GSoC Proposal 2019 - Oppia

Yash Jipkate

Project Name: Migrate the frontend to Angular 2

Personal Information:

- **Name:** Yash Jipkate
- **Institute:** Indian Institute of Technology, Varanasi, Uttar Pradesh, India
- **E-mail ID:** yashjipkate@gmail.com
- **Github handle: @**YashJipkate

Prior Experience:

I am currently an active member of the institute's Club of Programmers. I have done an institute project in the field of Computer Vision, having employed various ML algorithms like CNN and Image Captioning. I have used various technologies used in the project such as Python, Virtual Machines (both AWS and Azure), Android development and frameworks like Flask, Tensorflow, and Keras. I designed the website of our institute's annual techno-management fest, Technex. It was a Django based website with an Angular frontend hosted on Heroku. I was a part of the team that designed the website for the Club of Programmers of our college which was also a Django based web app.

Why I am interested in working with Oppia:

Oppia's mission to provide education to all in an enjoyable way is something that aroused my interest in the organization. I wanted to use my skills as a programmer to help bring some change in society. The community here is also very helpful in every way possible and are all ready for guidance in every difficulty which I love about the community the most. Apart from this, working with Oppia's codebase has made me learn about a lot of things about Open Source as a whole and the standards of a fully developed organization. I also improved my coding style by a huge degree and learned to collaborate on team projects. I got the opportunity of becoming a full-time member of Oppia and also was the deputy team leader for the Angular Pre-migration project.

What interests me about this project and why it is worth doing:

The project has been of my interest ever since I joined the Angular pre-migration team. I have worked as a deputy leader in the project and got to work closely with the other contributors. While in the team we made the codebase to a pre-migration state, and hence I have a clear idea about the structure of the codebase. Also, since I was the one who took part in the migration of the whole codebase to a pre-migration state, I have a fair amount of experience with AngularJS used in the codebase. Also since the pre-migration also requires that one keeps knowledge of Angular's basics too, I also have a good understanding of the basics of Angular.

It would be a very great experience for me trying hands-on with the full-scale migration of a codebase this vast into a very different framework. The project is worth doing because AngularJS is now no longer maintained and has entered a 3 Year LTS. This would make the codebase obsolete at some point.

Also, the modularity structure and the kind of organization of code that Angular provides is surely a huge advantage of this migration.

Hence there is a need to migrate the codebase to Angular.

My previous contributions:

I am currently a full member of the Oppia organization. I am also the deputy team lead for the Angular Pre-Migration Project team. Apart from this, I am also a member of the Oppia Dev Workflow team and the Bug Fixing Team. I have had many PRs to the codebase and many of them were successfully merged. Some of the PRs are:

- Fix alignment of Exploration Graph legend in history tab (a part of the Bug Fixing Team).
- Successfully load the mobile view in '/parents' page (a blocking bug).
- Added check for prohibited imports
- Custom pylint extension for checking single character files and newline characters.
- Refactoring and clean up backend tests.
- Remove GLOBALS from HTML

The full list of my PRs could be found here

I also opened a number of issues some of which are listed below. The complete list can be found here:

- Increase the reachability of sorted imports check for JS files for the low level controllers.
- Console error during an exploration. (a blocking bug)
- Erroneous navigation in creator dashboard in mobile view.
- Faulty layout of the feedback page in mobile view

Project Plan

The plan is to use an Angular tool called ngUpgrade. This is an *Angular* module and is available as an npm package. That means that it can only be used by an Angular app. However, this module allows AngularJS components and Angular Components to be used side-by-side, as a hybrid app. Or as the Angular team says:

[…] what you're really doing is running both AngularJS and Angular at the same time. All Angular code is running in the Angular framework and AngularJS code in the AngularJS framework.

The major advantage is that old AngularJS components can still be used in the Angular app, while Angular components and services can be *downgraded* to AngularJS components and services. This makes dependency injection still possible.

The steps followed can be broadly classified into these points:

- Setting up the hybrid app
- Refactor constants
- Refactor services and the corresponding tests
- Refactor components and the corresponding tests
- Refactor directives and the corresponding tests
- Phase out AngularJS services
- Remove dependent libraries and remove AngularJS
- Deployment and build

References used:

1. The rough outline of the upgrade is a standard process and is more or less the same in all the guides. The outline followed here in particular (and the sample

code blocks included) are inspired from the Competa guide on upgrading Angular.
2. Official Angular Docs on Angular upgrade

Timeline

Goal	Initial PR	Get merged by
Milestone 1 (May 6-Jun 28)		
The whole codebase comes to a hybrid state i.e. the Angular and AngularJS code are able to run simultaneously.	**May 10**	**May 12**
Refactor the global constants and are in a position to serve both Angular as well as AngularJS parts of the code.	**May 12**	**May 14**
Refactor constants; the constants are separated in a file of their own and are in a position to serve both Angular as well as AngularJS parts of the code.	**May 15**	**May 18**
A sample PR that would remove one service and upgrade the corresponding tests.	**May 17**	**May 19**
Refactor services; the services now are fully Angular. Also, they are in a state to be able to be used in the AngularJS code. The corresponding tests are upgraded too.	**June 22**	**June 28**
Milestone 2 (June 29-July 26)		
A sample PR that would refactor one component upgrading the corresponding tests.	**June 25**	**June 27**

Refactor components; All components are Angular compatible and usable in AngularJS as well. The corresponding tests are upgraded too.	**July 10**	**July 15**
A sample PR that would remove one (or two) AngularJS native service.	**July 12**	**July 14**
A sample PR that would remove one (or two) AngularJS helper service.	**July 13**	**July 14**
Phase out AngularJS services: All AngularJS native services and helper services are replaced by their Angular equivalents or been replaced by a custom implementation.	**July 20**	**July 25**
Milestone 3 (July 27 - August 26)		
A sample PR that would remove one (or two) AngularJS library and replace them with a suitable Angular replacement.	**July 22**	**July 24**
Remove all remaining dependent libraries and remove AngularJS: All AngularJS dependent third party libraries are replaced with their Angular counterparts or had a custom implementation for the same.	**August 12**	**August 18**
Deployment and build: All the AngularJS code has been cleaned and the hybrid bootstrapping is removed in favor of a fully Angular app. The Angular app builds successfully.	**August 19**	**August 25**

Technical Details and Implementation.

- **Setting up the hybrid app:**

We currently have `ng-app="oppia"` in base.html. This would be removed and instead the app would be bootstrapped to have both AngularJS and Angular. The starting point/entry point (this is required to be defined in the Webpack config) of the app would be one main.ts which would look like -

```
import { platformBrowserDynamic } from
'@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
platformBrowserDynamic().bootstrapModule(AppModule);
```

The app.module would look like:

```
import { UpgradeModule } from '@angular/upgrade/static';

@NgModule({
  imports: [
    BrowserModule,
    UpgradeModule,
  ],
})
export class AppModule {
  constructor(private upgrade: UpgradeModule) { }
  // This is to bootstrap the AngularJS part.
  ngDoBootstrap() {
    this.upgrade.bootstrap(document.documentElement, [‘oppia’]);
  }
}
```

The `oppia` module of AngularJS is defined in `app.module.ajs.ts` which looks like:

```
declare var angular: angular.IAngularStatic;
angular.module(
 'oppia', [
   'angularAudioRecorder', 'dndLists', 'headroom', 'infinite-scroll',
   'ngAnimate', 'ngAudio', 'ngCookies', 'ngImgCrop', 'ngJoyRide', 'ngMaterial',
```

```
  'ngResource', 'ngSanitize', 'ngTouch', 'pascalprecht.translate', 'toastr',
  'ui.bootstrap', 'ui.sortable', 'ui.tree', 'ui.validate'
].concat(
  window.GLOBALS ? (window.GLOBALS.ADDITIONAL_ANGULAR_MODULES || []) : []));
```

.This will allow both AngularJS and Angular to run simultaneously in our app. This is known as 'hybrid app'.

- **Refactoring constants:**

  AngularJS constants and values do not exist anymore in Angular. The constants are used in both the AngularJS and the Angular parts of the code. The constants currently in use are initialized in their respective controller files. The global constants i.e. the ones that are used across modules are defined in App.js. These constants would be stored in a file (say *oppia.constants.js*). The constants of belonging to a particular module would be stored in a file (*<name-of-module-in-kebab-case>.constants.js*).

  In the file, for every constant or value, there would be a member of the class `Constants`:

  (this will serve the constants for Angular)

```
export class Constants {
  public static DEVELOPMENT_MODE = true;
  public static GRAPHS_API = 'http://my.site.com/api';
  public static UPDATE_INTERVAL_DATA = 10000;
}
```

  For each of these constants, create an AngularJS constant in the same file:

  (This would serve the constants in AngularJS)

```
declare var angular: angular.IAngularStatic;
angular
```

```
  .module('oppia')
  .constant('DEVELOPMENT_MODE', Constants.DEVELOPMENT_MODE)
  .constant('GRAPHS_API', Constants.GRAPHS_API)
  .constant('UPDATE_INTERVAL_DATA', Constants.UPDATE_INTERVAL_DATA)
```

Note that here `DEVELOPMENT_MODE, GRAPHS_API, UPDATE_INTERVAL_DATA` are examples of some constants totally irrevelant to the existing constants in the codebase.

This makes sure that the constants can be used in both the old AngularJS code, as well as the Angular app.

- **Refactoring services, factories and providers:**

  Services are dependency injections for components (controllers in AngularJS) and other services. Because they are dependencies it is wise to refactor them first. However, there is also an order to consider when refactoring. Like components, services also can inject dependencies. Therefore, we would first refactor the services that have the *least dependencies*. This will save a lot of overhead. Currently (at the time of writing this proposal) there are a total of 308 services which are dependent on each other and the native Angular services and some are independent. The independent ones would be refactored first and then the dependent ones.

  The strategy adopted to refactor a service:

  1. Import Injectable, Inject (angular core), downgradeInjectable from ngUpgrade

```
import { Injectable, Inject } from '@angular/core';
declare var angular: angular.IAngularStatic;
import { downgradeInjectable } from '@angular/upgrade/static';
```

  2. Refactor function into class:

```
// AngularJS
function NewService ($scope) {
```

```
// Angular
@Injectable()
export class NewService {
```

3. For all AngularJS services (either own services or native AngularJS services), inject them in in the constructor of the class using the @Inject() decorator:

```
constructor (
   @Inject('$q') private $q,
   @Inject('MyOldService') private myOldService,
   Private myNewService: MyNewService // this is a typescript service
) {
  // Do constructor stuff
}
```

For all AngularJS services, we need to add them as a provider in app.module.ts. This is needed to use AngularJS services in Angular Services and components:

```
@NgModule({
  providers: [
   {
      provide: '$q',
      useFactory: ($injector: any) => $injector.get('$q'),
      deps: ['$injector']
  },
  {
      provide: '$MyOldService',
      useFactory: ($injector: any) => $injector.get('$MyOldService
'),
      deps: ['$injector']
  },
  ],
  ...
```

```
})
```

4. Refactor AngularJS factory:
    a. Make all functions that are returned from function public: Change `return { myPublicFn }` to `public myPublicFn`
5. Add types to variables, parameters, return values of functions using typescript Interfaces.
6. Downgrade the new Angular service to an AngularJS service using downgradeInjectable at the end of the file:
    a. Alternative: import newService into app.module.ajs.ts and downgrade it there.

```
angular.module('oppia')
    .service('newService', downgradeInjectable(NewService));
```

7. Register service as a provider in Angular module (app.module.ts):

```
import { NewService } from './service.new';
@NgModule({
  providers: [ NewService ],
  ...
})
```

8. Find where new service is used in other services in .ts files. Remove @Inject annotation from constructor in those .ts files. Import them as a regular Angular service.
    a. If the old AngularJS service was registered, remove provider from app.module.ts that uses the AngularJS $injector. (not breaking if not done, but cleaner)

- **Refactor components**

    Like services, the best strategy to refactor components is to start with the smallest components. That is because AngularJS element components are difficult to use inside Angular component templates. The second rule

of thumb is to refactor the components with the least dependencies such as bindings and requires. The basic idea of refactoring an AngularJS component to an Angular component is to use the hybrid approach. The refactored component is written in Angular but will be *both available as an AngularJS component as well as an Angular component.* To do that, the refactored Angular component will have to be *downgraded* to be used as an AngularJS component. The following steps may be taken to refactor a component:

1. Remove all IIFE's
2. Change controller function to component class and export it: `function NewController => export class NewComponent`
3. Import Component from @angular/core
4. Decorate Class with @Component

```
import { Component, Inject, OnInit, OnDestroy } from '@angular/core';
@Component({
  selector: 'my-new-component',
  templateUrl: './component.new.html'
})
export class NewComponent implements OnInit, OnDestroy {
```

5. Create constructor for class and add all AngularJS services and Angular Services:

```
constructor (
        @Inject('$http') public $http: any,
    private myService: MyService
) { }
```

6. Register AngularJS services in *app.module.ts*:

```
@NgModule({
  providers: [
   {
```

```
    provide: '$q',
    useFactory: ($injector: any) => $injector.get('$q'),
    deps: ['$injector']
  },
```

7. Downgrade this new Angular Component to service to AngularJS directive using downgradeComponent at the end of the file. Now, the component can be used as both as an Angular and AngularJS component:

```
declare var angular: angular.IAngularStatic;
import { downgradeInjectable } from '@angular/upgrade/static';

export class MyComponent { ... }

angular.module('app').directive('myComponent',
downgradeComponent(MyComponent));
```

- **Refactor Unit Tests/E2E Tests:**
  - E2E Tests:

    By definition, E2E tests access the application from the outside by interacting with the various UI elements the app puts on the screen. E2E tests aren't really that concerned with the internal structure of the application components. That also means that, although the project is modified quite a bit during the upgrade, the E2E test suite should keep passing with just minor modifications since we didn't change how the application behaves from the user's point of view.

    During hybrid bootstrapping, a small addition needs to be done in `protractor.conf.js` to sync with hybrid apps:

```
ng12Hybrid: true
```

    Also, the native AngularJS services that are being used if any in the test files need to be replaced with their Angular compatible equivalents.

- Unit Tests:

  Unlike the E2E tests, they need to be upgraded along with the production code. Here too, the native AngularJS services that are being used if any in the test files need to be replaced with their Angular compatible equivalents and the code needs to be refactored accordingly.

  The AngularJS dependency services, if any, are made injectable to the upgraded Angular unit test.

  Reference here

```
// upgrading angular js service to make it possible to inject into
angular services/components
const UpgradedProvider: FactoryProvider = {
    provide: AngularJSService,
    useFactory: ($injector: angular.auto.IInjectorService) =>
{$injector.get('AngularJSService')},
    deps: ['$injector']
};
```

- **Phase out AngularJS services.**
  AngularJS has a number of helper functions that can be used as helpers in javascript code. For each method, a replacement has to be found in Angular. These methods are bound to the global angular object. The ones used in our codebase are:

Service	Replacement
angular.isDefined	`typeof x !== 'undefined'`
angular.forEach	`forEach` method of ES6 `.forEach(function(val,index){}`
angular.copy	TS shorthand `const copy = { ...original }`
angular.isObject	`x != null && typeof x === 'object'`

angular.isString	`typeof x === 'string'`
Angular.extend	`jQuery.extend({}, object);`
angular.isUndefined	`typeof o === 'undefined'`
angular.isNumber	rxjs/util/isNumeric
angular.isArray	`Array.isArray(value)`
angular.isFunction	`typeof x === 'function'`
Angular.equals	`_.isEqual(value, other)` (Lodash)
angular.toJson	`JSON.stringify(x)`
angular.fromJson	`JSON.parse(x)`
angular.element	The uses of angular.element and their possible replacements are: • Adding event listeners to the document object: Using `HostListener` • Sanitize HTML: Can be achieve by using `DOMSanitizer`. • To prepare content for `$compile`, dynamic template rendering: These instances would be removed since we won't be using `$compile` anymore. • To add/remove class: This can be achieved by using the `@HostBinding` decorator. (Reference)

Replace the native AngularJS services with their respective equivalents. The native services used in our codebase are:

Service	**Possible replacement**
$anchorScroll	There are two types of scrolling by $anchorScroll in the codebase. • Custom implementation using angular router for hash location scrolling(Reference for hash location scrolling) • Using JS method `window.scrollTo` for scrolling to yOffset.
$templateCache	CachedResourceLoader (Reference) The $templateCache is used in the codebase for two reasons: • To retrieve the joyride template. This can be achieved by `window['$templateCache'] = { 'template.html': `...`};` • To retrieve the templates from cache in Spec files. This can be done by the `.get()` method of `CachedResourceLoader`
$compile	Used for: • Building templates in runtime: Can be done by custom implementation using `compileModuleAndAllComponentsAsync` method of Compiler from @angular/core (Reference) • Testing directives: This would not be needed in Angular as the way to test directives is totally different in Angular 2.
$document	(Reference) Use the @HostListener decorator.
$exceptionHandler	ErrorHandler from @angular/core
$filter	Own implementation (@pipe)

$http	HttpClient
$httpBackend	HttpBackend
$interval	rxjs/observable/interval
$location	@angular/common (Location)
$log	Angular-logger (reference implementation)
$q	Native ES6 (Promise)
$rootScope	The usages of $rootScope and their replacements are as follows: • Broadcasting events(`$rootScope.$broadcast`): This can be achieved by using EventEmitter from node.js (Reference) • Changing global variables: This can be achhieved by using a services that is injected in the bootstrapping function to make it available to the whole app. The service would use rxjs Observable to update values. (Reference)
$sce	This is used to bypass sanitization of HTML and URLs. This can be achieved by the `bypassSecurityTrustHtml`, `bypassSecurityTrustResourceUrl` methods of `DomSanitizer`.
$timeout	rxjs/observable/timer
$window	Custom Implementation (reference)

$compileProvider	Used only for disabling debug logging. This can be done by enableProdMode. The method is called just before calling bootstrap.
	
	`import {enableProdMode} from '@angular/core';`
	`enableProdMode();`

	`bootstrap(....);`
$httpProvider	@angular/http
$interpolateProvider	Use interpolation option in Component metadata. (Reference)
$locationProvider	PathLocationStrategy
$cookiesProvider	Used to configure `$cookies` service. But since the `$cookies` service is going to get replaced, this will not be needed
$cookies	Not used directly, must be used as a dependency. Would be removed as the new libraries won't depend on it anymore.

- **Remove dependent libraries and remove AngularJS**

  If all AngularJS services, controllers, services, etc. have been refactored to their Angular equivalents, the last step is to replace all AngularJS dependent libraries, such as angular-ui-bootstrap. Firstly, it may be that these libraries **do not have an Angular implementation!** In that case, a new Angular compatible library has to be found and implemented. This, of course, comes with certain risks:

  - The replacing library may visually be different
  - The replacing library may functionally be different

  Clearly, changes in functionality are unwanted. Therefore, keep functional requirements and or specifications in mind when replacing. After replacing, the AngularJS dependent (npm) package is removed and also from the `app.module.ajs.ts`

  The above two steps would require some amount of collaboration with the other teams of Oppia. For example, the `angularAudioRecorder` library which takes care of audio recordings is related to Audio Translations Team

and would require some coordination from their side (i.e. the purpose it serves in the codebase and the way it is used).

Also, if anyone is selected in the upgrade third party GSoC project, collaborate with them to ensure that they don't end up in upgrading a library that would be removed.

Library	Replacement
angularAudioRecorder	Record RTC. Reference here. RecordRTC is a JavaScript-based media-recording library instead of being an Angular library.
dndLists	ng2-dnd
headroom	ngx-headroom
ngAudio	Howler.js (Reference)
infinite-scroll	angular2-infinite-scroll
ngAnimate	@angular/animations
ngImgCrop	ng2-img-cropper
ngJoyRide	ngx-joyride
ngMaterial	@angular/material
ngResource	No usages found, will be removed instead of replacement.
ngSanitize	Angular Sanitizer
ngTouch	Hammerjs
pascalprecht.translate	@ngx-translate/core
toastr	ngx-toastr
ui.bootstrap	ngx-bootstrap
ui.sortable	ng2-dnd

ui.tree	angular-tree
ui.validate	Angular validation

Implementation details:

- angularAudioRecorder:
    - Usage: `core/templates/dev/head/pages/exploration_editor/translation_tab/AudioTranslationBarDirective.js`
    - Replacement (Reference):
        - Use RecordRTC, a Javascript-based media-recording library.
        - The functions used are replaced by those of RecordRTC.
            - Status of recording (`recorder.status`): `getState(), state`
            - Pause recording (`recorder.playbackPause()`): `pauseRecording`
            - Stop recording (`recorder.stopRecord()`): `stopRecording`
            - Start recording (`recorder.startRecord()`): `startRecording`
- dndLists:
    - Usage:
        - `core/templates/dev/head/pages/topic_editor/subtopics_editor/subtopics_list_tab_directive.html`
    - Replacement (Reference):
        - Use ng2-dnd, the Angular 2 Drag-and-Drop.

        - Import the `style.css` into the web page from `node_modules/ng2-dnd/bundles/style.css`
        - The basic way of usage is the same, i.e. the attributes that used to be in dndLists, with some minor changes that were needed to adjust it with Angular 2.
- Headroom:
    - Usage:
        - `core/templates/dev/head/pages/base.html`
    - Replacement (Reference):
        - Use ngx-headroom.
        - Requires `@angular/animations`
        - Use like so-

```
<ngx-headroom>
  <h1>You can put anything you'd like inside the Headroom Component</h1>
</ngx-headroom>
```

        - This is a demo provided by the ngx-headroom team. The transition time can be altered by the `duration` property.
- ngAudio
    - Usage:
        - `core/templates/dev/head/services/AudioPlayerService.js`
    - Replacement (Reference):
        - Use howler.js.
        - The functions are replaced by those if howler.

- `.stop()` -> `stop()`
- `.play()` -> `play()`
- `.pause()` -> `pause()`
- `.progress` -> `seek()` divided by `duration()`
- `.duration` -> `duration()`
- `.currentTime` -> `seek()`
- `.paused` -> `playing()`

- Infinite-scroll
  - Usage:
    - `core/templates/dev/head/pages/library/activity_tiles_infinity_grid_directive.html`
  - Replacement (Reference):
    - Use angular2-infinite-scroll
    - Replace the attributes with those of angular2-infinite-scroll:
      - `Infinite-scroll` -> `scrollWindow`
      - `Infinite-scroll-distance` -> `infiniteScrollDistance`
      - `Infinite-scroll-disabled` -> `infiniteScrollDisabled`

- ngAnimate
  - Usage:
    - `core/templates/dev/head/pages/exploration_player/audio_bar_directive.html`

        - `core/templates/dev/head/pages/exploration_player/conversation_skin_embed_directive.html`
        - `core/templates/dev/head/pages/exploration_player/supplemental_card_directive.html`
    - Replacement (Reference):
        - Use @angular/animations
        - The way Angular does it is that it defines a trigger, a start state, an end state, and the transition from start state to end state and vice versa.
        - A sample taken from the Angular animation docs:

```
animations: [
   trigger('openClose', [
     // ...
     state('open', style({
       height: '200px',
       opacity: 1,
       backgroundColor: 'yellow'
     })),
     state('closed', style({
       height: '100px',
       opacity: 0.5,
       backgroundColor: 'green'
     })),
     transition('open => closed', [
       animate('1s')
     ]),
     transition('closed => open', [
       animate('0.5s')
     ]),
   ]),
```

```
],
```

        - The `ng-enter` and `ng-leave` (and their `active` counterparts) are to be replaced by the open and closed states respectively and the trigger would be called on the event.
        - The `active` suffix would now have no meaning. The suffix was added by AngularJS when the event is underway. But since Angular manages all the transitions by the animations trigger, therefore there would be no need to define `active`'s CSS properties.

- ngImgCrop
    - Usage:
        - `core/templates/dev/head/pages/preferences/edit_profile_picture_modal_directive.html`
    - Replacement:
        - Use [ng2-img-cropper]
        - ([Reference]) The image to be cropped is set by the `ImageCropperComponent` in a child view.

```
@ViewChild('cropper', undefined)
cropper:ImageCropperComponent;
```

The image can be set using `cropper.setImage(image);`

The result image can be stored by specifying the name of the final image as a string in the `[image]` attribute.

The default cropping shape is square. Can also do `cropperSettings1.rounded = false`

`Result-image-size -> croppedWidth, croppedHeight`

`Result-image-format -> fileType`

- ngJoyRide
    - Usage:
        - `core/templates/dev/head/pages/exploration_editor/exploration_editor.html`
        - `core/templates/dev/head/pages/exploration_editor/translation_tab/translation_tab_directive.html`
    - Replacement (Reference):
        - Use ngx-joyride
        - The library works in a different way than ngJoyRide. The steps in the tour/tutorial is not defined by a config dict list type variable, but the steps of the tour are marked as attributes in the CSS class of the step like so.

```
  template: `<div joyrideStep="joy1" title="title" (prev)="onPrev()"
(next)="onNext()">Hello!</div>
            <div joyrideStep="joy2" title="title2"
(done)="onDone()">Hello!</div>`
```

The steps are then registered like so:

```
  onClick() {
```

```
    this.joyrideService.startTour(
      { steps: ['firstStep', 'secondStep']}r
    );
  }
```

There is no `on-skip` in ngx-joyride. Instead the `done` event can be used for the skip button.

Unlike ngJoyRide, the on-skip and on-finish events are not defined in the container. These are defined in every element. In other words, the finish/skip is defined by a step rather than the whole attribute.

We can also use custom template that would be the same as the current tour templates using the `stepContent` input.

- ngMaterial
    - Usage: widespread
    - Replacement (Reference)
        - Use Angular Material
        - Need to import `BrowserAnimationsModule`.
        - A theme must be included (as stated here)
        - Gesture Support using HammerJS.
        - Addition of material icons (`mat-icons` instead of `md-icon`)
            - `Md-input-group -> mat-form-field`
            - `Md-card -> mat-card`
            - `Md-button -> button`
            - `Md-progress-linear -> mat-progress-bar`

        - `Md-slider -> mat-slider`
        - `Md-checkbox -> mat-checkbox`
- ngResource
    - No usages found. This means that it is a dependency for other library.
    - SInce all the libraries would be replaced with new ones, the ngResource will be simply removed.
    - There would not be any need for a replacement library.
- ngSanitize
    - Usage:
        - `core/templates/dev/head/components/forms/ConvertUnicodeToHtmlFilter.js`
    - Replacement (Reference):
        - Use `DomSanitizer`
        - The text can be sanitized by using the `sanitize()` function of DomSanitizer
        - A sample would look like:
          `.sanitize(SecurityContext.HTML, this.text)`
- ngTouch
    - Usage: No direct usage
    - Since this library has no direct usage, this means that the library is a dependency to another library. Since we would be replacing the libraries with new ones, this library would be removed.
- Pascalprecht.translate
    - Usage:

        - `core/templates/dev/head/I18nFooter.js`
    - Replacement:
        - Use @ngx-translate/core
        - This would be imported as `TranslateModule.forRoot` in the main app module since this is used in `base.html` which is the root template (see this).
        - To load the i18n json file, `TranslateHttpLoader` would be used.
        - The docs recommend to use Http from @angular/http with http-loader@0.1.0 for Angular 2 for the `TranslateHttpLoader`.
        - A sample for `TranslateHttpLoader` implementation :

```
TranslateModule.forRoot({
    loader: {
        provide: TranslateLoader,
        useFactory: HttpLoaderFactory,
        deps: [HttpClient]
    }
})
```

- Toastr
    - Usage:
        - `/core/templates/dev/head/componentsalerts/AlertMessageDirective.js`
    - Replacement:
        - Use ngx-toastr

        - This would be imported as `ToastrModule.forRoot()` in the main app module since this is used in `base.html` which is the root template (see this).
        - The methods are similar to its Angular counterpart.

- Ui.bootstrap
    - Usage: widespread
    - Replacement:
        - Use ngx-bootstrap
        - The replacements of various components are as follows:
            - `Uib-dropdown -> dropdown`
            - `Uib-popover -> popover`
            - `Uib-tooltip -> tooltip`
            - `Uib-tabset -> tabset`
            - `Uib-accordion -> accordion`
- Ui.sortable
    - Usage:
        - `core/templates/dev/head/pages/exploration_editor/param_changes_editor_directive.html`
        - `core/templates/dev/head/pages/learner_dashboard/learner_dashboard.html`
        - `core/templates/dev/head/pages/skill_editor/editor_tab/skill_concept_card_editor_directive.html`

        - `core/templates/dev/head/pages/state_editor/state_hints_editor_directive.html`
        - `core/templates/dev/head/pages/state_editor/state_responses_directive.html`
    - Replacement:
        - Use ng2-dnd
        - (Reference)The data for the list is populated by `[sortableData]` instead of `ng-model`.
- Ui.tree
    - Usage:
        - `extensions/interactions/DragAndDropSortInput/directives/drag_and_drop_sort_input_interaction_directive.html`
    - Replacement:
        - Use angular-tree
        - (Reference) Instead of `ng-model` it uses nodes to populate its data. A typical nodes list would look like:

```
nodes = [
    {
      id: 1,
      name: 'root1',
      children: [
        { id: 2, name: 'child1' },
        { id: 3, name: 'child2' }
      ]
    },
```

```
  {
    id: 4,
    name: 'root2',
    children: [
      { id: 5, name: 'child2.1' },
      {
        id: 6,
        name: 'child2.2',
        children: [
          { id: 7, name: 'subsub' }
        ]
      }
    ]
  }
];
```

- Ui.validate
    - Usage:
        - `core/templates/dev/head/components/forms/schema_editors/schema_based_float_editor_directive.html`
        - `extensions/objects/templates/filepath_editor_directive.html`
        - `extensions/objects/templates/parameter_name_editor_directive.html`
    - Replacement:
        - Use the Angular validation (in-built in Angular)
        - (Reference) Custom validators can be made. A directive is to be added to the template that handles the validations.

- **Deployment and build**

  At this point, there should be no AngularJS files left to be refactored or to be replaced. That means that the only hybrid bootstrapping is in `app.module.ts`. The UpgradeModule import is removed::

```
@NgModule({
...
})
export class AppModule { }
```

  Then, in main.ts, remove the require.context functionality that is used to load the AngularJS files as modules (for webpack):

```
import { platformBrowserDynamic } from
'@angular/platform-browser-dynamic';
import { enableProdMode } from '@angular/core';
import { environment } from './environments/environment';
import { AppModule } from './app/app.module';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

  Then, all the downgraded components and services will be removed from the Angular component and services:

```
declare var angular: angular.IAngularStatic; // This line is to be
removed
import { downgradeInjectable } from '@angular/upgrade/static'; //
This line is to be removed
import * as angular from 'angular'; // remove this line
```

```
@Injectable()
export class NewService {
  ...
}

angular.module('app') // This line is to be removed
    .service('newService', downgradeInjectable(NewService)); // This
line is to be removed
```

Also, the `protractor.conf.js` had an addition due to the hybrid which needs to be removed. This line would be added instead.

```
useAllAngular2AppRoots: true,
```

This concludes the migration process. The codebase is now fully Angular!

Which timezone(s) will I primarily be in during the summer?

I will be in India (UTC+05:30) throughout the summer.

How much time will you be able to commit to this project?

I will be having my summer vacations for most of the period with the exception being the last couple of weeks and therefore I can easily devote 8-10 hours every day. But since I won't have exams so I still can devote at least 5-6 hours during weekdays and the usual 8-10 hours during weekends.

Devlogs

I shall maintain daily devlogs and share the same with my mentor which will help me and my mentor to keep track of my daily progress.

What jobs, summer classes, and other obligations might you need to work around?

There are no jobs, summer classes or any type of other obligations during this summer with the only exception being the last week of July and the following weeks of August in

which I would be having my classes.

Means of Communication:

I am comfortable with all kinds of communications like email (yashjipkate@gmail.com), Gitter, Hangouts. I will try to be in constant touch with my mentors through the methods mentioned above or any other method the mentor desires as I am open to any kind of mode of communication.