directives/services with no coverage.
- Migrating directives to components:
    - Adding a spec file and writing tests, if the spec file is not present.

Technical Design

Architectural Overview

Files to be tested lie in the following folders:

- /core/templates:
  - /base-components - contains the core non-reusable components
  - /components - contains the reusable components
  - /directive - contains directives not associated with files in /components
  - /domain - contains the files with logic for frontend
  - /expressions -contains files to make expressions work
  - /pages - contains all the pages of the website
  - /services - contains all the services used in the website
- /extensions:
  - /interactions - contains the files which have the interactive component of a card
  - /objects/templates - all objects that are used in the website
  - /rich_text_components - contains components for custom widgets
  - /value_generators/template - contains copier and random selector value generator
  - /visualizations - contains files for statistics display functionality

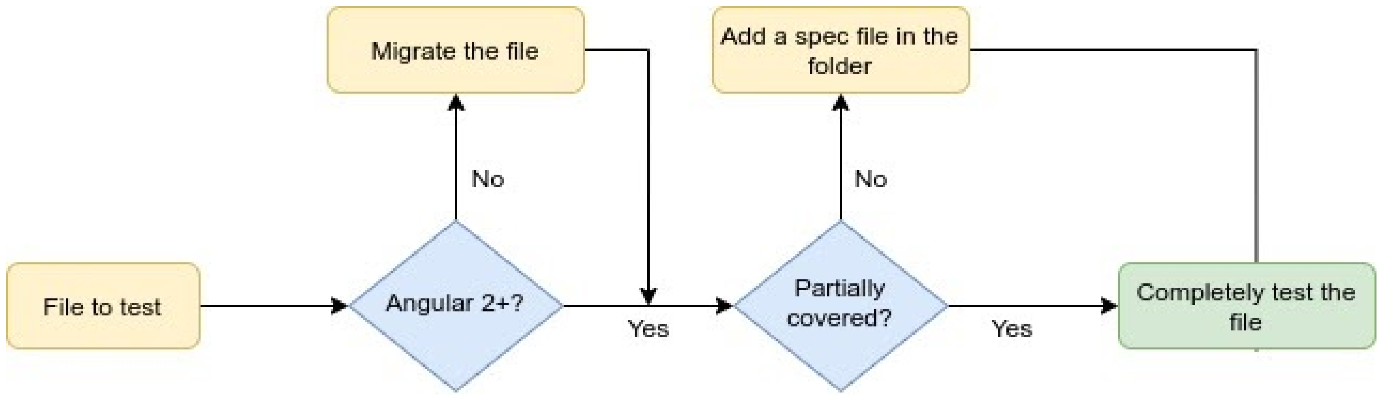
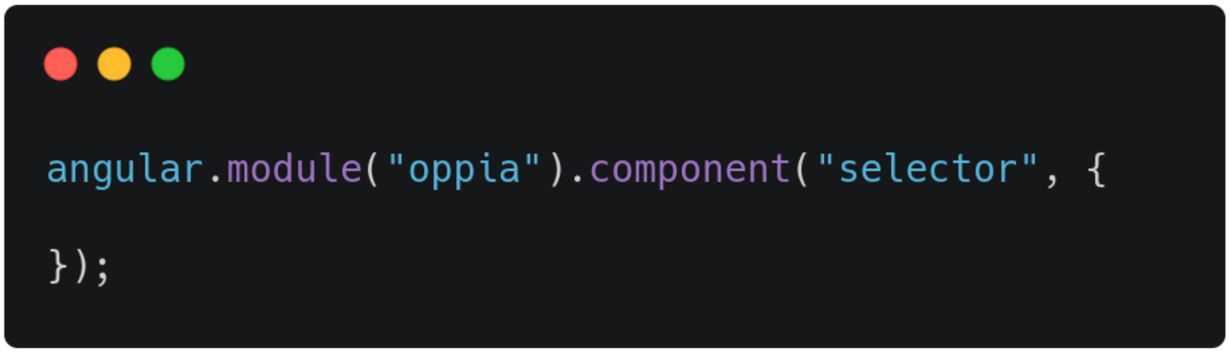


Fig - *Flow to test the files*

***Migrate the file -** Migrating AngularJS Directive to AngularJS Component.
Components in AngularJS are nothing but a special kind of directive, which were introduced in AngularJS 1.5.

Migrating a directive to a component

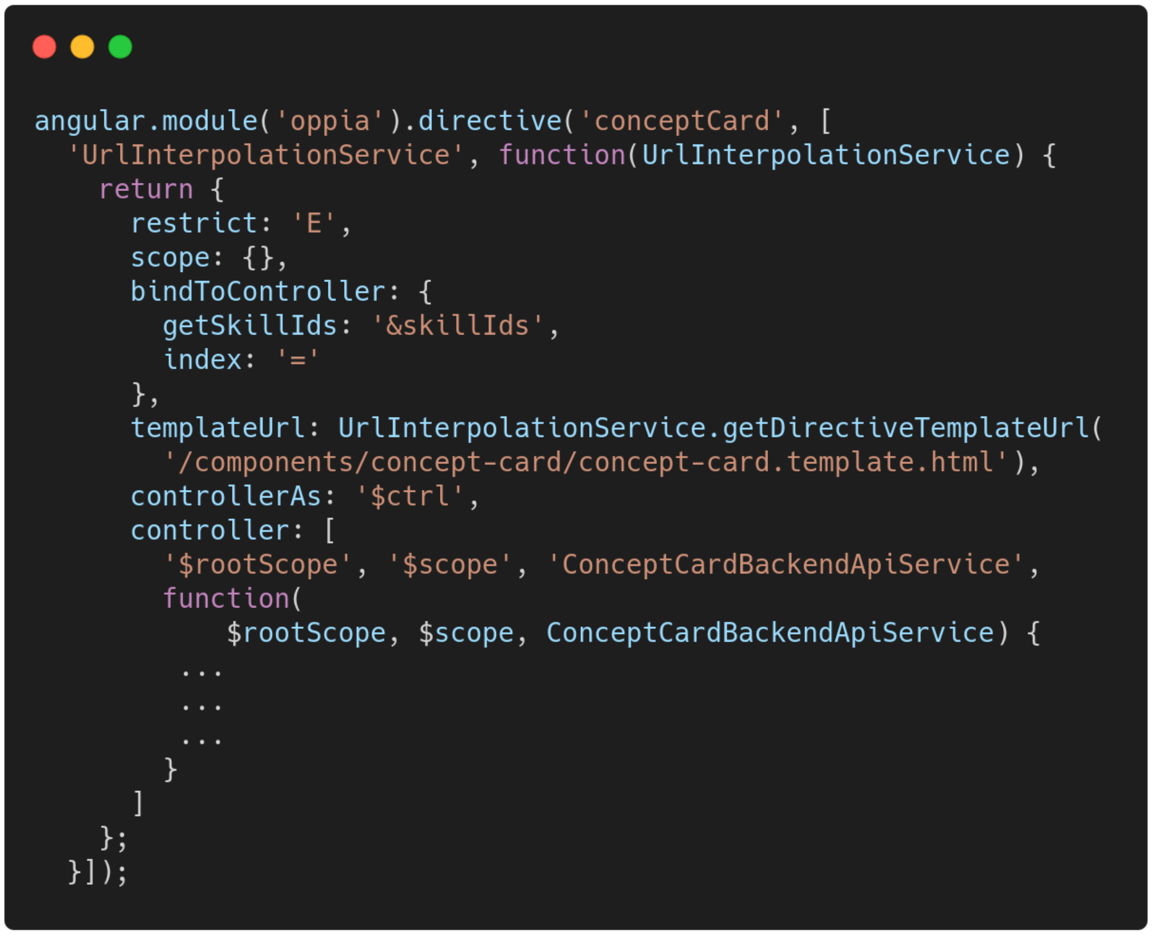
Before writing tests for a directive, we need to migrate it to a component. We can register a component with the following line:



```
angular.module("oppia").component("selector", {

});
```

Fig. Code to register a component in AngularJS

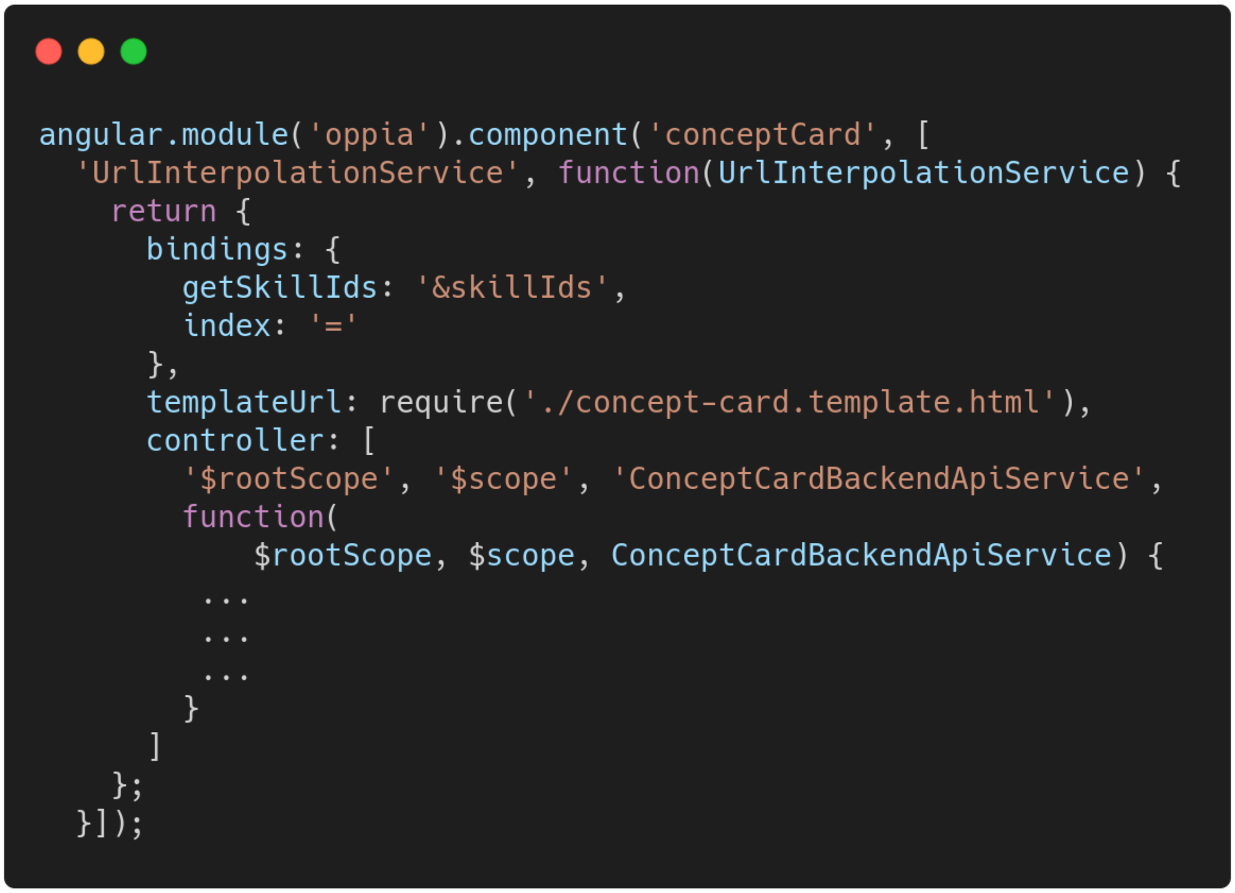
Let's take a look at an example, we have `concept-card.directive.ts`



```
angular.module('oppia').directive('conceptCard', [
  'UrlInterpolationService', function(UrlInterpolationService) {
    return {
      restrict: 'E',
      scope: {},
      bindToController: {
        getSkillIds: '&skillIds',
        index: '='
      },
      templateUrl: UrlInterpolationService.getDirectiveTemplateUrl(
        '/components/concept-card/concept-card.template.html'),
      controllerAs: '$ctrl',
      controller: [
        '$rootScope', '$scope', 'ConceptCardBackendApiService',
        function(
            $rootScope, $scope, ConceptCardBackendApiService) {
         ...
         ...
         ...
        }
      ]
    };
  }]);
```

- First we change the component registration line, as given above.
- Now we change the component definition properties
- Remove the restrict: ‘E’ property, as components are restricted to elements only
- Change bindToController to bindings
- Remove UrlInterpolationService and directly import the template using require()
- We can remove the controllerAs property too, as it’s default value in a component is $ctrl

Hence, the component would look something like this



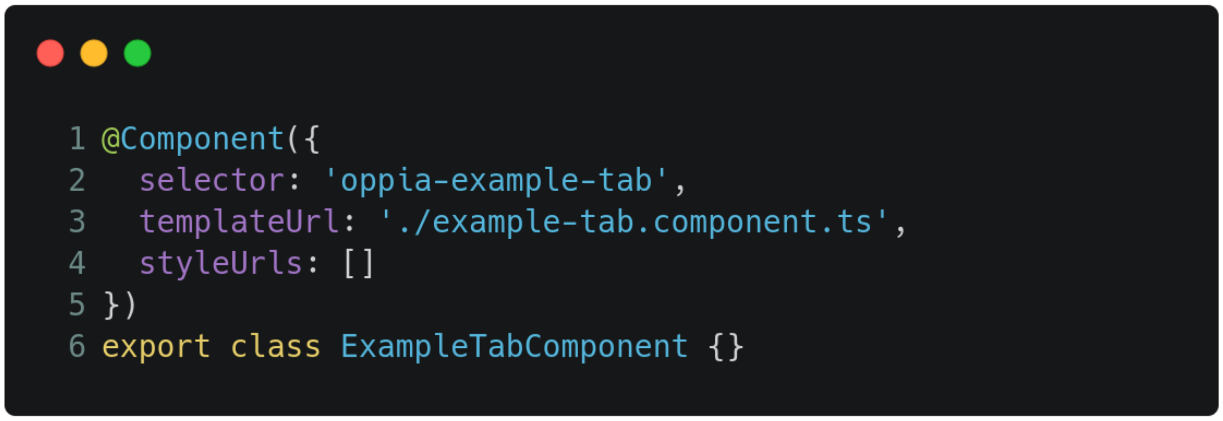
```
angular.module('oppia').component('conceptCard', [
  'UrlInterpolationService', function(UrlInterpolationService) {
    return {
      bindings: {
        getSkillIds: '&skillIds',
        index: '='
      },
      templateUrl: require('./concept-card.template.html'),
      controller: [
        '$rootScope', '$scope', 'ConceptCardBackendApiService',
        function(
            $rootScope, $scope, ConceptCardBackendApiService) {
          ...
          ...
          ...
        }
      ]
    };
  }]);
```

Components

Components are the building block of an Angular 2+ application, consisting of:

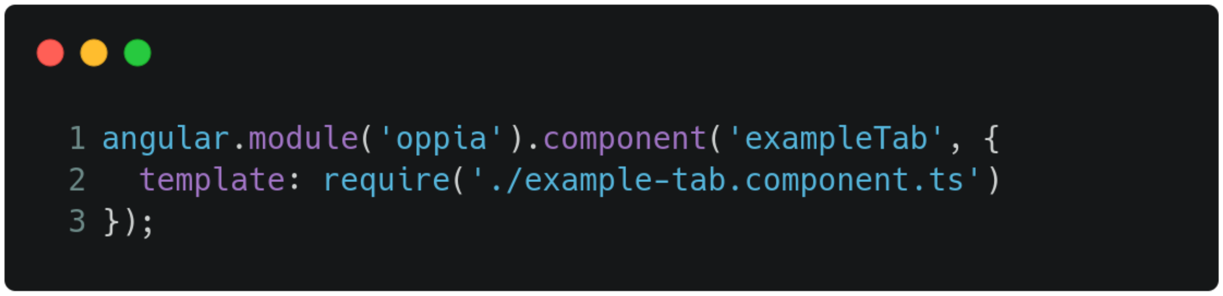
- HTML Template
- Typescript file -- defining the behavior
- CSS selector
- and a CSS file

Currently the oppia codebase consists of **162** components, out of which **138** have 100% coverage, while **24** components do not have 100% coverage. Thus, currently 400 lines of components are to be covered.



```
@Component({
  selector: 'oppia-example-tab',
  templateUrl: './example-tab.component.ts',
  styleUrls: []
})
export class ExampleTabComponent {}
```

Fig. *Starter template of a component in Angular 2+*



```
angular.module('oppia').component('exampleTab', {
  template: require('./example-tab.component.ts')
});
```

Fig. *Starter template of a component in AngularJS*

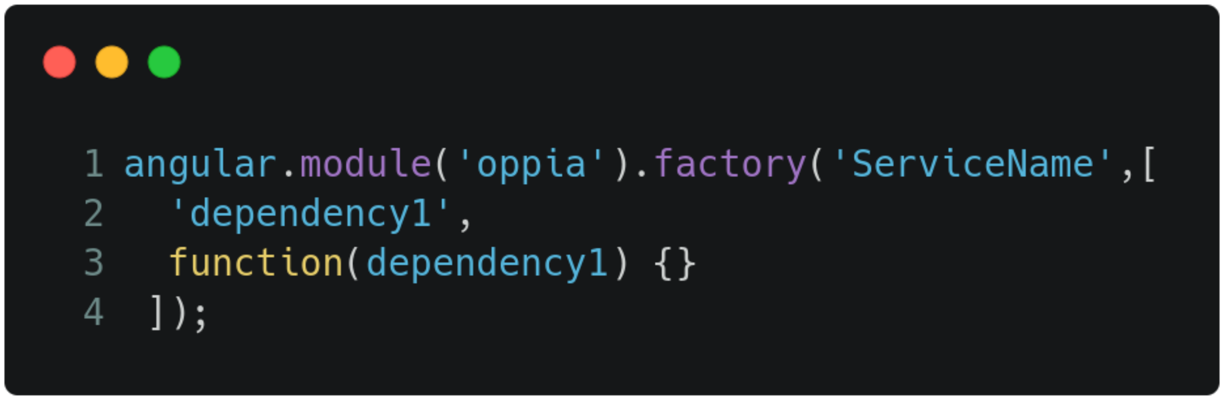
List of components to be tested

Component	Lines to cover
list-of-sets-of-translatable-html-content-ids-editor.component.ts	67
oppia-interactive-fraction-input.component.ts	62
opportunities-list.component.ts	43
subtopic-preview-tab.component.ts	37
contributions-and-review.component.ts	36
sharing-links.component.ts	34
oppia-interactive-continue.component.ts	17
set-of-translatable-html-content-ids-editor.component.ts	15
skill-mastery.component.ts	14
collection-local-nav.component.ts	9
social-buttons.component.ts	8
profile-link-image.component.ts	8
collection-navbar.component.ts	7
translatable-html-content-id.component.ts	7
skill-editor-navbar-breadcrumb.component.ts	6
background-banner.component.ts	5
collection-footer.component.ts	5
oppia-response-fraction-input.component.ts	5
oppia-short-response-fraction-input.component.ts	5
oppia-response-continue.component.ts	3
oppia-short-response-continue.component.ts	3
summary-list-header.component.ts	2
profile-link-text.component.ts	1
topic-editor-page.component.ts	1

Services

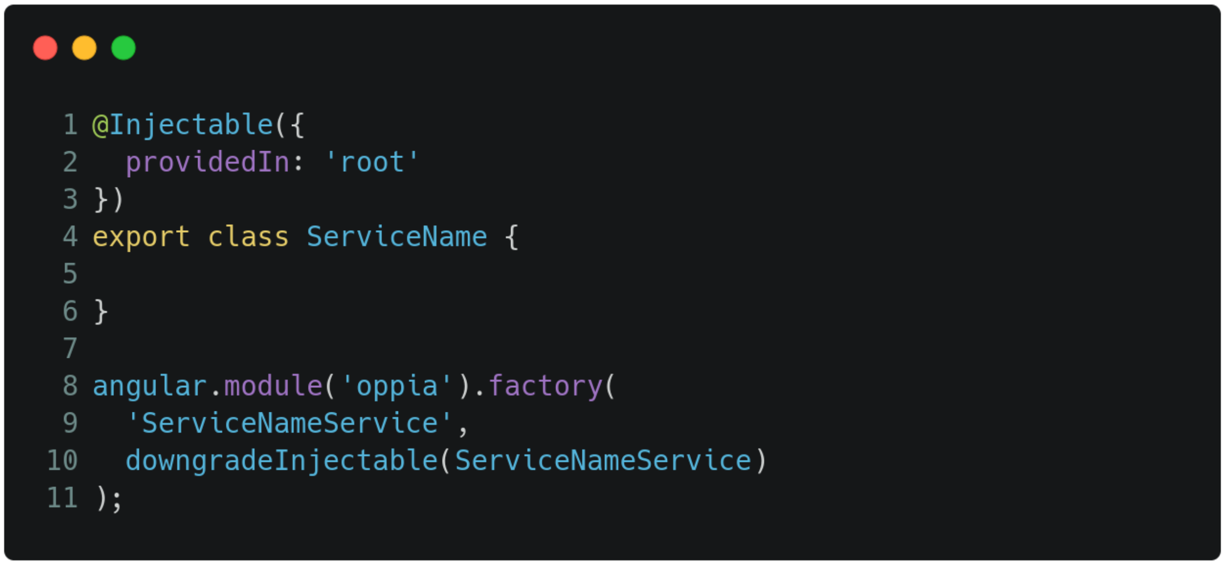
Services are files that can contain a wide variety of things, like a function, value or feature required in the app. Services are a great way to share data among classes that don't know each other. Oppia's codebase contains services that are in AngularJS and Angular 2+.

Currently, Oppia's codebase contains **328** services out of which **72** files are untested and **256** services have been fully tested. Thus, **1393** lines of code of services have to be tested.



```
angular.module('oppia').factory('ServiceName',[
  'dependency1',
  function(dependency1) {}
 ]);
```

Fig - *Factory definition of service in AngularJS*



```
@Injectable({
  providedIn: 'root'
})
export class ServiceName {

}

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

Fig - *Class definition of service in Angular2+ along with downgradeInjectable to make the service usable in AngularJS files*

List of services to be tested

Service	Lines to cover
graph-layout.service.ts	192
exploration-engine.service.ts	149
exploration-save.service.ts	120
stats-reporting.service.ts	92
question-player-engine.service.ts	91
exploration-player-state.service.ts	61
audio-player.service.ts	47
messenger.service.ts	40
story-update.service.ts	35
suggestion-modal-for-exploration-editor.service.ts	30
exploration-states.service.ts	28
topic-editor-state.service.ts	27
topics-and-skills-dashboard-backend-api.service.ts	26
voiceover-recording.service.ts	26
exploration-creation.service.ts	25
item-selection-input-validation.service.ts	24
graph-detail.service.ts	23
contribution-opportunities.service.ts	22
story-creation.service.ts	20
collection-editor-state.service.ts	20
state-editor.service.ts	18
skill-creation.service.ts	17
topic-creation.service.ts	17
editable-story-backend-api.service.ts	16
context.service.ts	16
numeric-input-validation.service.ts	15
change-list.service.ts	14
story-editor-state.service.ts	13
contribution-and-review.service.ts	12
learner-view-rating.service.ts	12
question-update.service.ts	11
autogenerated-audio-player.service.ts	11

request-interceptor.service.ts	11
music-phrase-player.service.ts	7
collection-update.service.ts	6
state-top-answers-stats.service.ts	6
url-interpolation.service.ts	5
audio-preloader.service.ts	5
learner-answer-info.service.ts	5
skill-editor-state.service.ts	5
contribution-opportunities-backend-api.service.ts	4
translate-text.service.ts	4
version-tree.service.ts	4
current-interaction.service.ts	4
player-transcript.service.ts	4
refresher-exploration-confirmation-modal.service.ts	4
state-property.service.ts	3
read-only-collection-backend-api.service.ts	3
email-dashboard-data.service.ts	3
graph-input-validation.service.ts	3
number-with-units-validation.service.ts	3
editable-collection-backend-api.service.ts	2
pretest-question-backend-api.service.ts	2
expression-interpolation.service.ts	2
exploration-diff.service.ts	2
answer-classification.service.ts	2
entity-creation.service.ts	2
csrf-token.service.ts	2
html-escaper.service.ts	2
promo-bar-backend-api.service.ts	2
utils.service.ts	2
svm-prediction.service.ts	2
base-interaction-validation.service.ts	2
code-repl-prediction.service.ts	2
language-util.service.ts	1
expression-evaluator.service.ts	1
fatigue-detection.service.ts	1

player-correctness-feedback-enabled.service.ts	1
translation-file-hash-loader-backend-api.service.ts	1
fraction-input-validation.service.ts	1
graph-input-rules.service.ts	1
music-notes-input-rules.service.ts	1

Directives

Directives are a feature of AngularJS, which act as markers to DOM, that tell the AngularJS compiler to attach some behaviour to the element or transform the element to another form.

Directives can be elements (E), attributes (A), class names (C), and comments (M). Type of directive can be specified using the ‘restrict’ property. Oppia’s codebase contains only element and attribute type of directive, while the un-tested files only have directives with restrict: ‘**E’** i.e element type.

Oppia’s codebase consists of **217** directives out of which only **15** are completely tested, while **202** directives are partially or completely un-tested. Thus, a total of **8197** lines of directives have to be covered.

List of directives to be tested

Directive	Lines to cover
conversation-skin.directive.ts	545
filepath-editor.directive.ts	431
state-responses.directive.ts	293
graph-viz.directive.ts	272
oppia-interactive-music-notes-input.directive.ts	271
questions-list.directive.ts	259
image-with-regions-editor.directive.ts	257
question-player.directive.ts	224
admin-roles-tab.directive.ts	151
collection-player-page.directive.ts	142
topic-editor-navbar.directive.ts	142
top-navigation-bar.directive.ts	130
admin-misc-tab.directive.ts	130
answer-group-editor.directive.ts	123
audio-bar.directive.ts	120
oppia-interactive-logic-proof.directive.ts	101

state-interaction-editor.directive.ts	99
version-diff-visualization.directive.ts	99
question-editor.directive.ts	96
outcome-editor.directive.ts	90
admin-dev-mode-activities-tab.directive.ts	89
oppia-interactive-code-repl.directive.ts	88
ck-editor-4-rte.directive.ts	87
tutor-card.directive.ts	87
schema-based-list-editor.directive.ts	86
rule-editor.directive.ts	86
oppia-interactive-image-click-input.directive.ts	85
state-hints-editor.directive.ts	81
thumbnail-uploader.directive.ts	78
skill-selector.directive.ts	73
rubrics-editor.directive.ts	66
exploration-summary-tile.directive.ts	66
story-summary-tile.directive.ts	66
collection-editor-navbar.directive.ts	66
outcome-destination-editor.directive.ts	65
hint-and-solution-buttons.directive.ts	62
story-editor-navbar.directive.ts	62
collection-node-creator.directive.ts	61
math-expression-content-editor.directive.ts	61
admin-jobs-tab.directive.ts	58
skill-concept-card-editor.directive.ts	58
misconception-editor.directive.ts	58
state-solution-editor.directive.ts	57
progress-nav.directive.ts	54
oppia-interactive-pencil-code-editor.directive.ts	54
feedback-popup.directive.ts	53
image-uploader.directive.ts	49
learner-dashboard-icons.directive.ts	49
skill-editor-navbar.directive.ts	46
state-content-editor.directive.ts	45
collection-details-editor.directive.ts	45
supplemental-card.directive.ts	45

schema-based-unicode-editor.directive.ts	42
state-editor.directive.ts	42
skill-prerequisite-skills-editor.directive.ts	42
oppia-interactive-graph-input.directive.ts	41
oppia-interactive-item-selection-input.directive.ts	40
schema-based-float-editor.directive.ts	39
oppia-interactive-interactive-map.directive.ts	39
oppia-noninteractive-image.directive.ts	39
topic-questions-tab.directive.ts	37
learner-view-info.directive.ts	36
oppia-interactive-multiple-choice-input.directive.ts	34
oppia-interactive-number-with-units.directive.ts	34
admin-config-tab.directive.ts	33
create-activity-button.directive.ts	32
worked-example-editor.directive.ts	32
skill-misconceptions-editor.directive.ts	32
base-content.directive.ts	31
hint-editor.directive.ts	31
normalized-string-editor.directive.ts	31
unicode-string-editor.directive.ts	31
learner-local-nav.directive.ts	29
oppia-interactive-drag-and-drop-sort-input.directive.ts	29
oppia-interactive-end-exploration.directive.ts	29
oppia-interactive-set-input.directive.ts	28
concept-card.directive.ts	27
select2-dropdown.directive.ts	27
solution-explanation-editor.directive.ts	27
input-response-pair.directive.ts	27
skills-mastery-list.directive.ts	26
promo-bar.directive.ts	25
admin-page.directive.ts	25
exploration-footer.directive.ts	25
topic-editor-stories-list.directive.ts	25
response-header.directive.ts	24
rule-type-selector.directive.ts	24

skill-description-editor.directive.ts	24
logic-question-editor.directive.ts	24
audio-file-uploader.directive.ts	23
score-ring.directive.ts	23
fraction-editor.directive.ts	23
role-graph.directive.ts	22
oppia-interactive-text-input.directive.ts	22
skills-list.directive.ts	21
oppia-noninteractive-video.directive.ts	21
collection-summary-tile.directive.ts	20
topic-summary-tile.directive.ts	19
oppia-noninteractive-math.directive.ts	19
review-material-editor.directive.ts	18
music-phrase-editor.directive.ts	18
collection-editor-page.directive.ts	17
number-with-units-editor.directive.ts	17
collection-node-editor.directive.ts	16
coord-two-dim-editor.directive.ts	16
oppia-noninteractive-link.directive.ts	16
codemirror-mergeview.directive.ts	15
skill-rubrics-editor.directive.ts	15
oppia-interactive-numeric-input.directive.ts	15
parameter-name-editor.directive.ts	15
skill-questions-tab.directive.ts	14
story-editor-navbar-breadcrumb.directive.ts	14
drag-and-drop-positive-int-editor.directive.ts	14
skill-selector-editor.directive.ts	14
oppia-noninteractive-skillreview.directive.ts	14
object-editor.directive.ts	12
rating-display.directive.ts	12
learner-answer-info-card.directive.ts	12
code-string-editor.directive.ts	12
subtopic-summary-tile.directive.ts	11
oppia-response-graph-input.directive.ts	11
oppia-response-music-notes-input.directive.ts	11

oppia-short-response-music-notes-input.directive.ts	11
graph-property-editor.directive.ts	11
logic-error-category-editor.directive.ts	11
oppia-root.directive.ts	10
alert-message.directive.ts	10
collection-editor-navbar-breadcrumb.directive.ts	10
oppia-response-drag-and-drop-sort-input.directive.ts	10
oppia-short-response-drag-and-drop-sort-input.directive.ts	10
random-selector.directive.ts	10
side-navigation-bar.directive.ts	9
focus-on.directive.ts	9
story-node-editor.directive.ts	9
topic-editor-navbar-breadcrumb.directive.ts	9
oppia-short-response-interactive-map.directive.ts	9
real-editor.directive.ts	9
schema-based-dict-editor.directive.ts	8
schema-based-int-editor.directive.ts	8
mathjax-bind.directive.ts	8
collection-editor-tab.directive.ts	8
story-editor.directive.ts	8
oppia-response-item-selection-input.directive.ts	8
oppia-short-response-item-selection-input.directive.ts	8
html-select.directive.ts	7
solution-editor.directive.ts	7
oppia-response-code-repl.directive.ts	7
oppia-short-response-multiple-choice-input.directive.ts	7
boolean-editor.directive.ts	7
schema-based-choices-editor.directive.ts	6
question-difficulty-selector.directive.ts	6
oppia-response-multiple-choice-input.directive.ts	6
oppia-response-numeric-input.directive.ts	6
oppia-short-response-numeric-input.directive.ts	6
int-editor.directive.ts	6
list-of-tabs-editor.directive.ts	6
list-of-unicode-string-editor.directive.ts	6

nonnegative-int-editor.directive.ts	6
set-of-unicode-string-editor.directive.ts	6
oppia-response-image-click-input.directive.ts	5
oppia-short-response-image-click-input.directive.ts	5
oppia-response-interactive-map.directive.ts	5
oppia-response-number-with-units.directive.ts	5
oppia-short-response-number-with-units.directive.ts	5
oppia-short-response-set-input.directive.ts	5
subtitled-html-editor.directive.ts	5
oppia-noninteractive-collapsible.directive.ts	5
oppia-visualization-enumerated-frequency-table.directive.ts	5
warnings-and-alerts.directive.ts	4
outcome-feedback-editor.directive.ts	4
oppia-short-response-code-repl.directive.ts	4
oppia-short-response-graph-input.directive.ts	4
oppia-response-logic-proof.directive.ts	4
oppia-short-response-logic-proof.directive.ts	4
oppia-response-pencil-code-editor.directive.ts	4
oppia-short-response-pencil-code-editor.directive.ts	4
oppia-response-set-input.directive.ts	4
oppia-response-text-input.directive.ts	4
oppia-short-response-text-input.directive.ts	4
graph-editor.directive.ts	4
html-editor.directive.ts	4
sanitized-url-editor.directive.ts	4
subtitled-unicode-editor.directive.ts	4
oppia-noninteractive-tabs.directive.ts	4
schema-based-dict-viewer.directive.ts	3
schema-based-primitive-viewer.directive.ts	3
schema-based-unicode-viewer.directive.ts	3
schema-based-bool-editor.directive.ts	1
schema-based-custom-editor.directive.ts	1
schema-based-editor.directive.ts	1
schema-based-expression-editor.directive.ts	1
schema-based-html-editor.directive.ts	1

schema-based-custom-viewer.directive.ts	1
schema-based-html-viewer.directive.ts	1
schema-based-list-viewer.directive.ts	1
schema-based-viewer.directive.ts	1
angular-html-bind.directive.ts	1
correctness-footer.directive.ts	1
continue-button.directive.ts	1
topics-and-skills-dashboard-navbar-breadcrumb.directive.ts	1
oppia-response-end-exploration.directive.ts	1
oppia-short-response-end-exploration.directive.ts	1

Implementation Approach

Difficulty of files

In order to complete the assigned files in the allotted time, a difficulty criteria must be created to determine how much time to devote to file. The files can be divided into 3 categories - Easy, Moderate and Hard based on the number of lines to be covered in that file.

Number of Lines	At Least 250	Upto 250	Upto 100
Difficulty	Hard	Moderate	Easy

Time to test files

Hard	Moderate	Easy
1 file in 3 days	2-3 files in 4 days	1-2 files per day

Now as per the project requirement, 3300 lines of code is to be tested in a period of 10 weeks, which means 330 lines of code in a week to test, which I think would be easily covered according to the time period given in the above table, considering the time to get PRs reviewed and merged and tackling flakes, if any.

Testing Approach

Understanding the working of a file

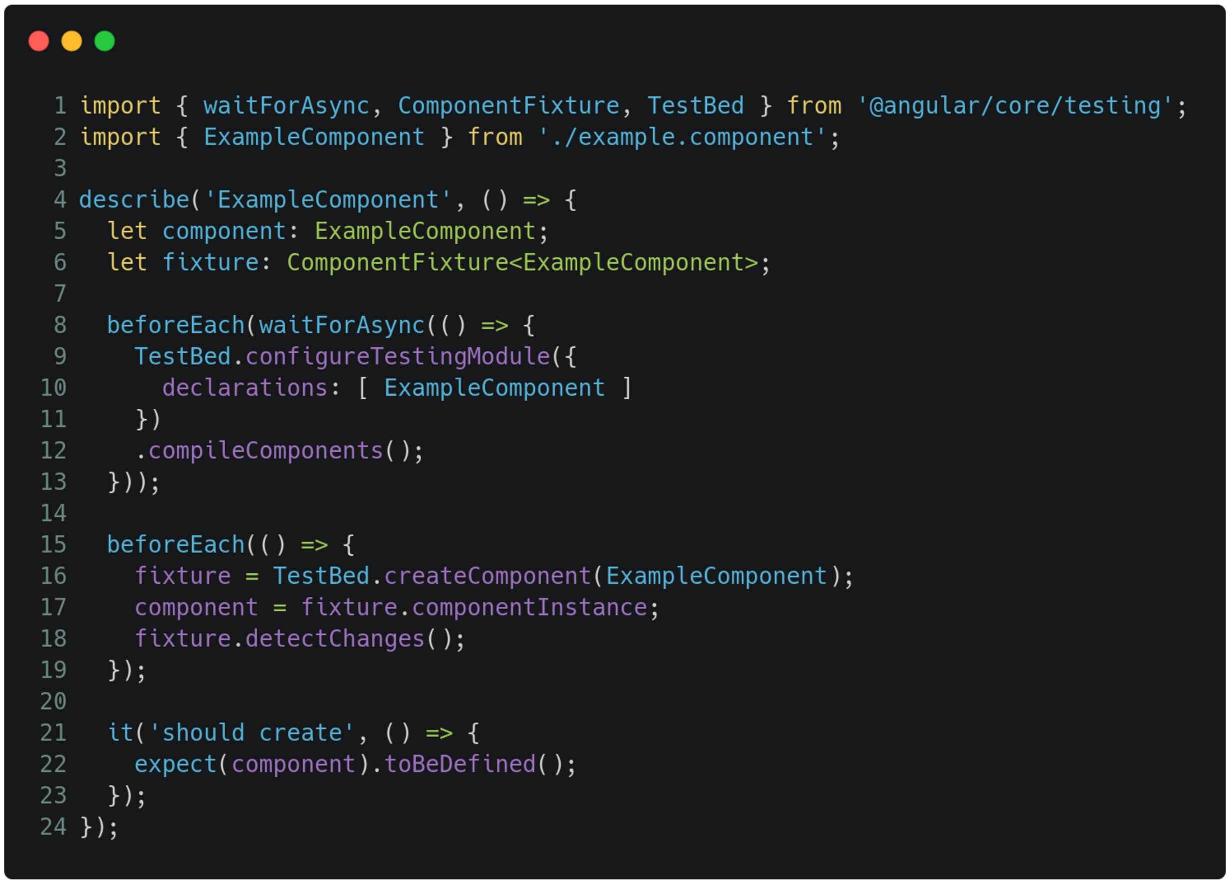
To write tests for a file, we must have a clear understanding of what is going on in the codebase. Now, when the source code is written by someone else, it gets a little hard to write tests for that file. Before beginning with the tests, we must understand what the code does, how it works, what files it is dependent upon, how the file is communicating with other files, are there any HTTP requests made by the code, does the code have any promises etc.

- Begin with starting the local server and trying out the page/component we are going to test. Oppia's 'How to access Oppia webpages' wiki is quite useful for this.
- To know where the file is being used, do a global search with the selector name. For example, for a file named `collection-editor-tab.directive.ts`, search for `<collection-editor-tab>` which would tell us where the file is being used, and then we can access the page.
- Now, a component or directive starts by initializing the data in it's init lifecycle hook. Hence, the function to understand is the init function.
- Follow the calls made inside the init block and get an idea of what those functions are doing.
- Now there would be some functions which are not called anywhere in the file. Do a global search of the function name and see where they are being called. This would give us an idea of how this file is used by other files. For service, a global search along with the service name tells us where and how the service is used. Example, for `admin-data.service.ts` we can do a global search of `adminDataService.getDataAsync(` to know where the `getDataAsync()` function is being called.
- Return type of functions and arguments also gives an insight to what the function is doing. Usually, the type names are self explanatory about the data they contain.
- Log data to the console to get a better understanding of the file.

Components

After understanding the working of a component with the above steps, we move on to writing the tests.

- If a spec file does not exist already, we will create one in the same folder. Name of the spec file should be the same as the component, but with `.spec.ts` ending.
- Oppia's codebase currently does not enforce DOM testing, so we can implement class testing.
- We begin with the starting boilerplate



```
import { waitForAsync, ComponentFixture, TestBed } from '@angular/core/testing';
import { ExampleComponent } from './example.component';

describe('ExampleComponent', () => {
  let component: ExampleComponent;
  let fixture: ComponentFixture<ExampleComponent>;

  beforeEach(waitForAsync(() => {
    TestBed.configureTestingModule({
      declarations: [ ExampleComponent ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(ExampleComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeDefined();
  });
});
```

Fig *- Boilerplate to start component testing*

- Then we can import all dependencies and inject them using TestBed.inject()
- **TestBed** - it is used to configure and initialize the testing environment. We can inject services and other dependencies using TestBed. It acts like a dummy Angular module.
- **createComponent()** - after configuring TestBed, we use createComponent() to create an instance of the component, which we can use in our tests to use the methods and properties of the component.
- **ComponentFixture** - it is a test harness used to debug and test a component
- **beforeEach()** - rather than adding the same code to each unit test, we can add it to the before each block, which runs before every test. Similarly, there exists a afterEach() block which runs after each test. We can use the afterEach() block to clear any data we want. There are other Jasmine hooks like beforeAll, afterAll which are also used in the codebase.
- **expect()** - this block is used for assertions. It has various matchers like toBe(), toEqual(), toBeDefined() etc., which can be chained along to assert a line of code.

After setting up the environment, we can continue with the class testing approach.

Services

Services are the easiest to test. Services can be tested using the class testing approach. TestBed can be used for dependency injection. For asynchronous functions in services, we can use `HttpClientTestingModule` and `HttpTestingController.`

Directives

After migrating a directive to a component we can start testing it. First we need to set up the testing environment. We start with dependency injection.

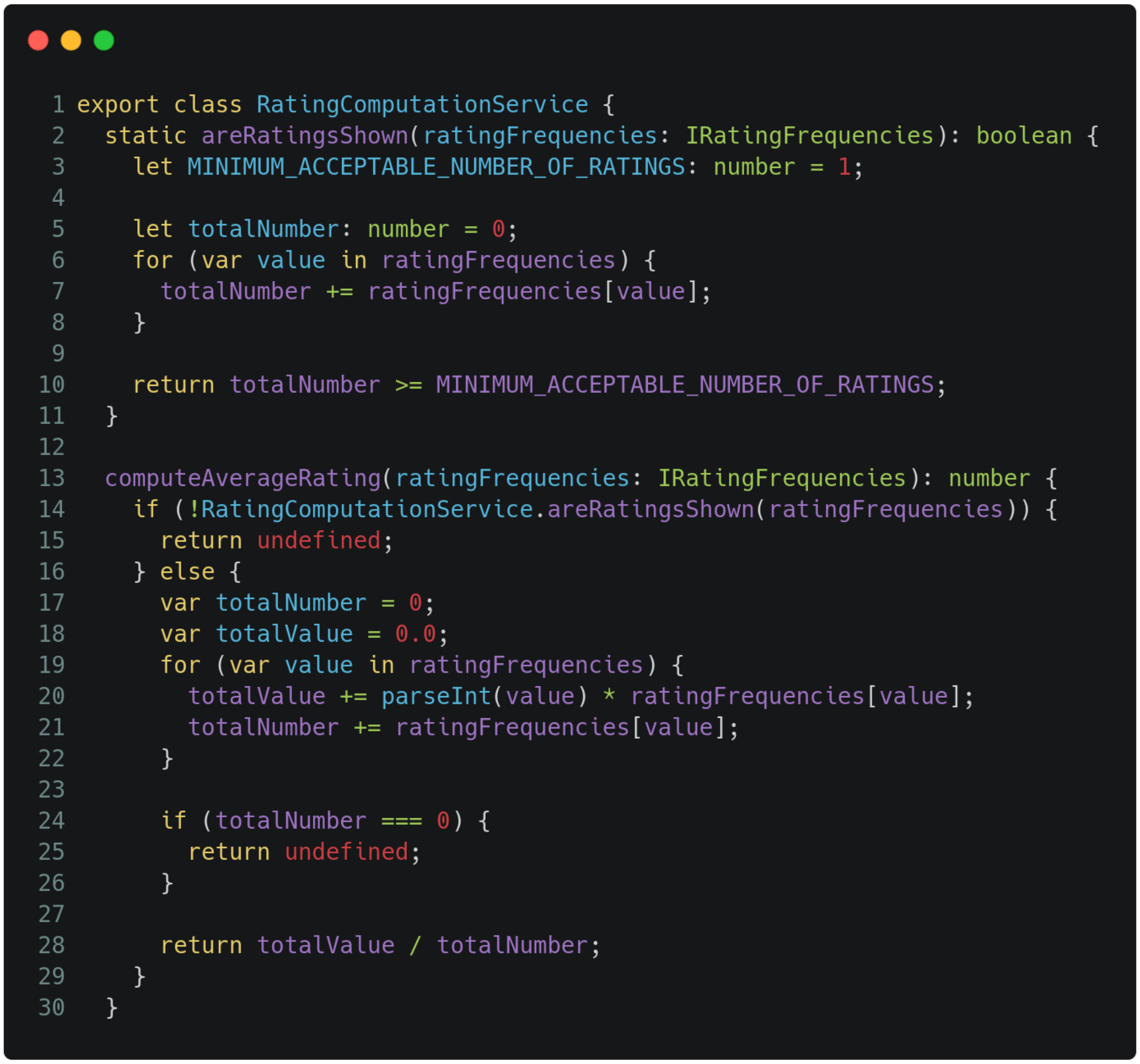
We use `angular.mock.inject( .. )` in a beforeEach block to inject our dependencies and configure spies. If we need to test async code we need to add `HttpClienttestingModule` into the imports inside a beforeEach block. To import upgraded services we can use `importAllAngularServices()`.

After setting up the environment, we test the component file.

Unreachable code

Sometimes, it may so happen that some part of the code is unreachable. There can be two such possibilities, first that the code is inside a block whose condition would never be true because of the logic, and second when the condition would not be true in normal circumstances. Let's see this by an example:

- In the following code inside the `if block` at line 24 was unreachable. Hence the coverage was not 100%.

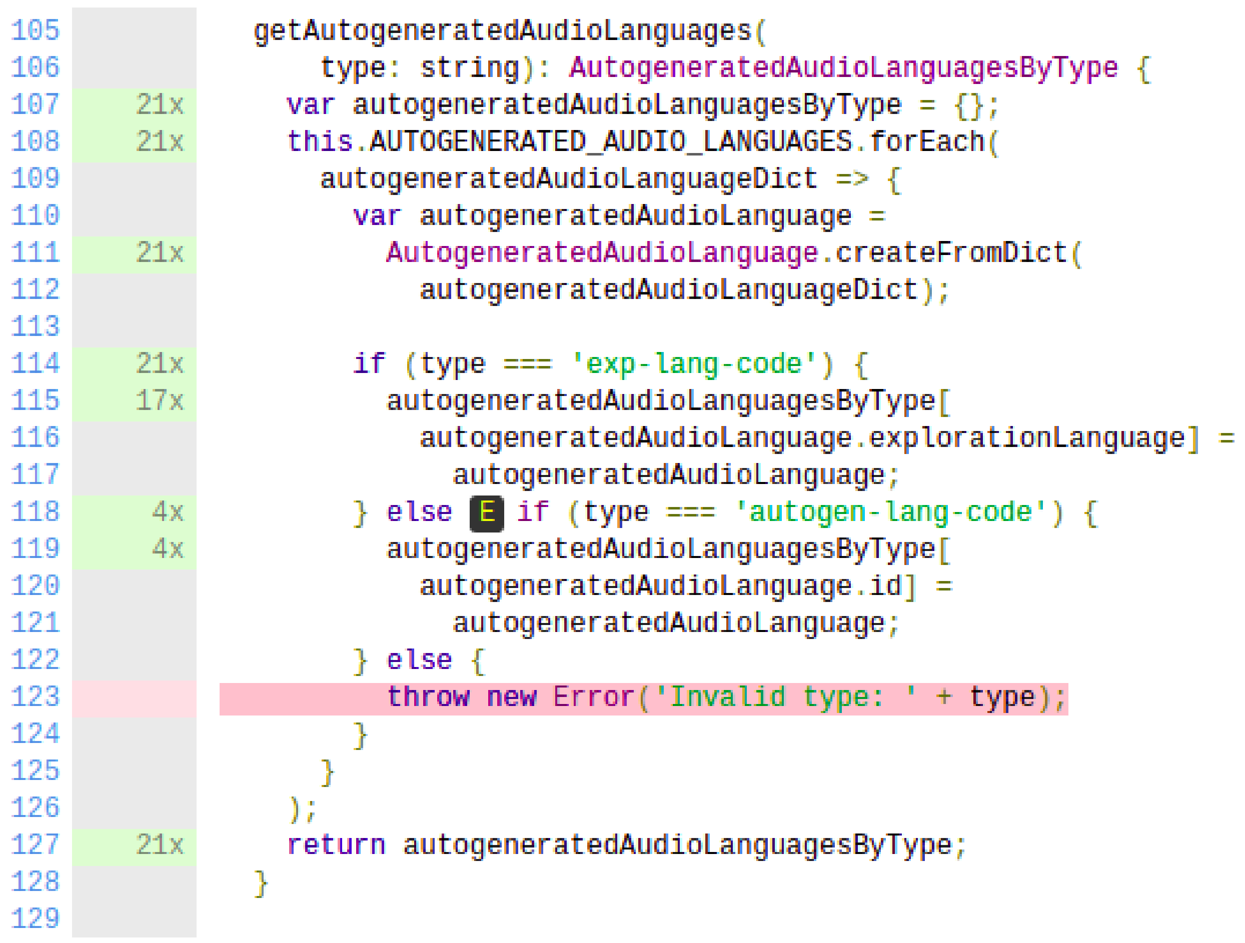


```
 1 export class RatingComputationService {
 2   static areRatingsShown(ratingFrequencies: IRatingFrequencies): boolean {
 3     let MINIMUM_ACCEPTABLE_NUMBER_OF_RATINGS: number = 1;
 4
 5     let totalNumber: number = 0;
 6     for (var value in ratingFrequencies) {
 7       totalNumber += ratingFrequencies[value];
 8     }
 9
10     return totalNumber >= MINIMUM_ACCEPTABLE_NUMBER_OF_RATINGS;
11   }
12
13   computeAverageRating(ratingFrequencies: IRatingFrequencies): number {
14     if (!RatingComputationService.areRatingsShown(ratingFrequencies)) {
15       return undefined;
16     } else {
17       var totalNumber = 0;
18       var totalValue = 0.0;
19       for (var value in ratingFrequencies) {
20         totalValue += parseInt(value) * ratingFrequencies[value];
21         totalNumber += ratingFrequencies[value];
22       }
23
24       if (totalNumber === 0) {
25         return undefined;
26       }
27
28       return totalValue / totalNumber;
29     }
30   }
```

- Now if we try to understand the code, the if block is inside an else block which is only executed when RatingComputationService.areRatingsShown(ratingFrequencies) is true.
- Looking at the areRatingsShown() function, we can see that it only returns true when, totalNumber >= MINIMUM_ACCEPTABLE_NUMBER_OF_RATING (which is always 1).
- Hence when the control reaches the else block, it means totalNumber is >= 1. So, the if condition at line 24 will never be true, and the code is unreachable.
- To solve the coverage problem, we can remove(delete) this block of code, which makes the coverage 100%.
- Hence, there may be cases where a little debugging would tell us why the code is unreachable and how we can improve the code.

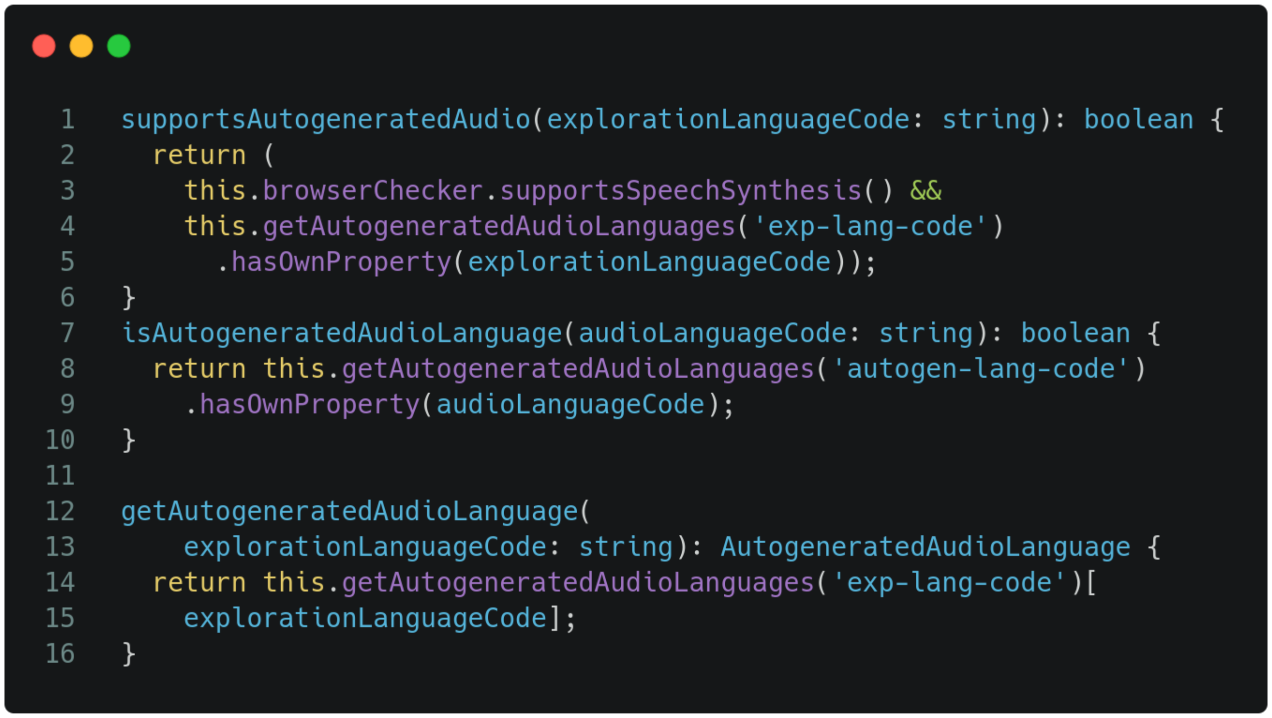
Now for the second case, let’s take an example:

- In some services, there are functions A which are called by other functions B, and we test function B. Now what happens is sometimes, a condition in A never gets executed, as it may be placed as a check if something changes in the future, and the code does not break.
- Let us see the example of language-util.service.ts which has 1 untested line:



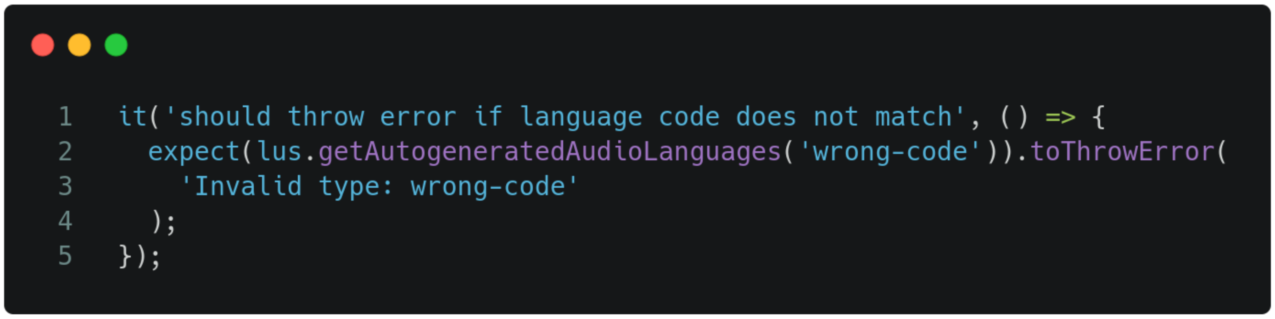
```
getAutogeneratedAudioLanguages(
    type: string): AutogeneratedAudioLanguagesByType {
  var autogeneratedAudioLanguagesByType = {};
  this.AUTOGENERATED_AUDIO_LANGUAGES.forEach(
    autogeneratedAudioLanguageDict => {
      var autogeneratedAudioLanguage =
        AutogeneratedAudioLanguage.createFromDict(
          autogeneratedAudioLanguageDict);

      if (type === 'exp-lang-code') {
        autogeneratedAudioLanguagesByType[
          autogeneratedAudioLanguage.explorationLanguage] =
            autogeneratedAudioLanguage;
      } else E if (type === 'autogen-lang-code') {
        autogeneratedAudioLanguagesByType[
          autogeneratedAudioLanguage.id] =
            autogeneratedAudioLanguage;
      } else {
        throw new Error('Invalid type: ' + type);
      }
    }
  );
  return autogeneratedAudioLanguagesByType;
}
```

- getAutogeneratedLanguages() is called by other functions, and the value of type is hardcoded:



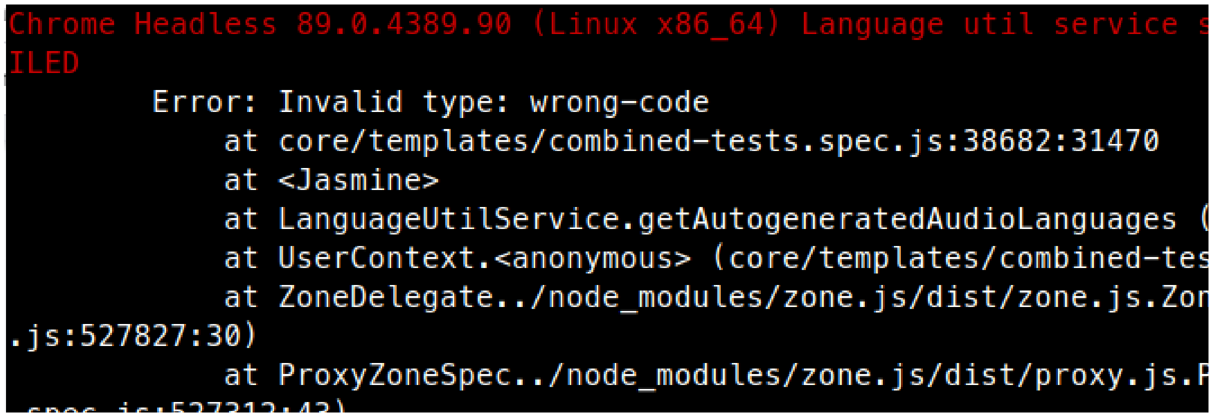
```
supportsAutogeneratedAudio(explorationLanguageCode: string): boolean {
  return (
    this.browserChecker.supportsSpeechSynthesis() &&
    this.getAutogeneratedAudioLanguages('exp-lang-code')
      .hasOwnProperty(explorationLanguageCode));
}
isAutogeneratedAudioLanguage(audioLanguageCode: string): boolean {
  return this.getAutogeneratedAudioLanguages('autogen-lang-code')
    .hasOwnProperty(audioLanguageCode);
}

getAutogeneratedAudioLanguage(
    explorationLanguageCode: string): AutogeneratedAudioLanguage {
  return this.getAutogeneratedAudioLanguages('exp-lang-code')[
    explorationLanguageCode];
}
```

- Now, we can simply test the getAutogeneratedLanguages() function, by calling it in an expect block.



```
it('should throw error if language code does not match', () => {
  expect(lus.getAutogeneratedAudioLanguages('wrong-code')).toThrowError(
    'Invalid type: wrong-code'
  );
});
```

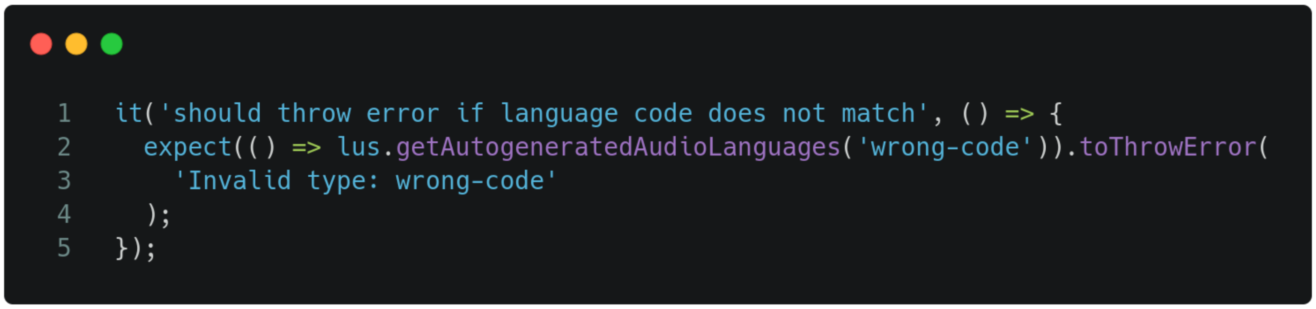
but this gives an error,



```
Chrome Headless 89.0.4389.90 (Linux x86_64) Language util service s
ILED
        Error: Invalid type: wrong-code
            at core/templates/combined-tests.spec.js:38682:31470
            at <Jasmine>
            at LanguageUtilService.getAutogeneratedAudioLanguages (
            at UserContext.<anonymous> (core/templates/combined-tes
            at ZoneDelegate../node_modules/zone.js/dist/zone.js.Zon
.js:527827:30)
            at ProxyZoneSpec../node_modules/zone.js/dist/proxy.js.P
```

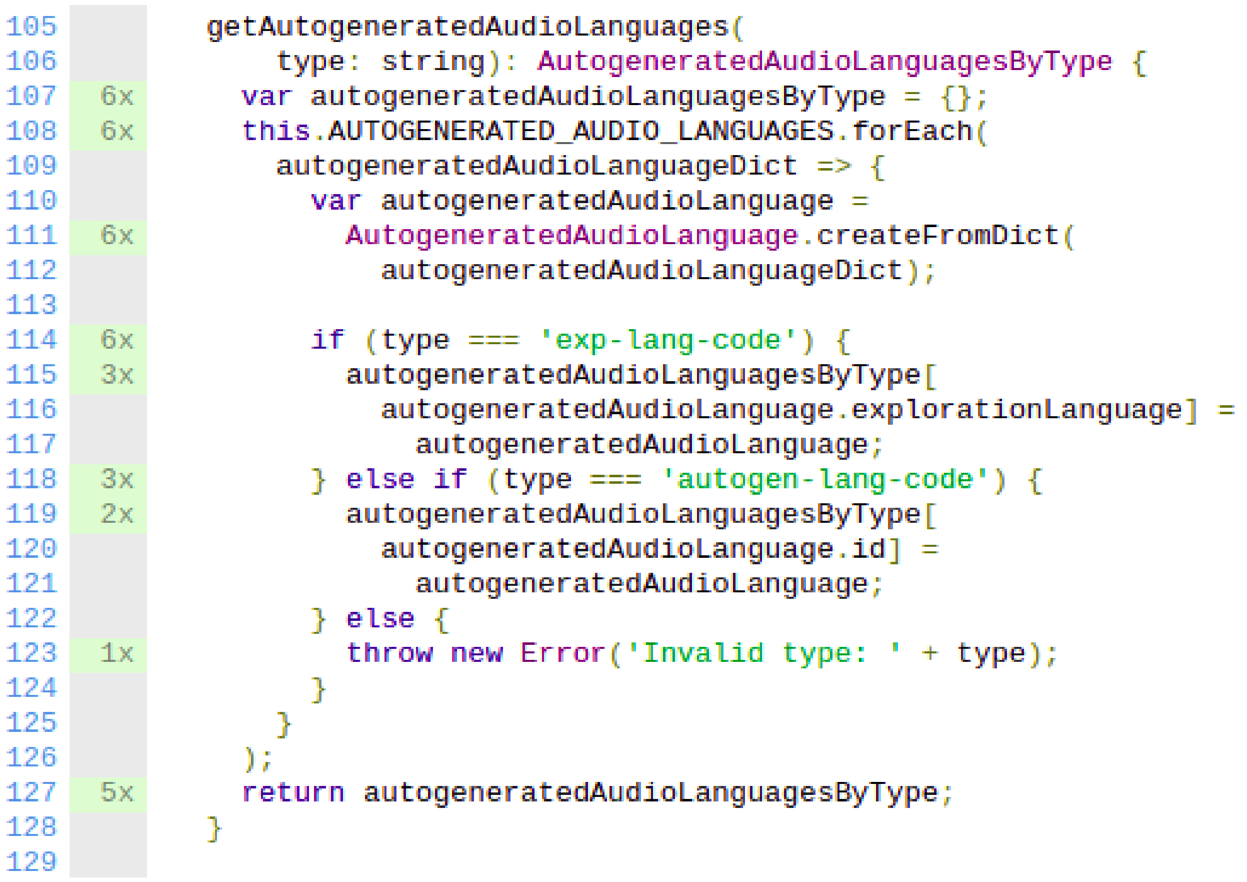
- This is because we are trying to invoke the function, but .toThrow expects a function.

- So the test becomes



```
it('should throw error if language code does not match', () => {
  expect(() => lus.getAutogeneratedAudioLanguages('wrong-code')).toThrowError(
    'Invalid type: wrong-code'
  );
});
```

- Now the test will pass, and the line is covered.



```
        getAutogeneratedAudioLanguages(
            type: string): AutogeneratedAudioLanguagesByType {
 6x       var autogeneratedAudioLanguagesByType = {};
 6x       this.AUTOGENERATED_AUDIO_LANGUAGES.forEach(
            autogeneratedAudioLanguageDict => {
              var autogeneratedAudioLanguage =
 6x             AutogeneratedAudioLanguage.createFromDict(
                  autogeneratedAudioLanguageDict);

 6x           if (type === 'exp-lang-code') {
 3x             autogeneratedAudioLanguagesByType[
                  autogeneratedAudioLanguage.explorationLanguage] =
                    autogeneratedAudioLanguage;
 3x           } else if (type === 'autogen-lang-code') {
 2x             autogeneratedAudioLanguagesByType[
                  autogeneratedAudioLanguage.id] =
                    autogeneratedAudioLanguage;
              } else {
 1x             throw new Error('Invalid type: ' + type);
              }
            }
          );
 5x       return autogeneratedAudioLanguagesByType;
        }
```

- Similarly, there can be cases where we can spy on the function and try to invoke the case which was left untested, or even refactor the code to test the line.

Avoiding flaky tests

When tests are non-deterministic, that is, they can pass or fail for the same code they are called flaky tests. A flaky tests hampers development flow, so it is important to write code in a way to not introduce flaky tests. And if a flaky test goes into the develop branch, it should be identified and corrected as soon as possible.

Reasons for tests to be flaky can be:

- Tests depending on each other
- Not having a clean state setup before each test
- Not handling dynamic content carefully

To avoid writing flaky tests we must follow the AAA of testing. Arrange, Act, Assert.This means that we should divide our tests into three sections.

- In the first section we should set-up our tests, the mocks, spies etc we will be using.
- Next, in the act section we should invoke the methods we want to test.
- And, lastly, in the asset section we should add the ‘expect’ statements, and whether the assertions were true or false.

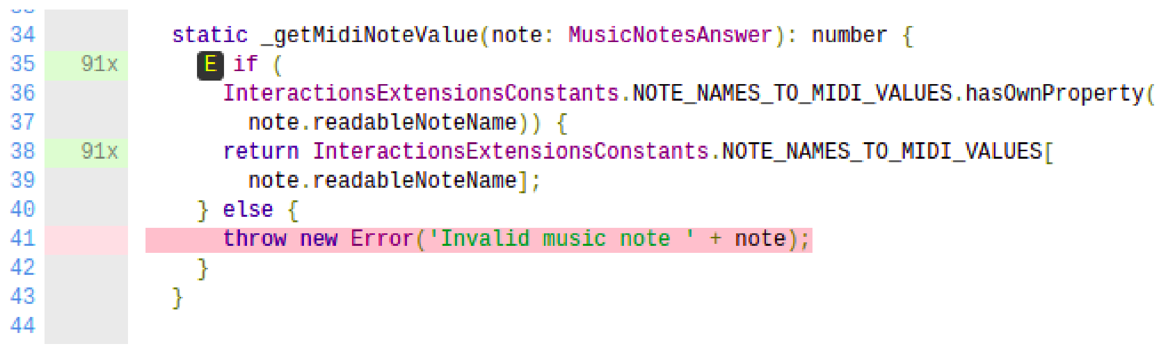
Now, if somehow we introduce a flaky test into the develop branch, we would have to fix them immediately. First we should identify the flaky tests, by running the tests a few times and noting down the flaking tests. Then we begin by fixing the flaking tests one by one.

I am planning to create my milestones with enough buffer time to handle unexpected problems like flaky tests. As soon a flake is reported, an issue would be files for the same and I would be assigned to the issue. Any issue for flaking tests would be given high priority as a flaky test would hamper development flow.

Tricky Scenarios

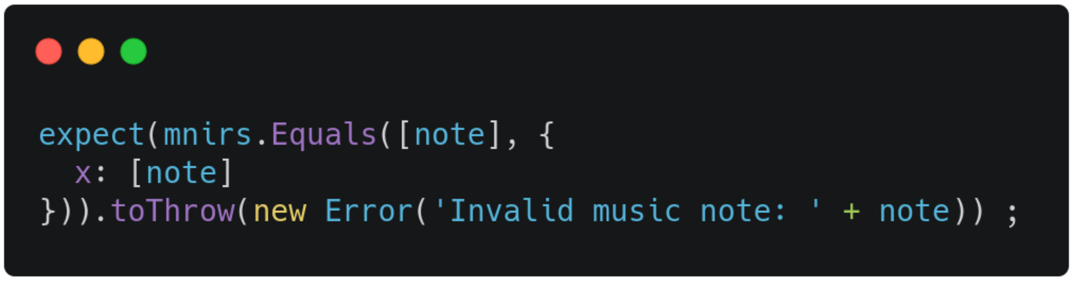
- Testing Error()

  Let’s take music-note-input-rule.service.ts as an example. The service has 1 untested line,



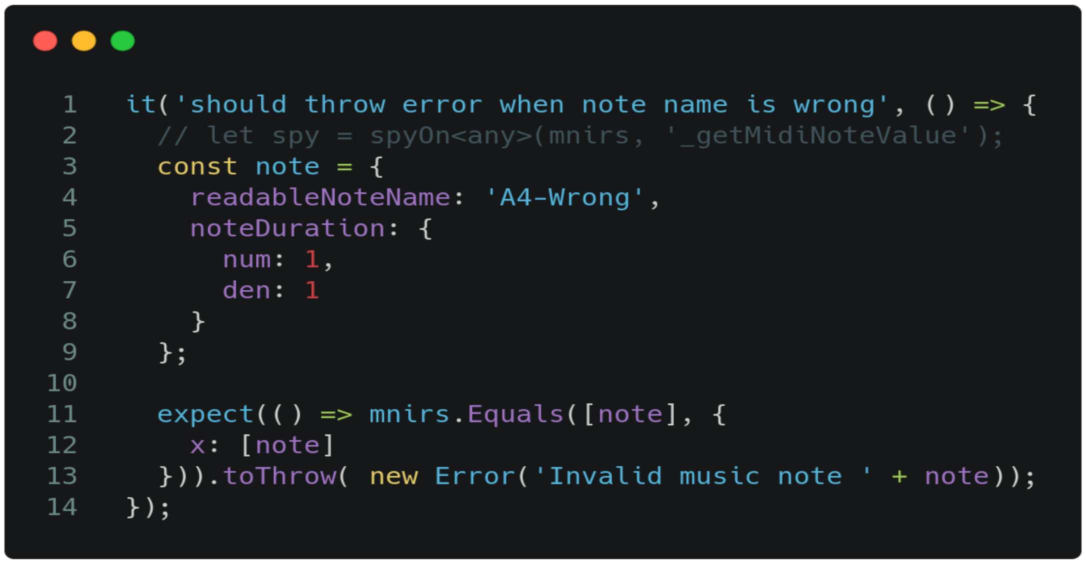
```
  static _getMidiNoteValue(note: MusicNotesAnswer): number {
    if (
      InteractionsExtensionsConstants.NOTE_NAMES_TO_MIDI_VALUES.hasOwnProperty(
        note.readableNoteName)) {
      return InteractionsExtensionsConstants.NOTE_NAMES_TO_MIDI_VALUES[
        note.readableNoteName];
    } else {
      throw new Error('Invalid music note ' + note);
    }
  }
```

Testing an error is tricky because, if we directly invoke the method and expect it to throw an error, then the test will fail. Instead, we should wrap the return value inside a function.



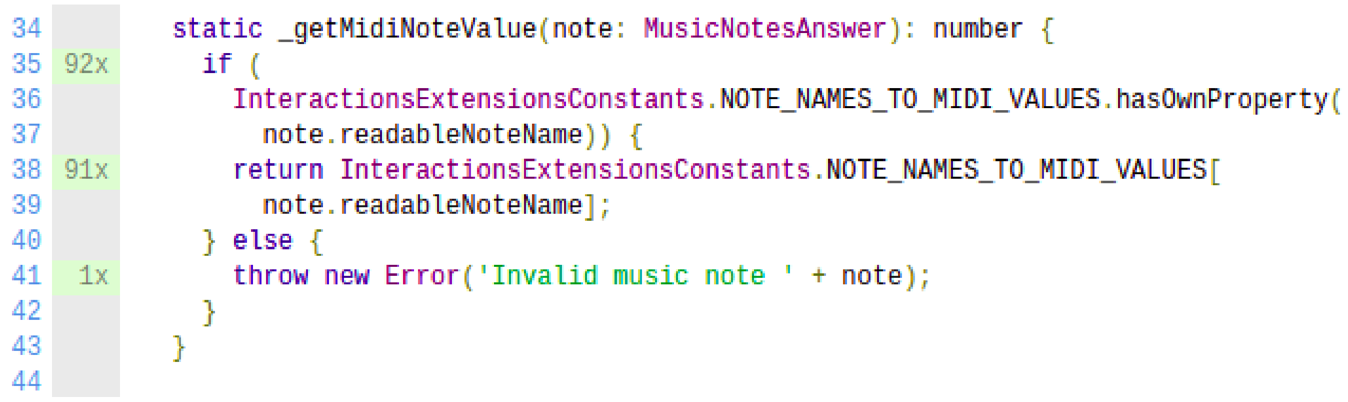
```
expect(mnirs.Equals([note], {
  x: [note]
})).toThrow(new Error('Invalid music note: ' + note)) ;
```

Would throw an error, and the test would fail; But the code below would test the line.



```
it('should throw error when note name is wrong', () => {
  // let spy = spyOn<any>(mnirs, '_getMidiNoteValue');
  const note = {
    readableNoteName: 'A4-Wrong',
    noteDuration: {
      num: 1,
      den: 1
    }
  };

  expect(() => mnirs.Equals([note], {
    x: [note]
  })).toThrow( new Error('Invalid music note ' + note));
});
```

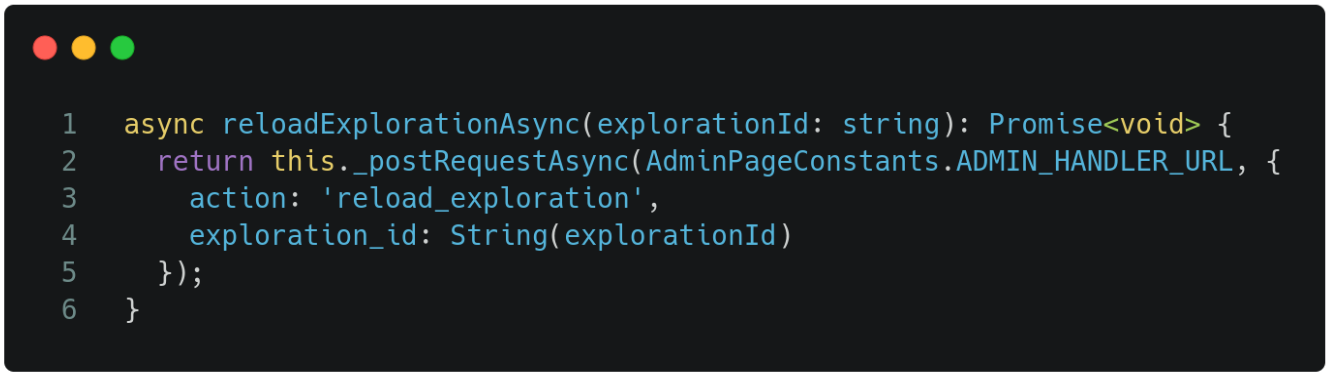
Now, the line is covered.



```
34        static _getMidiNoteValue(note: MusicNotesAnswer): number {
35  92x     if (
36            InteractionsExtensionsConstants.NOTE_NAMES_TO_MIDI_VALUES.hasOwnProperty(
37              note.readableNoteName)) {
38  91x       return InteractionsExtensionsConstants.NOTE_NAMES_TO_MIDI_VALUES[
39              note.readableNoteName];
40          } else {
41   1x       throw new Error('Invalid music note ' + note);
42          }
43        }
44
```

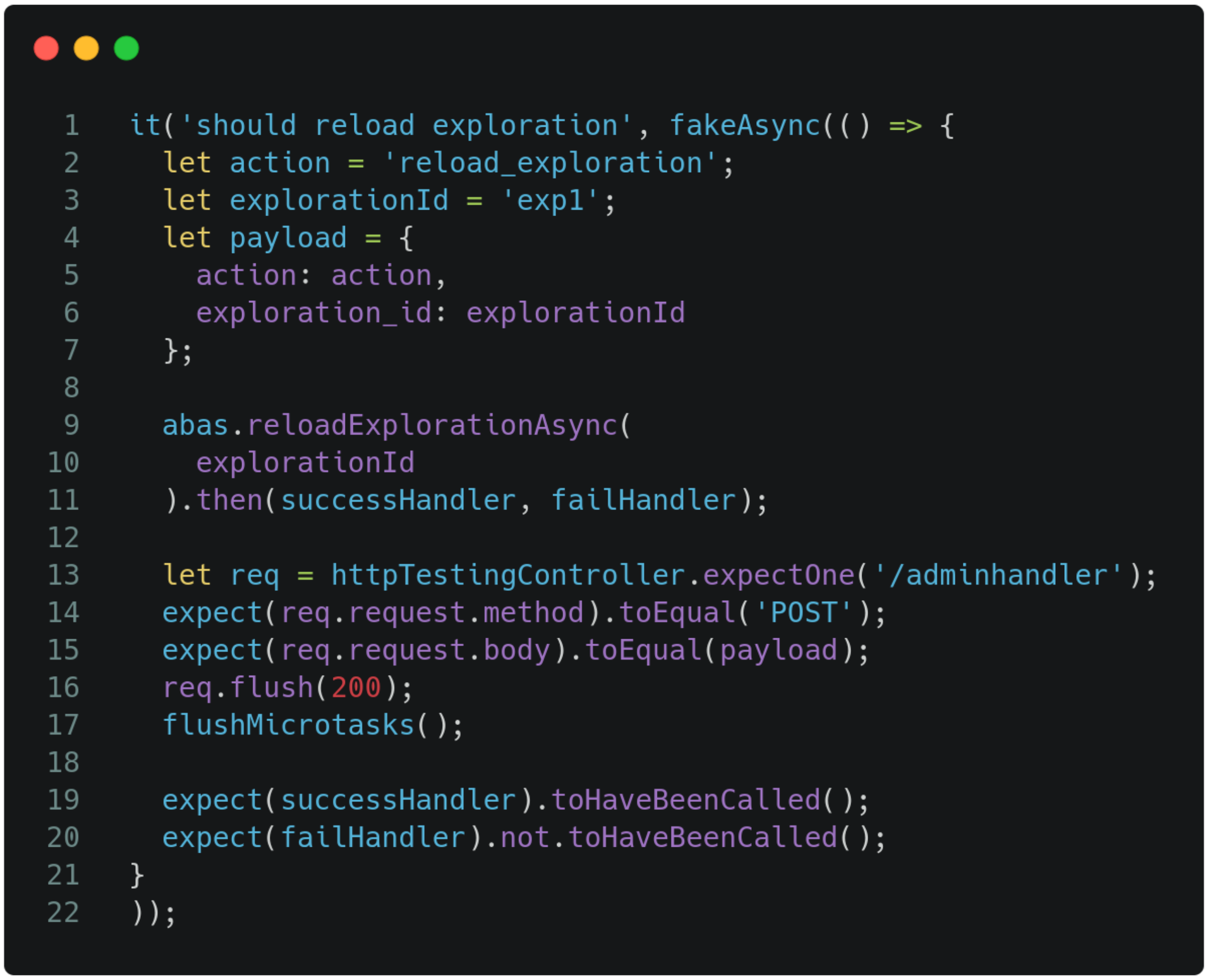
- Testing HTTP

  Suppose we need to test the following code:



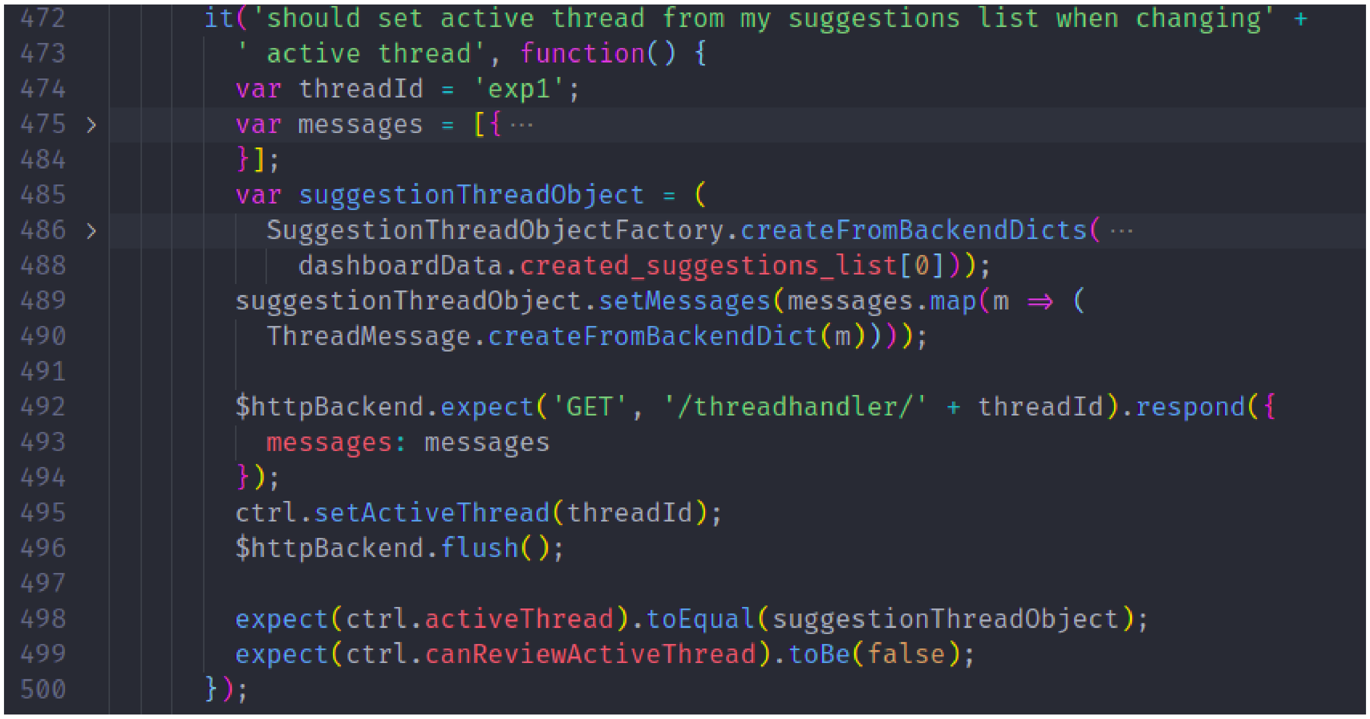
```
async reloadExplorationAsync(explorationId: string): Promise<void> {
  return this._postRequestAsync(AdminPageConstants.ADMIN_HANDLER_URL, {
    action: 'reload_exploration',
    exploration_id: String(explorationId)
  });
}
```

  We can do so by using fakeAsync, HttpTestingController and flushmicrotasks()



```
it('should reload exploration', fakeAsync(() => {
  let action = 'reload_exploration';
  let explorationId = 'exp1';
  let payload = {
    action: action,
    exploration_id: explorationId
  };

  abas.reloadExplorationAsync(
    explorationId
  ).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();
}
));
```

- In AngularJS, for HTTP tests we can use $httpBackend.
  For example, in creator-dashboard-page.component.spec.ts we have,

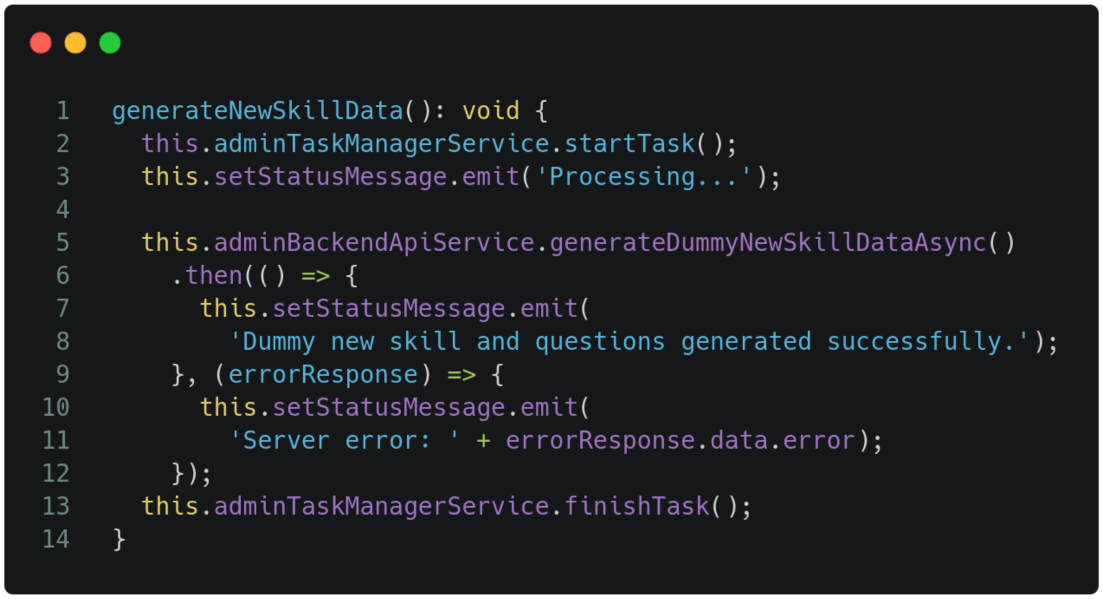


```
  it('should set active thread from my suggestions list when changing' +
    ' active thread', function() {
    var threadId = 'exp1';
    var messages = [{ ⋯
    }];
    var suggestionThreadObject = (
      SuggestionThreadObjectFactory.createFromBackendDicts( ⋯
        dashboardData.created_suggestions_list[0]));
    suggestionThreadObject.setMessages(messages.map(m => (
      ThreadMessage.createFromBackendDict(m))));

    $httpBackend.expect('GET', '/threadhandler/' + threadId).respond({
      messages: messages
    });
    ctrl.setActiveThread(threadId);
    $httpBackend.flush();

    expect(ctrl.activeThread).toEqual(suggestionThreadObject);
    expect(ctrl.canReviewActiveThread).toBe(false);
  });
```

$httpBackend() will ensure that the mock request call will be executed.

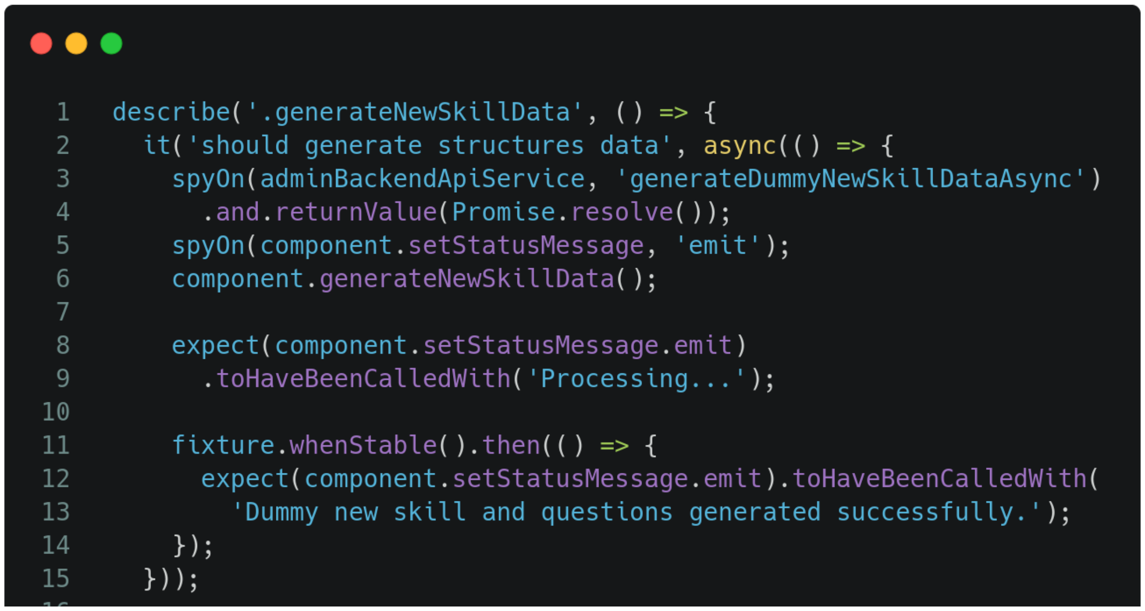
- Testing Promises

  There are many ways to test a promise, one of them is to use whenStable, suppose we need to test this code:



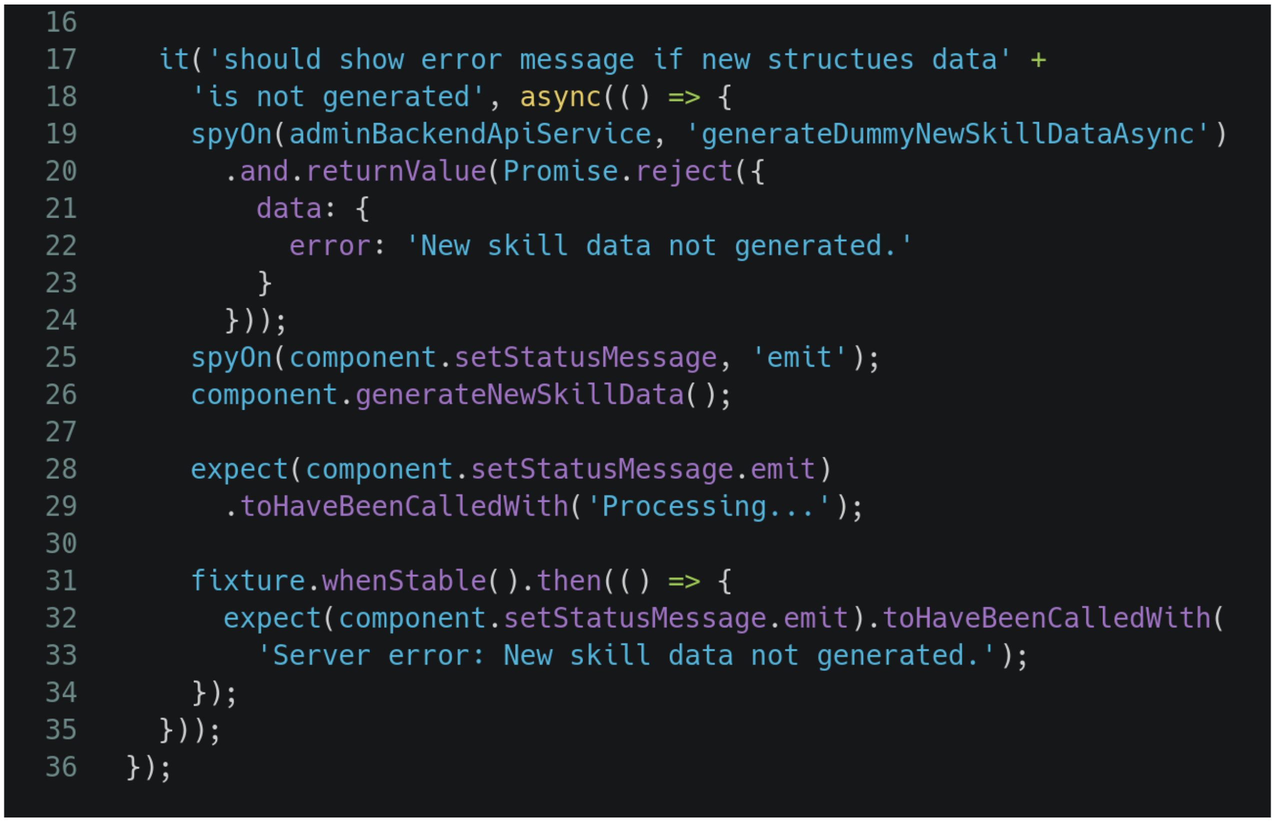
```
generateNewSkillData(): void {
  this.adminTaskManagerService.startTask();
  this.setStatusMessage.emit('Processing...');

  this.adminBackendApiService.generateDummyNewSkillDataAsync()
    .then(() => {
      this.setStatusMessage.emit(
        'Dummy new skill and questions generated successfully.');
    }, (errorResponse) => {
      this.setStatusMessage.emit(
        'Server error: ' + errorResponse.data.error);
    });
  this.adminTaskManagerService.finishTask();
}
```

Here we need to make sure that setStatusMessage is called with the correct message when the Promise is resolved or rejected. We can test the resolved case by:



```
describe('.generateNewSkillData', () => {
  it('should generate structures data', async(() => {
    spyOn(adminBackendApiService, 'generateDummyNewSkillDataAsync')
      .and.returnValue(Promise.resolve());
    spyOn(component.setStatusMessage, 'emit');
    component.generateNewSkillData();

    expect(component.setStatusMessage.emit)
      .toHaveBeenCalledWith('Processing...');

    fixture.whenStable().then(() => {
      expect(component.setStatusMessage.emit).toHaveBeenCalledWith(
        'Dummy new skill and questions generated successfully.');
    });
  }));
```

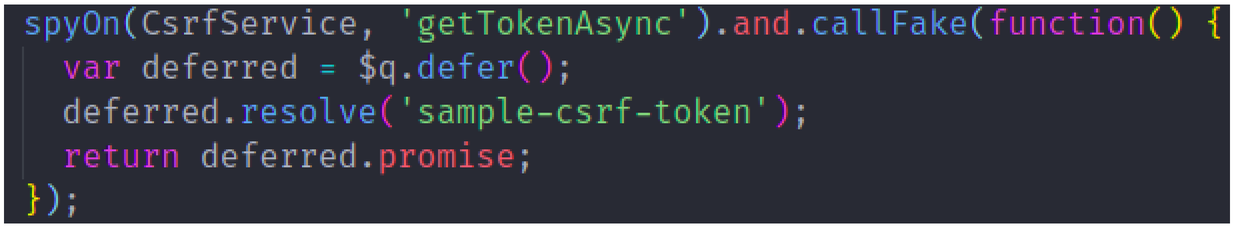
And the rejected case by,



```

    it('should show error message if new structues data' +
      'is not generated', async(() => {
      spyOn(adminBackendApiService, 'generateDummyNewSkillDataAsync')
        .and.returnValue(Promise.reject({
          data: {
            error: 'New skill data not generated.'
          }
        }));
      spyOn(component.setStatusMessage, 'emit');
      component.generateNewSkillData();

      expect(component.setStatusMessage.emit)
        .toHaveBeenCalledWith('Processing...');

      fixture.whenStable().then(() => {
        expect(component.setStatusMessage.emit).toHaveBeenCalledWith(
          'Server error: New skill data not generated.');
      });
    }));
  });
```

- In AngularJS we can mock a promise using $q API. $q.defer() returns the instance of the promise. We can use the following properties after we create a deferred object:
    - resolve - resolves the derived promise
    - reject - rejects the derived promise
    - promise - promise object associated with it
    - notify - gives updates on the status of the promise's execution

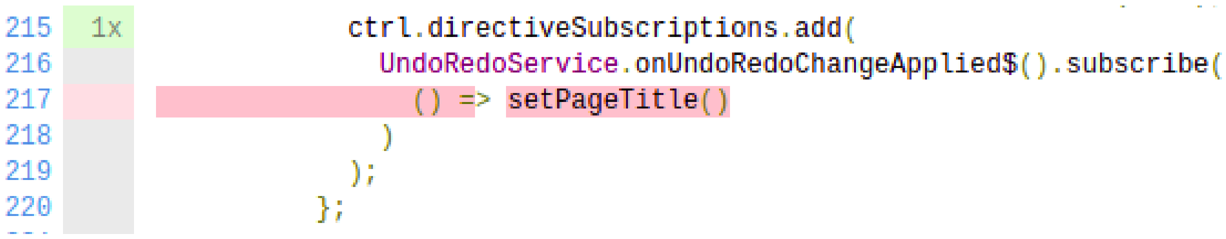
  Example - to setup csrf token



```
spyOn(CsrfService, 'getTokenAsync').and.callFake(function() {
  var deferred = $q.defer();
  deferred.resolve('sample-csrf-token');
  return deferred.promise;
});
```

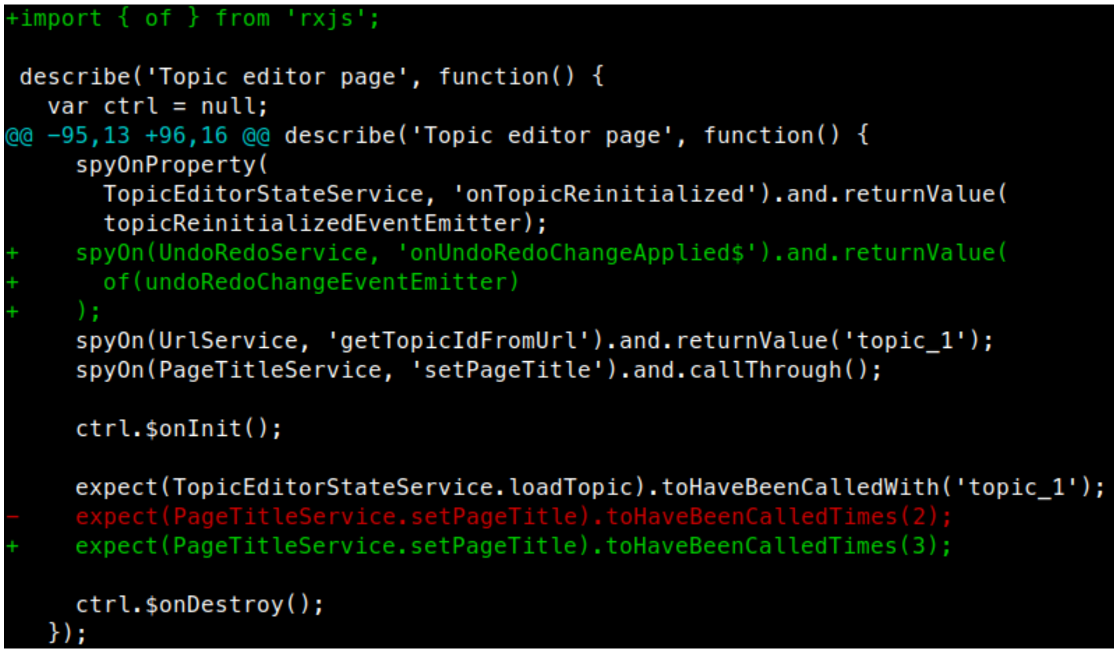
- Testing Observables

  In topic-editor-page.component.ts the following line is not covered,



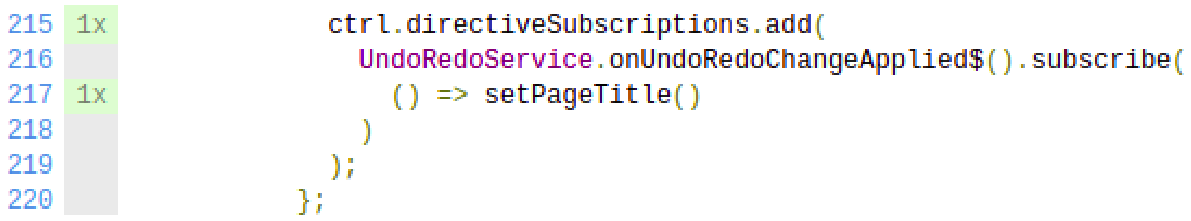
```
          ctrl.directiveSubscriptions.add(
            UndoRedoService.onUndoRedoChangeApplied$().subscribe(
              () => setPageTitle()
            )
          );
        };
```

  Here, we can spyOn the method onUndoRedoChangeApplied()$ which has return type Observable (not EventEmitter). We can test in the following way:



```
+import { of } from 'rxjs';

 describe('Topic editor page', function() {
   var ctrl = null;
@@ -95,13 +96,16 @@ describe('Topic editor page', function() {
     spyOnProperty(
       TopicEditorStateService, 'onTopicReinitialized').and.returnValue(
       topicReinitializedEventEmitter);
+    spyOn(UndoRedoService, 'onUndoRedoChangeApplied$').and.returnValue(
+      of(undoRedoChangeEventEmitter)
+    );
     spyOn(UrlService, 'getTopicIdFromUrl').and.returnValue('topic_1');
     spyOn(PageTitleService, 'setPageTitle').and.callThrough();

     ctrl.$onInit();

     expect(TopicEditorStateService.loadTopic).toHaveBeenCalledWith('topic_1');
-    expect(PageTitleService.setPageTitle).toHaveBeenCalledTimes(2);
+    expect(PageTitleService.setPageTitle).toHaveBeenCalledTimes(3);

     ctrl.$onDestroy();
   });
```

Now the line is covered.

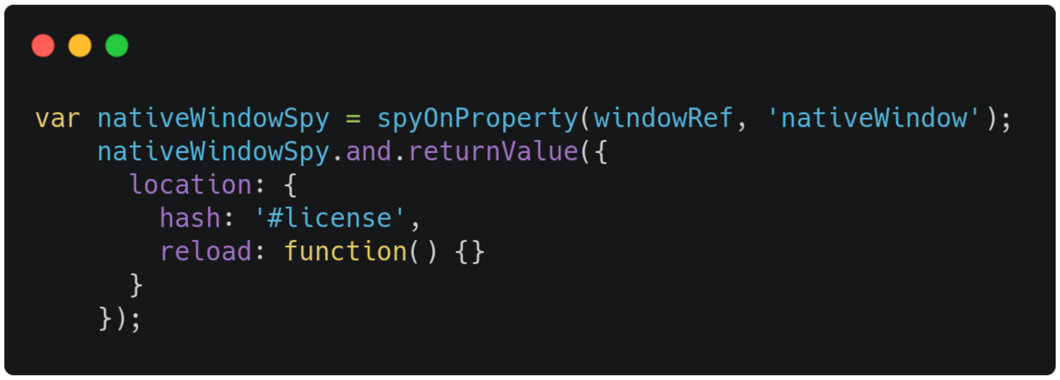


```
1x              ctrl.directiveSubscriptions.add(
                  UndoRedoService.onUndoRedoChangeApplied$().subscribe(
1x                  () => setPageTitle()
                  )
                );
              };
```

- Handling Window Reload

  To test a window reload, we can not directly call it in native form, as the tests would fail. Instead we can spy on `windowRef` and use `returnValue()` with it.

  For ex, we can test native `reload()` function using an empty mock



```
var nativeWindowSpy = spyOnProperty(windowRef, 'nativeWindow');
    nativeWindowSpy.and.returnValue({
      location: {
        hash: '#license',
        reload: function() {}
      }
    });
```

Milestones

Milestone 1

Key Objective: Complete approximately **1800** lines of code
Time: June 7 - July 12

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
1.1	300 lines of code have been tested	-	7/06/2021	12/06/2021
1.2	300 lines of code have been tested	-	12/06/2021	17/06/2021
1.3	300 lines of code have been tested	-	17/06/2021	22/06/2021
1.4	300 lines of code have been tested	-	22/06/2021	27/06/2021
1.5	300 lines of code have been tested	-	27/06/2021	2/07/2021
1.6	300 lines of code have been tested	-	2/07/2021	07/07/2021
1.7	Complete any remaining work, flakes or things that may go wrong	-	7/07/2021	11/07/2021

Milestone 2

Key Objective: Complete the remaining lines of code (approximately **1500** lines)
Time: July 17 - August 16

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
2.1	300 lines of code have been tested	-	17/07/2021	22/07/2021
2.2	300 lines of code have been tested	-	22/07/2021	27/07/2021
2.3	300 lines of code have been tested	-	27/07/2021	01/08/2021
2.4	300 lines of code have been tested	-	01/08/2021	06/08/2021
2.5	300 lines of code have been tested	-	06/08/2021	11/08/2021
2.6	Complete any remaining work, flakes or things that may go wrong	-	11/08/2021	15/08/2021

Optional Sections

Additional Project-Specific Considerations

Documentation Changes

- Improving the existing guide to write frontend tests in order to help future developers.
- Add debugging docs made during the course of the project.