About You

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

Since high school, I have been interested in programming and have consistently tried to improve my skillset in that regard. To this end, I started looking for open source organizations that I could work with and contribute code. During this process, I came across the Oppia organization. In addition to a high quality and interesting codebase, Oppia's objective, to create a set of free, high-quality lessons regardless of where they are and or what traditional resources they have access to, really spoke to me. My original hometown is a remote village and a platform such as Oppia would allow children and adults living there to get access to quality education regardless of what resources they have access to.

When I was running through the list of ideas, this particular project caught my eye. In addition to the idea itself, I believe that working on this project will help me understand the overall codebase better. Even though the idea looks easy at first look, the details in this project make the implementation a challenge. Specifically, to work on this project, one needs to not only understand the working of the directive that is being tested but also all the services and other dependent elements being used in the file. Writing tests is also of immense value and can be viewed as being synonymous with writing code documentation.

Prior experience

I had started with no knowledge in writing frontend unit tests but as time went by I started to see the complexity and thought that goes into testing. The common misconception is that the tests are easy to write. However, one has to understand the working of the complete file and also other files and functions used in the file one is going to write unit tests for.

As I started writing more frontend tests I started to gain more knowledge on how to write a frontend unit test and I have helped other contributors in Gitter to write frontend tests.

PRs

1. Wrote a complete service along with the frontend test file. - #11654
2. Wrote a frontend unit test file for AngularJS directive admin-navbar. - #11962
3. Migrated the files from AngularJS directive to Angular Component and wrote a complete frontend unit test file. - #12354
4. This was the first time I had worked with Angular, Typescript, and frontend tests. I had learned a lot from this PR. I thank Kevin Thomas and Sandeep Dubey for explaining and teaching me the ropes. Not the most impressive PR but one that I had learned the most in. - #11690
5. Identified an unused directive - #11201

Contact info and timezone(s)

Email: praneethg2001@gmail.com
Hangouts: praneethg2001@gmail.com
Timezone: IST

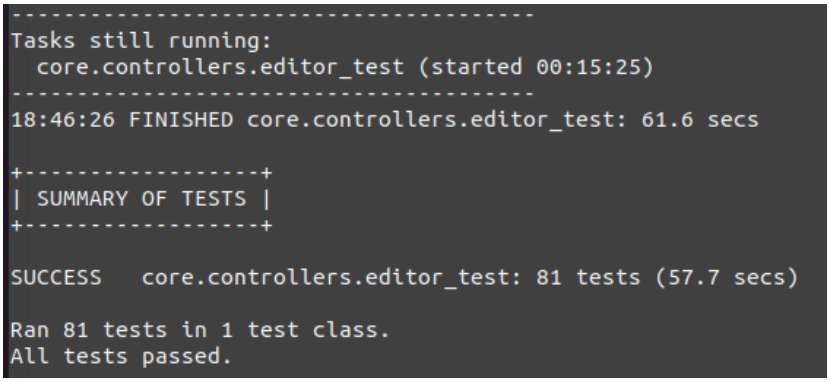
Time commitment

From	To	Hours
17/5	14/6	5 hours/week
15/6	18/7	70 hours/week
19/7	23/8	30 hours/week

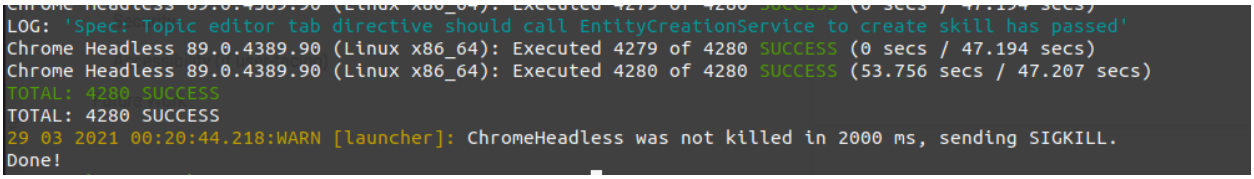
Essential Prerequisites

Answer the following questions:

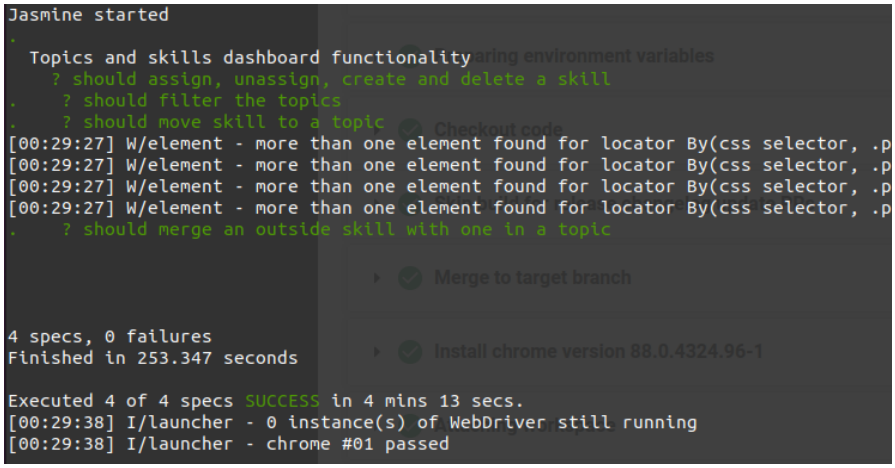
- *I can run a single backend test target on my machine.*



- *I can run all the frontend tests at once on my machine.*



- *I can run one suite of e2e tests on my machine.*



Other summer obligations

- I have my College classes till May 28th
- I have my College exams from 29th - 15th

Communication channels

- Meeting twice a week on google meet to discuss how the tasks are going.
- Hangouts chat for quick messages and questions.
- Google sheet to track what files I'm working on and their PR status.

Application to multiple orgs

I have applied only to Oppia for my GSoC 2021

I have applied only to Oppia for my GSoC 2021

I have written 2 proposals for Oppia.

I have no particular preference between the 2. Hence I'm going with the order I have written the proposals in

1. Write frontend tests
2. Solving dev-workflow issues

Project Details

Product Design

Oppia is an educational online platform that has taken a new approach to provide education and knowledge to the world. When “the world” is mentioned it not only means places with good connectivity with the internet but also places that have low network coverage/internet speed. Oppia is able to achieve this thanks to a couple of features such as, pre-load lessons as soon as the first card appears, continuously improving the speed at which the website loads, etc…

In order to have features supporting all kinds of activities, Oppia has a large codebase, and to make sure that all these features and the user experience don’t get affected, Oppia’s codebase is tested in numerous methods (e2e test, backend test, frontend test, Lighthouse, to mention a few).

In a frontend test, the code is tested for its functionality and not its coverage, i.e., the number of lines tested. This way it helps to ensure that none of the Oppia’s features get affected. This is done with the help of unit testing. A ***Unit*** is the smallest group of code that can be maintained and executed independently.

Importance of Unit tests:

- **Provides Documentation:** Unit testing provides documentation of the system. Anyone looking to learn what functionality is provided by a unit of code and how to use it can look at the unit tests to gain a basic understanding of its functionality.
- **Maintains healthy code:** Unit tests make sure that every unit of code functions as intended, hence drastically reducing the chance of a regression (re-occurrence of a fixed bug) or a new bug.
- **Ensures quality of interactions:** We can test components and directives that affect the frontend of a website. This ensures that the user experience is consistent and does not introduce bugs.

Goal:

- 100% coverage of Component, Directive, and Services.
- Accelerate the process of migration by converting AngularJS Directive to Component

Technical Design

Architectural Overview

Oppia file structure:

Oppia:

- **Core/Templates**
    - All the frontend directives, controllers, filters & services.
- **Extensions**
    - **interactions**: contains interactions supported on the Oppia website.
    - **objects**: contains various typed editors like boolean editor, file path editor, etc.
    - **rich_text_components**: contains rich text components supported on Oppia RTE.
    - **value_generators**: contains value generators used in Oppia.
    - **visualizations**: contains visualizations used in Oppia.

File Type	Total Lines	Covered Lines	Uncovered Lines
Component	7646	7254	392
Directive	10794	2658	8136
Service	11636	10247	1389
Other	9828	9028	800

Table: No: of lines covered by frontend tests

Components

Components are the main building block for Angular applications. Each component consists of:

- An HTML template (declares what renders on the page)
- A Typescript class (defines behavior)
- A CSS selector (defines how the component is used in a template)
- CSS styles applied to the template (Optional)

Currently, we have components and directives that are supposed to be converted to components due to the presence of `restrict: E`. The presence of `restrict: E` indicates that the file will be used as an element, .i.e, <oppia-component> (More Details in Migrate AngularJS Directive to AngularJS Component)

Directives

Directives are custom HTML attributes that tell angular to change the style or behavior of the Dom elements.
The directive files are generally of 2 types:

- Component Directives
    - These form the main class having details of how the component should be processed, instantiated, and used at runtime.
- Attribute Directives
    - Attribute directives deal with changing the look and behavior of the DOM element. You can create your directives as shown below.

Given below are 35 directive and component files that are required to be tested. I have also included information with respect to number of lines and if a *Spec file is already present.

S. No:	File Name	No.of Lines	Complexity	Is Migrated	Category	Spec file
1.	learner-view-info.dire	105	Medium	No	core/templates	No

	ctive.ts					
2.	list-of-tabs-editor.directive.ts	71	Easy	No	extensions/objects	No
3.	list-of-unicode-string-editor.directive.ts	49	Easy	No	extensions/objects	No
4.	logic-error-category-editor.directive.ts	77	Easy	No	extensions/objects	No
5.	logic-question-editor.directive.ts	103	Medium	No	extensions/objects	No
6.	math-expression-content-editor.directive.ts	172	Medium	No	extensions/objects	No
7.	mathjax-bind.directive.ts	45	Easy	No	core/templates	No
8.	misconception-editor.directive.ts	174	Medium	No	core/templates	No
9.	music-phrase-editor.directive.ts	88	Easy	No	extensions/objects	No
10.	nonnegative-int-editor.directive.ts	52	Easy	No	extensions/objects	No
11.	normalized-string-editor.directive.ts	99	Easy	No	extensions/objects	No
12.	number-with-units-edit	59	Easy	No	extensions/objects	No

	or.directive.ts					
13.	object-editor.directive.ts	74	Easy	??	core/templates	No
14.	oppia-interactive-code-repl.directive.ts	270	Hard	No	extensions	No
15.	oppia-interactive-continue.component.ts	81	Easy	Yes	extensions/interactions	No
16.	oppia-interactive-drag-and-drop-sort-input.directive.ts	120	Medium	No	extensions/interactions	No
17.	oppia-interactive-end-exploration.directive.ts	121	Medium	No	extensions/interactions	No
18.	oppia-interactive-image-click-input.directive.ts	232	Medium	No	extensions/interactions	No
19.	oppia-interactive-interactive-map.directive.ts	184	Medium	No	extensions/interactions	No
20.	oppia-interactive-item-selection-input.directive.ts	142	Medium	No	extensions/interactions	No
21.	oppia-inter	323	Hard	No	extensions	No

	active-logic-proof.directive.ts				/interactions	
22.	oppia-interactive-multiple-choice-input.directive.ts	119	Medium	No	extensions/interactions	No
23.	oppia-interactive-music-notes-input.directive.ts	880	Hard	No	extensions/interactions	No
24.	oppia-interactive-number-with-units.directive.ts	121	Medium	No	extensions/interactions	No
25.	oppia-interactive-numeric-input.directive.ts	82	Easy	No	extensions/interactions	No
26.	oppia-interactive-pencil-code-editor.directive.ts	212	Medium	No	extensions/interactions	No
27.	oppia-interactive-set-input.directive.ts	114	Medium	No	extensions/interactions	No
28.	oppia-interactive-text-input.directive.ts	90	Easy	No	extensions/interactions	No
29.	oppia-noninteractive-collapsible.directive.ts	47	Easy	No	extensions/rich_text_components	No

30.	oppia-noninteractive-image.directive.ts	142	Medium	No	extensions/rich_text_components	No
31.	oppia-noninteractive-link.directive.ts	74	Easy	No	extensions/rich_text_components	No
32.	oppia-noninteractive-math.directive.ts	107	Medium	No	extensions/rich_text_components	No
33.	oppia-noninteractive-skillreview.directive.ts	89	Easy	No	extensions/rich_text_components	No

Services

Services allow you to define code that's accessible and reusable to other files. A common use case for services is when you need to communicate with the backend to send and receive data.

Given below are 35 service files that are required to be tested. I have also included information with respect to number of lines and if a *Spec file is already present.

S.No:	File Name	No.of Lines	Complexity	Is Migrated	Spec file present
1.	answer-classification.service.ts	200	M	Not needed	Yes
2.	audio-player.service.ts	177	M	Not needed	No
3	audio-preloader.service.ts	139	Medium	Not Needed	Yes
4	audio-translation-manager.service.ts	92	Easy	Not Needed	Yes
5	autogenerated-audio	78	Easy	Not Needed	No

	-player.service.ts				
6	base-interaction-validation.service.ts	124	Medium	Not Needed	No
7	change-list.service.ts	262	Hard	Not Needed	No
8	code-repl-prediction.service.ts	335	Hard	Not Needed	Yes
9	collection-editor-state.service.ts	219	Medium	Not Needed	Yes
10	collection-update.service.ts	284	Hard	Not Needed	Yes
11	context.service.ts	308	Hard	Not Needed	Yes
12	contribution-and-review.service.ts	125	Medium	Not Needed	Yes
13	contribution-opportunities-backend-api.service.ts	201	Medium	Not Needed	Yes
14	contribution-opportunities.service.ts	132	Medium	Not Needed	No
15	csrf-token.service.ts	64	Easy	Not Needed	Yes
16	current-interaction.service.ts	167	Medium	Not Needed	Yes
17	editable-collection-backend-api.service.ts	149	Medium	Not Needed	Yes
18	editable-story-backend-api.service.ts	266	Hard	Not Needed	Yes
19	email-dashboard-data.service.ts	120	Medium	Not Needed	Yes
20	entity-creation.service.ts	82	Easy	Not Needed	Yes
21	exploration-creation.service.ts	122	Medium	Not Needed	No

22	exploration-diff.service.ts	373	Hard	Not Needed	No
23	exploration-engine.service.ts	480	Hard	Not Needed	No
24	exploration-player-state.service.ts	263	Hard	Not Needed	Yes
25	exploration-save.service.ts	468	Hard	Not Needed	No
26	exploration-states.service.ts	536	Hard	Not Needed	Yes
27	expression-evaluator.service.ts	133	Medium	Not Needed	Yes
28	expression-interpolation.service.ts	105	Medium	Not Needed	Yes
29	fatigue-detection.service.ts	73	Easy	Not Needed	No
30	fraction-input-validation.service.ts	327	Hard	Not Needed	Yes
31	graph-detail.service.ts	81	Easy	Not Needed	No
32	graph-input-rules.service.ts	225	Medium	Not Needed	Yes
33	graph-input-validation.service.ts	137	Medium	Not Needed	Yes
34	graph-layout.service.ts	625	Hard	Not Needed	No
35	hint-and-solution-modal.service.ts	77	Easy	Not Needed	Yes

Implementation Approach

For implementation purposes, I am dividing files based on the number of lines present. I have referred to the 4057 issue. However, this method is not accurate as the difficulty may vary depending on the experience in testing various parts of the code.

Complexity	Easy	Medium	Hard
Number of lines	up to 100 lines	up to 250 lines	at least 250 lines

Flowchart:

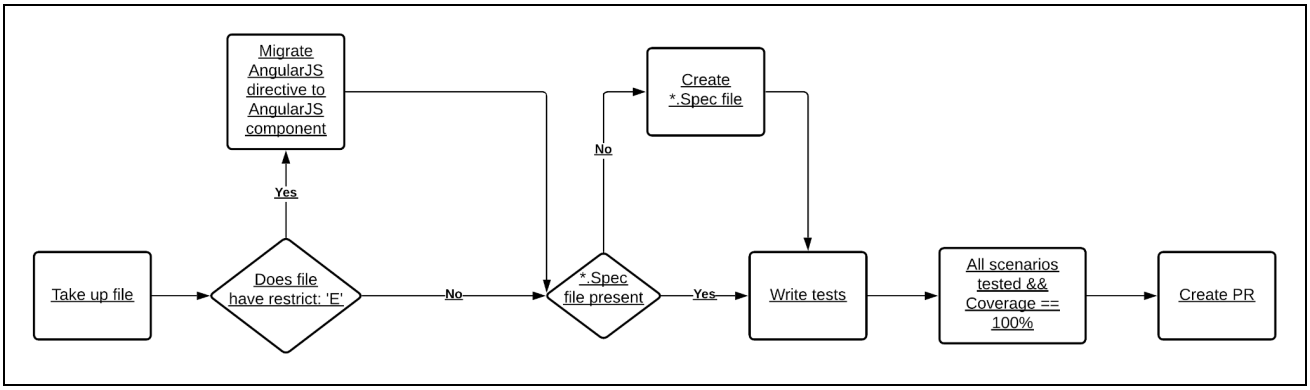


Fig - Flowchart on how to write frontend tests for a file

Parts of the code that could take more time than expected:

- Tricky scenarios

1. Number of lines to be tested per student = 3,300 lines of code
2. Amount of time = 10 weeks
3. NO: of lines per week = 3,300/10 = 330 lines of code
4. Assuming 4 hours a day is spent on writing tests
5. No: of lines per day = 330/4 = 12 (approx.)

Note:

When writing frontend tests not only includes writing tests but also understanding the file going to be tested, other files being utilized within the file being tested, and much more. This will be explained in detail in the next section.

Testing Approach

Analyze the file

Before writing tests for the file we must have a clear understanding of the functionality of the file and how it achieves it. It will also require us to understand the services being used by that file.

1. Find where the file is being implemented on the Oppia website.
2. Run localhost and understand how each function works with the help of the console log. Google Chrome developer tools will also be helpful here to place breakpoints in the code to understand what is present in the variables.
3. We then have to determine if the file has more than 1 high-level task since these will become the initial describe functions. Ex: for webpages we have:

```
describe('when the device is narrow', () => {
…
}
describe('when device is not narrow', () => {
…
}
```

Fig - settings-tab-component.spec.ts

   a. In the above image, we are testing with 2 describe() functions since one of the functions present in the file is only triggered when the device is narrow. In this test, the WindowDimensionsService is mocked to set the window as narrow.

4. We then write a unit test. This is explained in detail in the subsequent sections.

Migrate AngularJS Directive to AngularJS Component

Which files to Migrate?

- The AngularJS directive files that have restrict: ‘E’ have to be migrated to an AngularJS component file to make it easier to test and assist in Angular Migration.

- The presence of `restrict: E` indicates that the file will be used as an element, .i.e, <oppia-component>. Hence after converting it to a component we no longer have to use `restrict: E` (Since components can be elements only)

Migration Explained

Given below are examples converting an AngularJS directive file to anAngularJS component file.

Example - 1

/contributor-dashboard-page/opportunities-list-item/opportunities-list-item.directive.ts

```
angular.module('oppia').directive('opportunitiesListItem', [
 'UrlInterpolationService', function(
    UrlInterpolationService) {
  return {
    restrict: 'E',
    scope: {
      getOpportunity: '&opportunity',
      onClickActionButton: '=',
      isLabelRequired: '&labelRequired',
      isProgressBarRequired: '&progressBarRequired',
      getOpportunityHeadingTruncationLength:
        '&opportunityHeadingTruncationLength'
    },
    bindToController: {},
    templateUrl: UrlInterpolationService.getDirectiveTemplateUrl(
      '/pages/contributor-dashboard-page/opportunities-list-item/' +
      'opportunities-list-item.directive.html'),
    controllerAs: '$ctrl',
    controller: [
```

Converted to

/contributor-dashboard-page/opportunities-list-item/opportunities-list-item.component.ts

```
angular.module('oppia').component('opportunitiesListItem', {
 bindings: {
  getOpportunity: '&opportunity',
  onClickActionButton: '=',
```

```
  isLabelRequired: '&labelRequired',
  isProgressBarRequired: '&progressBarRequired',
  getOpportunityHeadingTruncationLength:
    '&opportunityHeadingTruncationLength'
},
template: require('./opportunities-list-item.component.html'),
controller: [
```

Example-2

```
angular.module('oppia').directive('scoreRing', [
'UrlInterpolationService', function(UrlInterpolationService) {
  return {
    restrict: 'E',
    scope: {},
    bindToController: {
      getScore: '&score',
      testIsPassed: '&testIsPassed'
    },
    templateUrl: UrlInterpolationService.getDirectiveTemplateUrl(
      '/components/score-ring/score-ring.directive.html'),
    controllerAs: '$ctrl',
    controller: ['$scope', 'COLORS_FOR_PASS_FAIL_MODE',
```

Converted to

```
angular.module('oppia').component('opportunitiesListItem', {
bindings: {
     getScore: '&score',
     testIsPassed: '&testIsPassed'
},
template: require('./score-ring.directive.html'),
controller: [
```

Steps to convert AngularJS Directive to AngularJS Component

1. Change `.directive(` to `.component(` This registers the directive as a component
2. Remove `restrict: 'E',` since it is no longer required. (Since components can be elements only)
3. Change `scope` and `bindToController` to `bindings`
4. Remove `controllerAs: '$ctrl',` since it is no longer required.

5. Convert templateUrl to template with require

Unit tests in General

Components of a *.spec.ts file

- **Import/requires** - This is used to import services, directives, etc.. required for testing the file.
- **describe** - This is almost like a high-level test suite. Within the describe() function, the unit tests are present.
- **beforeEach** - To avoid repeated code we use this function to execute all the code that is to be run before the execution of each unit test
- **afterEach** - This is the same as `beforeEach` but the code present in afterEach runs after each unit test.
- **expect** - We use this to test the result of the function/variable bearing tested. This is commonly paired with matchers. Some commonly used matchers are. toBe(), toEqual, toBeNull() etc...

Unit tests format

A unit test can be split into the following categories:

- **Setup** - inputs/environment needed for the test are prepared in this step
- **Baseline verification** - check the initial values here. The values should be checked again in the endline verification. This is done to make sure the action is working (expected change occurs).
- **Action** - perform the action or function call that leads to the expected change.
- **Endline verification** - check if the values present in the baseline verification have changed,i.e., the expected change has occurred.

The code inside the it(' ' …. Function will be separated into the categories mentioned above by an empty/blank line. Ex:

```
it('should set profileDropdownIsActive to false', () => {
 // Setup
 ctrl.profileDropdownIsActive = true;
 // Baseline verification
```

```
 expect(ctrl.profileDropdownIsActive).toBe(true);
 // Action
 ctrl.deactivateProfileDropdown();
 // Endline verification
 expect(ctrl.profileDropdownIsActive).toBe(false);
});
```

Approach to write unit tests:

1. Test whether all the variables initialized with the correct values and the functions are called in the init function.
2. Write a test for each possible test case (test all possible scenarios, .i.e, all the cases that the unit to be tested can be used for).

Difference between an Angular 2+ unit test and an AngularJS test:

Mentioned below are a couple of notable differences seen in unit tests for Angular 2+ and AngularJS testing.

	AngularJS	Angular 2+
HTTP	$httpBackend	httpTestingController with fakeAsync() and flushMicrotasks()
Import	Uses require	Uses import
Injecting service/component	injector	TestBed

Testing a Component

Given below is a boilerplate that can be used for writing a frontend test for an Angular 2+ Component. We can start a new spec file using this boilerplate. (Source: WIKI)

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { BannerComponent } from './banner.component';
 describe('BannerComponent', () => {
  let component: BannerComponent;
  let fixture: ComponentFixture<BannerComponent>;
```

```
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ BannerComponent ]
    })
    .compileComponents();
  }));
  beforeEach(() => {
    fixture = TestBed.createComponent(BannerComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });
  it('should create', () => {
    expect(component).toBeDefined();
  });
});
```

- Import the files required for writing the frontend tests
- We can inject all the dependencies using TestBed.configureTestingModule
  - declarations: It is the component being tested
  - providers: List all the dependencies here.

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [PromoBarComponent],
    providers: [
      WindowRef,
      {
        provide: PromoBarBackendApiService,
        useClass: MockPromoBarBackendApiService
      }]
  }).compileComponents();
}));
```

  - If a service need to be mocked we need to put provide: {Service} useClass: {MockService}
  - For a normal dependency we can inject using TestBed.get()

```
ecs = TestBed.get(StateEditorService);
```

  - fixture.detectChanges(); It is similar or $scope.apply() in AngularJS. It is used for change detection. It is usually accompanied by a tick() (Used to simulate passage of time) which can be used only inside a fakeAsync()

Testing a Directive

The AngularJS directive files that have to `restrict: 'E'` will have to be migrated to an AngularJS component file to make it easier to test. The presence of `` `restrict: E` `` indicates that the file will be used as an element, .i.e, `<oppia-component>`

The steps for migration are listed in the Migrate AngularJS Directive to AngularJS Component Section.

The following are the advantages of converting an AngularJS directive to an AngularJS component:

- simpler configuration than plain directives

writing component directives will make it easier to upgrade to Angular 2+

Testing a Service

Possible extensions under service:

- *.service.ts
- *Factory.ts
- *.factory.ts
- *.tokenizer.ts

Steps to write frontend tests for services

- Import the files required for writing the frontend tests
- We can inject all the dependencies using `TestBed.configureTestingModule`
    - `declarations`: It is the component being tested
    - `providers`: List all the dependencies here.

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ExampleComponent],
    imports: [HttpClientTestingModule],
    providers: [
      WindowRef,
      {
        provide: ExampleBackendApiService,
        useClass: MockExampleBackendApiService
```

```
        }]
    })
      httpTestingController =
TestBed.get(HttpTestingController);

 }));
```

- If a service needs to be mocked we need to put provide: {Service} useClass: {MockService}
- In AngularJS we use httpBackend instead of HttpTestingController.
  - Along with HttpTestingController we use flush and flushMicrotasks

Tricky Tests

Given below are some of the cases that are tough to write a frontend test for.

1. **Reloading a web page**
   - There are a few instances when the page has to be reloaded or redirected which results in a full page reload. However, Reloading a page in the frontend test results in an error being raised in Karma.
   - Example: core/templates/pages/about-page/about-page.component.spec.ts
     Given below is a mock window used for testing location.href since changing location.href causes a full page reload.

```
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;
}
}
```

2. **HTTP call**
   - This involves finding the URL that is used to perform the HTTP call. The URL can be found by running on localhost and triggering the HTTP request.
   - The URL will be seen in the terminal.
   - We can also use the console log to see the contents of the http request.
   - Example:

     EndExploration/directives/oppia-interactive-end-exploration.component.ts

```
      if (ctrl.isInEditorPage) {
              // Display a message if any author-recommended
explorations are
        // invalid.
        $http.get(EXPLORATION_SUMMARY_DATA_URL_TEMPLATE, {
          params: {
            stringified_exp_ids: JSON.stringify(
              authorRecommendedExplorationIds)
          }
        }).then(function(response) {
          var data = response.data;
          var foundExpIds = [];
```

```
            data.summaries.map(function(expSummary) {
              foundExpIds.push(expSummary.id);
            }); ...
```

- Test

```
const httpResponse = {
  summaries: [{
    id: '0'
  }]
};
      ...
      ...
      ...
  const explorationIds = ['0', '1'];
      const  requestUrl  =  '/explorationsummarieshandler/data?'  +
'stringified_exp_ids='+ encodeURI(JSON.stringify(explorationIds));
      ...
      ...
      ...
   $httpBackend.expectGET(requestUrl).respond(httpResponse);
    ctrl.$onInit();
    $httpBackend.flush();
    $scope.$apply();
```

3. **Subscribe to an event**
   - This involves testing an Observable. This can be tested by mocking the eventemitter. This will run the code present inside the .subscribe
   - Example: exploration-editor-page/settings-tab/settings-tab.component.ts
   - Code:

```
      ctrl.directiveSubscriptions.add(
        UserExplorationPermissionsService.onUserExplorationPermissionsFetched
          .subscribe(
            () => {
              UserExplorationPermissionsService.getPermissionsAsync()
                .then(function(permissions) {
                  ctrl.canUnpublish = permissions.canUnpublish;
                  ctrl.canReleaseOwnership = permissions.canReleaseOwnership;
```

```
                // TODO(#8521): Remove the use of $rootScope.$apply()
                // once the controller is migrated to angular.
                $rootScope.$applyAsync();
              });
            }
          )
      );
```

- Test

```
  it('should display Unpublish button', function() {
    ctrl.canUnpublish = false;
    expect(ctrl.canUnpublish).toBe(false);

    userExplorationPermissionsService.
      onUserExplorationPermissionsFetched.emit();
    $scope.$apply();

    expect(userExplorationPermissionsService.getPermissionsAsync)
      .toHaveBeenCalled();
    expect(ctrl.canUnpublish).toBe(true);
  });
```

4. **Promises**
   - In some cases, a mock object is required to be created which will be returned from the promise.
   - The promise can be returned by either spying on the function or creating a mocking of the service itself.
     - Spying Ex:

```
spyOn(explorationImprovementsBackendApiService,
'getConfigAsync')
      .and.returnValue(Promise.resolve(newExpImprovementsConf
ig(true)));
```

     - Mocking the service

```
class MockQuestionBackendApiService {
 fetchTotalQuestionCountForSkillIdsAsync() {
   return Promise.resolve(1);
```

```
 }
}
```

- We can use promise.reject() for testing failed promises.
- Example - 2 (Angular 2+):
  - Code:

```
async fetchStoryDataAsync(
    topicUrlFragment: string,
    classroomUrlFragment: string,
    storyUrlFragment: string): Promise<StoryPlaythrough> {
  return new Promise((resolve, reject) => {
    this._fetchStoryData(
                  topicUrlFragment,   classroomUrlFragment,
storyUrlFragment,
      resolve, reject);
  });
}
```

  - Test

```
it('should handle errorCallback for fetching an existing
story',
  fakeAsync(() => {
    let successHandler = jasmine.createSpy('success');
    let failHandler = jasmine.createSpy('fail');

    storyViewerBackendApiService.fetchStoryDataAsync(
            'abbrev', 'staging', '0').then(successHandler,
failHandler);

    let req = httpTestingController.expectOne(
      '/story_data_handler/staging/abbrev/0');
    expect(req.request.method).toEqual('GET');
    req.flush('Invalid request', {
      status: 400,
      statusText: 'Invalid request'
    });

    flushMicrotasks();
```

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

- This test involves testing a promise that is returned along with a success or reject callback.

5. **Error**
   - Errors cannot be tested how functions are normally tested since the problem is that the exception is thrown before an `expect()` can deal with it, therefore it slips out and fails the test case. To fix this the call has to be wrapped in a function which Jasmine will invoke from within expect
   - Example: `core/templates/domain/story/story-node.model.ts`
   - Test

```
 it('should correctly throw error when duplicate values are added
to arrays',
   () => {
     expect(() => {
       _sampleStoryNode.addDestinationNodeId('node_2');
        }).toThrowError('The given node is already a destination
node.');
     expect(() => {
       _sampleStoryNode.addPrerequisiteSkillId('skill_1');
      }).toThrowError('The given skill id is already a prerequisite
skill.');
     expect(() => {
       _sampleStoryNode.addAcquiredSkillId('skill_2');
        }).toThrowError('The given skill is already an acquired
skill.');
   });
```

6. **UI actions**
   - For this, I'm taking an example of testing the scroll action.
   - This was a tricky scenario to test when I first attempted to help a contributor on Gitter write a frontend test involving the scroll action.
   - It is always better to mock UI elements. Since this has a chance to flake if it is not mocked (Source)

```
      it('should animate html and body to 20px top when calling function' +
      ' from option 1', function() {
      ctrl.$onInit();

      var elementMock = $(document.createElement('div'));
                              spyOn(window,      '$').withArgs('html, body').and.returnValue(elementMock);
      spyOn(elementMock, 'animate');

      $scope.TRANSLATION_TUTORIAL_OPTIONS[1].fn(false);

      expect(elementMock.animate).toHaveBeenCalledWith({
        scrollTop: 20
      }, 1000);
    });
```

- The code above mocks a UI element . The scroll value is checker with respect to this .

7. **Difficult to reach code**
   - There are cases where a variable or function cannot be accessed directly for testing. In these cases, we have to fake function calls/responses, create a mock service or window, etc…
   - General solution:
     - Understand the requirements that are needed to be satisfied for the code to be executed.
       - **Note:** This might involve understanding the functionality of other files too.
       - Running localhost can help to understand how each function works. The utilization of console logs can assist in this method.
     - Check which functionalities have to be mocked to satisfy the requirements.
     - After satisfying the requirements the difficult to reach code should execute.
   - Some scenarios

- Anonymous functions are tough to test since they cannot be accessed directly. We must use the function that calls the anonymous function to test it.
- Local variables defined using var, let, etc.. cannot be used for testing a function. Instead, we have to test functions or variables that it directly or indirectly affects.
- In very rare cases a function or the complete file itself is not being utilized anywhere. In these cases, we have to search if the function is being utilized anywhere either by doing a global search for a function or trying to find its implementation by running localhost.

- For callbacks, a mock function has to be created to
- Example: Anonymous functions

```
var _isSuggestionHandled = () => {
  return (
    ctrl.activeThread !== null &&
    ctrl.activeThread.isSuggestionHandled());
};

var _isSuggestionValid = () => {
  return (
    ctrl.activeThread !== null &&
    ExplorationStatesService.hasState(
      ctrl.activeThread.getSuggestionStateName()));
};

var _hasUnsavedChanges = () => {
  return ChangeListService.getChangeList().length > 0;
};

ctrl.getSuggestionButtonType = () => {
  return (
    !_isSuggestionHandled() && _isSuggestionValid() &&
    !_hasUnsavedChanges()) ? 'primary' : 'default';
};
```

- Here we notice that there are 3 functions that are to be tested. However, we cannot directly access the `_isSuggestionHandled`, `_isSuggestionValid`, and `_hasUnsavedChanges` directly. Hence we must test them by using `getSuggestionButtonType` function instead.
- `getSuggestionButtonType` calls the difficult to reach functions and we can test them using the value returned.

- Adding the test below covers all the 3 functions. However only using this test will result in another problem known as Branch not covered. The branch not covered issue can be resolved by writing another test to test the `getSuggestionButtonType` function to return 'default' instead.

```
  it('should evaluate suggestion button type to be default
when a feedback' +
    ' thread is selected', function() {
                                var          thread          =
suggestionThreadObjectFactory.createFromBackendDicts({
      status: 'open',
      subject: '',
      summary: '',
      original_author_username: 'Username1',
      last_updated_msecs: 0,
      message_count: 1,
      thread_id: '1',
    }, {
      suggestion_type: 'edit_exploration_state_content',
      suggestion_id: '1',
      target_type: '',
      target_id: '',
      status: 'open',
      author_name: '',
      change: {
        state_name: '',
        new_value: '',
        old_value: '',
      },
      last_updated_msecs: 0
    });
                              spyOn(threadDataBackendApiService,
'getThread').and.returnValue(thread);
                              spyOn(threadDataBackendApiService,
'getMessagesAsync').and.returnValue(
      $q.resolve());

    ctrl.setActiveThread('1');
```

```
  $scope.$apply();

  expect(ctrl.getSuggestionButtonType()).toBe('default');
});
```

Flaky test

A flaky test is a test that both passes and fails periodically without any code changes. A flaky test not only causes delays but also wastes computer resources. Since each time a flaky test fails then the test has to be restarted.

Some of the causes:

- Tests failing due to timeout
    - This occurs when done is used in the unit test. Usage of done is error-prone and must be avoided. In place of done, async() can be used in the unit test.
- Incomplete isolation
    - If the setup and cleanup after each test is over are not done properly it can lead to flakes.
    - Example:
    - An HTTPRequest that is generated in one test is flushed in the next test.
- Spying on a window rather than mocking it for testing UI.
    - Example: #12064
        - In this example, window is spied upon with spyOn, and this leads to a flaky test.
        - To resolve a dummy div was added for testing it and removed at the end of the test.

How to avoid

- Run the frontend tests multiple times
- Following the format for unit tests mentioned here.
- Clean setup before each test (`beforeEach()`) and cleanup after each test (`afterEach()`).

  Note: afterEach() is used in cases like http tests etc.

```
afterEach(() => {
  httpTestingController.verify();
});
```

What to do if a flaky test is encountered?

To handle flakes we can follow a general plan:

1. Run all the frontend tests multiple times. This will show the flaky tests.
2. Identify the test suite which might contain the flaky test.
3. Run the flaky test multiple times by using fit(...).
    a. It determines whether the method of testing for that unit test is the cause or not.
    b. Collect error logs to obtain as much information about the flaky test.
4. Run the describe(...) where the flaky test is present multiple times by using fdescribe(...).
    a. It determines whether the other unit tests are interfering with the current test.
5. Fix flake
    a. Given below are some scenarios that might cause a flaky test.
    b. Multiple $timeouts
        i. When testing a $timeout callback, another $timeout in the unit tests was called to wait for the original callback to be called. This lead to errors
        ii. Hence when testing $timeout use $flushPendingTasks instead.
    c. Reassigning global values with a spy
        i. Example: #9660#issuecomment
        ii. This should be resolved by finding another approach
    d. Forgotten to remove a mock HTML body element.
        i. Fix: document.body.removeChild(dummyDiv);

I'm planning to allot some time to address issues such as flaky tests, PR Reviews, tricky tests, etc…
In case a flaky test has been discovered. I will revert the PR as soon as a flaky test is discovered and attempt to resolve the issue within 48 hours. If I'm facing issues in resolving the flaky tests then I will consult with my mentors on what steps I could potentially take to resolve the flaky test.

FAQ

1. How to deal with code I have no experience testing?

    a. I would first analyze the file. Then I would write unit tests to test the files. In some places, we might experience tricky tests. This can be resolved by either referring to similar tests or searching the internet for a similar test. I have also written some tricky tests and Difficult to reach code from my experience.
2. What would you do if a particular line of code is not easily reachable? What steps would you take to unblock yourself in this scenario?
    a. In cases like these, I'll have to use the function that indirectly runs the Difficult to reach code. I will find the function that indirectly runs the Difficult to reach code, understands the requirements required to execute the code, and then tests it.
3. How will you prevent flaky tests from being added to the codebase? Can we do anything to protect against flakes?
    a. This something I would like to avoid in the first place. It will not only waste time but also consume resources (computing resources). I have written a basic plan in this section.
4. What is your plan of action in case a test that you wrote starts to flake once it has been merged to develop? Considering that it would add to your already planned tasks?
    a. This information has also been written in this section in the last paragraph.

My Experience:

Prevent-page-unload-event.service.ts

This was the First service file I had written completely on my own. This PR taught me the thought process that goes into implementing just one feature. This initially started as simple PR with less than 20 lines of code changes, but as we started to look from the user perspective we found that this could cause a lot of problems. It was decided to implement a service file instead to display an alert on top when a user reloads/closes a page.

In this PR I had learned how to write tests for services. I had trouble with triggering the alert modal since it required the page to be reloaded and reloading a window is not permitted while running jasmine tests.

I had to mock the window reload event.

```
var reloadEvt = document.createEvent('Event');
reloadEvt.initEvent('mockbeforeunload', true, true);
```

```
 reloadEvt.returnValue = null;
 reloadEvt.preventDefault = () => {};

// Mocking window object here because beforeunload requres the
 // full page to reload. Page reloads raise an error in karma.
 var mockWindow = {
   addEventListener: function(eventname: string, callback: () => {}) {
     document.addEventListener('mock' + eventname, callback);
   },
   location: {
     reload: () => {
       document.dispatchEvent(reloadEvt);
     }
   }
 };
```

Admin-navbar.directive.ts

I had written a test for this AngularJS directive. Writing a test for this was fairly easy however it was the first time I wrote a new and complete *.spec.ts file for a directive. Writing a test for this was a stepping stone for me to understand how to write a whole new *.spec.ts for a directive.

Here I had to make sure that every possible tab that could be open was tested. This would make the test more robust and potentially reduce the number of bugs that could be created.

```
 it('should be routed to the activities tab by default', () => {
   expect(component.isActivitiesTabOpen()).toBe(true);
   expect(component.isConfigTabOpen()).toBe(false);
   expect(component.isFeaturesTabOpen()).toBe(false);
   expect(component.isRolesTabOpen()).toBe(false);
   expect(component.isJobsTabOpen()).toBe(false);
   expect(component.isMiscTabOpen()).toBe(false);
 });
```

I had also Found a function that was not being used at all. I was able to do this since when I was analyzing the file. I noticed this function was not being utilized anywhere.

```
ctrl.showTab = function() {
 return AdminRouterService.showTab();
```

```
};
```

Milestones

Milestone 1

Key Objective: Write frontend tests to Cover 1650 approximately.

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
1.1	Cover 100 lines		07/06	13/06
1.2	Cover 390 lines		14/06	18/06
1.3	Cover 390 lines		19/06	24/06
1.4	Cover 390 lines		25/06	01/07
1.5	Cover 390 lines		02/07	07/07
1.6	Buffer		08/07	11/07

Milestone 2

Key Objective: Write frontend tests to Cover 1650 approximately.

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
2.1	Cover 390 lines		17/07	21/07
2.2	Cover 315 lines		22/07	26/07
2.3	Cover 315 lines		27/07	31/07
2.4	Cover 315 lines		01/08	05/08
2.5	Cover 315 lines		06/08	10/08
2.6	Buffer time		11/08	16/08

Optional Sections

Future Work

- Make debugging Frontend tests easier