University of Victoria
Engineering & Computer Science Co-op
Work Term Report
Summer 2020 Year

Migration from Angular.JS to Angular

SKIO Music
Full-stack Development
Victoria, British Columbia, Canada

Trishala Bhasin
V00911367
1st Work Term
MSc Computer Science
trishalabhasin@uvic.ca
August 24,2020

In partial fulfillment of the academic requirements of this co-op term

Supervisor's Approval: To be completed by Co-op Employer

This report will be handled by UVic Co-op staff and will be read by one assigned report marker who may be a co-op staff member within the Engineering and Computer Science Co-operative Education Program, or a UVic faculty member or teaching assistant. The report will be retained and available to the student or, subject to the student's right to appeal a grade, held for one year after which it will be deleted.

I approve the release of this report to the University of Victoria for evaluation purposes only.

Signature:  Position: CTO Date: August 24, 2020

Name (print): Dan Nesbitt E-Mail: dan.n@skiomusic.com

For SKIO Music

Acknowledgement

I would like to thank the Hiring Team of SKIO Music who gave me the golden opportunity to be a part of an amazing development team and make me a part of their platform development, which also helped me in doing a lot of Research and I came to know about so many new things I am really thankful to them. I would like to express my special thanks to my supervisor at SKIO Music **Mr. Dan Nesbitt** who guided me through this project and supported me through various stages of my co-op.

I would like to thank my UVic on Co-op coordinators **Ms. Cheryl Beaumont** and **Mr. Ahmad Salman** who supported me through the co-op duration. I would also like to thank my teammates who helped me navigate through the company work environment and relied on me with major front-end tasks. Lastly, I would like to thank my family and friends who believe in me and keep me motivated through all my stressful days.

List of Tables

List of Diagrams

Abstract

In this report I will be discussing the research that I conducted in order to migrate the SKIO Music front-end platform from AngularJS to Angular 2+ version. I will begin with the motive behind the shift by highlighting the key strengths of each and highlight the main advantages of Angular over AngularJS. I will also be looking at the approach that I plan on taking in order to carry out the shift along with the discussions with my development team around the steps that we will be following in order to do the shift.

Index

1. Introduction

AngularJS was one of the initial languages that was being used in order to make interactive web pages. Launched in 2010 by Google, It was quickly and widely accepted by software developers as it was an extension of Javascript, that they were already familiar with. As AngularJS was gaining its popularity, there were other interactive languages coming in the developer market such as ReactJS and VueJS. These languages sparked a debate among the software developers between AngularJS and the other new languages for interactive web development. With increasing popularity of other languages, Google completely rewrote AngularJS (also known as Angular 1) to Angular 2 which used Typescript as a new language. TypeScript was introduced in the Angular 2 + in order to avoid the pitfalls of JavaScript and to introduce a small amount of static typing, a feature that many existing web developers were looking for in the dynamic web development.

2. Angular.JS for Front-end Development

AngularJS is being actively used in SKIO Music's frontend development. There were a variety of reasons why the platform was being built on AngularJS, familiarity and ease of development being the top reasons. We have listed a few more reasons that made AngularJS the choice of language in order to design the front-end.

1. Being based on JavaScript, it is much easier and faster to learn AngularJS.

2. AngularJS two-way data binding facilitates faster and easier data binding without the intervention of the developers.

3. AngularJS supports faster coding and prototyping, decreasing development times immensely.

4. The MVC and MVVM architecture of AngularJS separates data from design, which makes it easier to develop and maintain complex web applications.

5. Clean and organized coding makes AngularJS codes highly reusable.

Along with these developmental advantages, the two way data binding abilities and ease of development, the team is facing some challenges that are being listed below.

2.1. Challenges faced in the migration from Angular.js to Angular

1. If the system trying to run the AngularJS application has JavaScript disabled, the application will not run on it.
2. The developers must be familiar with MVC architecture to use AngularJS.

Thus we plan on further investigating the shift to Angular along with comparing it with other options such as React.

3. Using Angular vs Angular.JS

Angular and AngularJS have advantages and disadvantages of their own. In the previous section, we looked at the advantages of using AngularJS and how it has helped in forming the front-end of the application. Now we will be discussing some of the advantages of using Angular 2+ from a development point of view in

order to understand the benefits of migration more easily. The advantages of using Angular are as enlisted below.

3.1 Advantages of using Angular

1. Angular is at least five times faster than AngularJS due to a much better algorithm for data binding and a component-based architecture.

2. The components of an Angular application are quite independent and self-sufficient, which makes them reusable and test friendly.

3. The independent components are easier to replace, maintain, and scale-up.

4. Angular applications can be rendered both on browsers and mobile devices.

5. Angular has inbuilt extensions for server-side rendering of applications. This enables developers to sync client and server sides of content, a huge plus for SEO.

6. Angular supports lazy loading, which makes the applications faster as only those components that are needed are loaded.

7. TypeScript's first approach of Angular makes for cleaner code, better navigation, and high-quality product

Despite the advantages that Angular has to offer, there are some disadvantages that are usually faced by the developers who are used to working with Angular.JS and other older front end frameworks. These observations are based on the developer experience at SKIO.

3.2. Disadvantages of using Angular

1. The learning curve of Angular is steep so we also need to master typescript, a statically typed language.

2. With the advent of frameworks that enable quick development, many developers are not competent with statically typed languages.
3. As Angular 2 was a complete rewrite from AngularJS, legacy systems developed using AngularJS need to be migrated, which is something many developers do not like.
4. Angular is also sometimes called a verbose language because the components are managed in a very complex way.
5. The command-line interface of Angular is much loved by the engineers but they also complain that its documentation is not complete.

4. Differentiating factors between Angular and Angular.JS

Now, let us look at the key differences between Angular and AngularJS in terms of key aspects of current full-stack development.

Software Aspect	**Angular Vs Angular.JS**
1. Architecture	AngularJS supports the MVC or Model View Controller architecture. According to this architecture design, we put the business logic in the model, the desired output in the controller, and Angular does all the processing to derive that output. The model pipelines are automatically generated by AngularJS. By contrast, components and

	directives form the building block of Angular. Components are nothing but directives with a predefined template. They provide a modern structure to the applications, making it easier to create and maintain larger applications.
2. TypeScript	As mentioned earlier, AngularJS uses JavaScript but Angular 2 and later versions (clubbed together as Angular 2+ for the sake of discussions) use TypeScript. TypeScript is the superset of JavaScript and provides static typing during the development process. Static typing not just improves performance but avoids many runtime pitfalls that were making AngularJS difficult to use for larger and complex applications.
3. Dependencies	Both AngularJS and Angular use of dependency injection but the way they do are completely different. AngularJS dependencies are injected into various link functions, controller functions, and directive definitions. On the other hand, Angular implements a hierarchical dependency injection system using declarations, constructor functions, and providers.
4. Angular CLI	Angular 2+ ships with their own command-line interface or CLI. It is used for generating components, services, etc. and even complete projects quickly and efficiently. You can easily generate different versions of the same project for different platforms with dynamic type checking, linting, etc. AngularJS does not have a CLI of its own.
5. Expression Syntax	When it comes to dealing with data binding, Angular is more intuitive than AngularJS. An AngularJS developer must remember the correct ng directive for binding a property or an event. In the case of Angular, the language uses () for event binding and [ ] for property binding.
6. Performance	Angular is much faster than AngularJS. A lot of developers have claimed that if built correctly Angular applications can be up to five times faster than AngularJS

	applications. The concept of two-way binding, which made the original AngularJS popular among web developers has proved to be its undoing as more and more complex applications are being developed using it. To ensure and implement two-way binding, AngularJS keeps checking each scoped variable with its previous value using a digest cycle. The running of this digest cycle is random and hence as the size of the program grows, the checking can go on infinitely, impacting application performance.In contrast, Angular has a flux architecture where change detection is done through unidirectional data flow, making applications much faster.
7. Mobile Support	AngularJS does not provide mobile development support but Angular does. AngularJS is a bit old in this age of mobile-first computing thus the libraries do not support mobile development.

Table 1: Differentiating factors between Angular and Angular.JS

5. Working Diagrams of Angular and Angular.JS

In order to understand the difference between the two languages, it is necessary to look at their diagrams and how the interaction between various components in the application happens.

5.1 Angular.JS Diagram

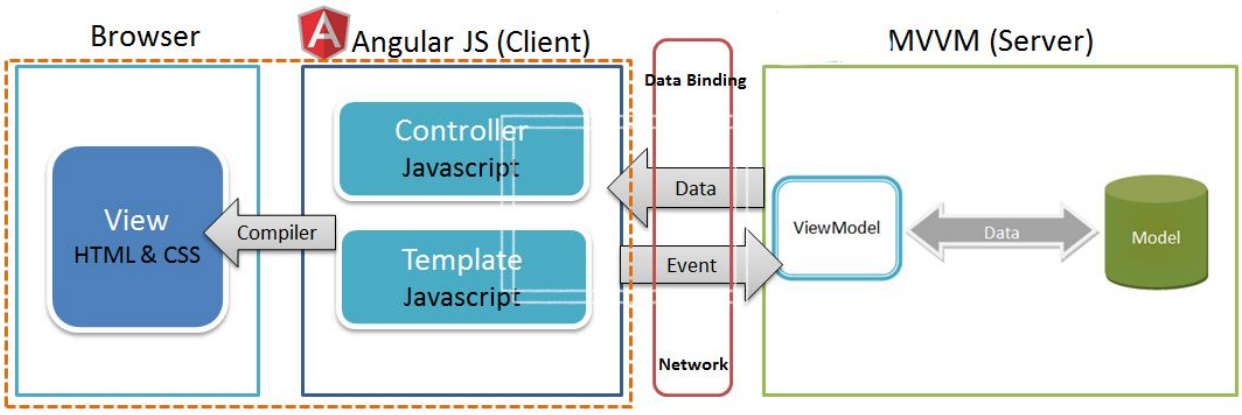

Figure 1: Angular.JS Architecture

The key components in the Angular.JS are as follows. Their placement in the whole architecture of the front-end of an application is shown above:

1. **Controller** represents the layer that has the business logic. User events trigger the functions which are stored inside your controller. The user events are part of the controller.
2. **Views** are used to represent the presentation layer which is provided to the end users.
3. **Models** are used to represent your data. The data in your model can be as simple as just having primitive declarations. For example, if you are maintaining a student application, your data model could just have a student id and a name. Or it can also be complex by having a structured data model. If you are maintaining a car ownership application, you can have structures to define the vehicle itself in terms of its engine capacity, seating capacity, etc.

5.2 Angular Diagram

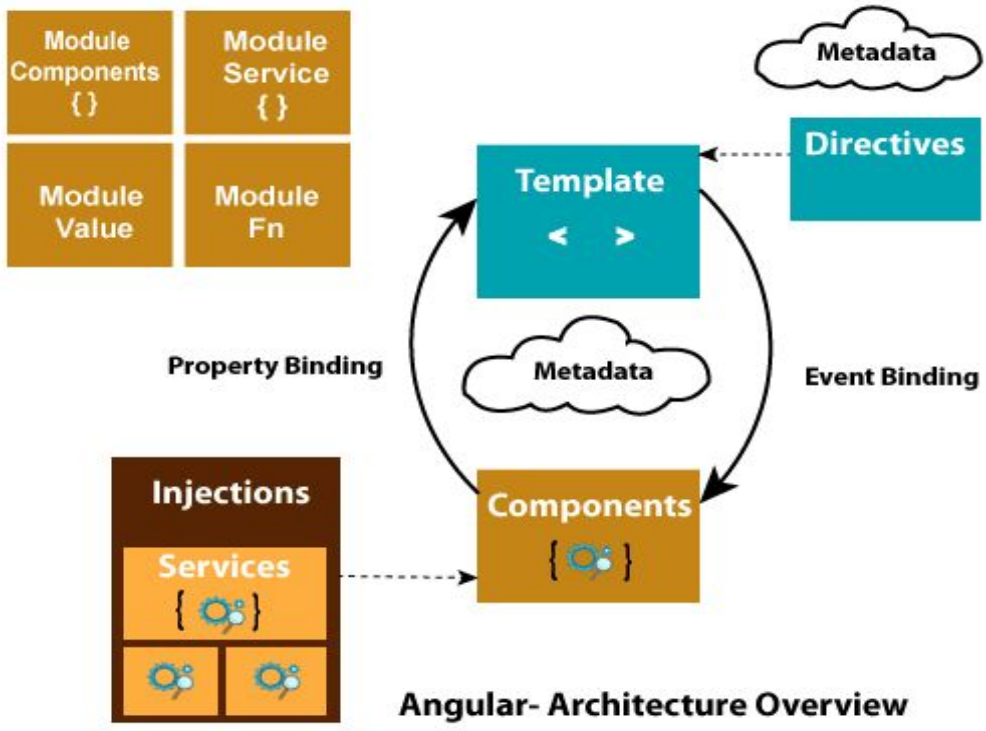

Figure 2: Angular Architecture

1. **Modules:** Angular Modules are different from other JavaScript modules. Every Angular app has a root module known as AppModule. It provides the bootstrap mechanism that launches the application.

2. **Components:** Components and **Services** both are simply classes with decorators that mark their types and provide metadata which guide Angular to do things. Every Angular application always has at least one component known as root component that connects a page hierarchy with page DOM. Each component defines a class that contains application data and logic, and is associated with an HTML template that defines a view to be displayed in a target environment.

3. **Dependency Injection:** Dependency Injection (DI) is used to make your component classes lean and efficient. DI doesn't fetch data from the server,

validate user input, or log directly to the console; it simply renders such tasks to services.

6. Migration Process

There are various migration paths that can be taken in order to carry out the shift from Angular.JS to Angular. Since it is a very popular shift that is being done extensively in the industry today, our team has decided to use the 'Hybrid approach' along with using helper libraries in order to carry out the shift. We planned not to do it all at once but to do it parallelly as we do not want to break the front-end production due to unknown dependencies. Below is a diagram showing the different ways in which migration can happen from Angular.JS to Angular.

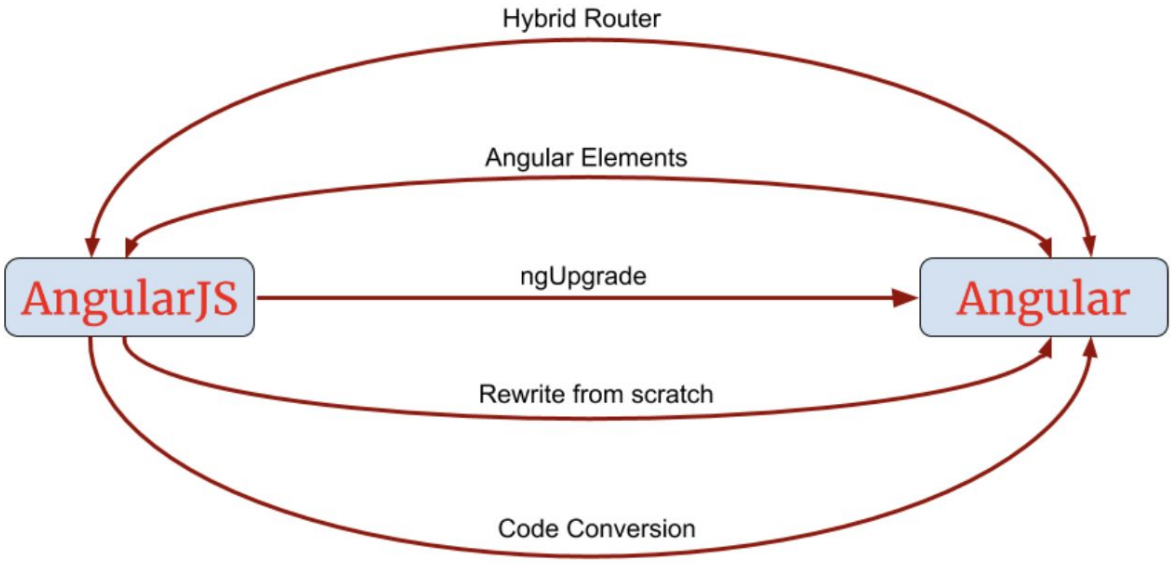

Figure 3: Different migration methods from Angular.JS to Angular

6.1 ngMigration Assistant

ngMigration Assistant is a command-line tool that analyzes an AngularJS application and recommends a migration path. It provides statistics on the complexity, size, and patterns of an app. It outlines the necessary preparation work for migrating to Angular.

ngMigration Assistant is designed to supply simple, clear, and constructive guidance for how to migrate your application. It analyzes any AngularJS app — big or small — and makes recommendations about possible approaches that may suit your application. It doesn't take into account every possibility, but considers some of the most important factors to consider, such as code size/complexity. With the simple command ***ngma*** with the app directory, we can see an analysis of our app and can see recommendations. The console log sample for the same is shown in the screen capture below. I cannot show the code snippets of my work due to the non disclosure agreement I signed before the beginning of my co-op.

```
Documents $ngma angular-phonecat-step-14

Welcome to ngMigration Assistant! \(^◡^)/
Scanning your files for...
  * Complexity
  * App size in lines of code and amount of relevant files and folders
  * AngularJS patterns
  * AngularJS version
  * Preparation necessary for migration

To learn more about criteria selection, visit https://angular.io/guide/upgrade#preparation

App Statistics
  * Complexity: 0 controllers, 2 AngularJS components, 19 JavaScript files, and 0 Typescript files.
  * App size: 49890 lines of code
  * File Count: 155 total files/folders, 46 relevant files/folders
  * AngularJS Patterns:  JavaScript
```

First the tool provides statistics about what was detected in the application. If necessary, preparation steps for migrating to Angular are shown. The example for the same is shown in the next screenshot.

```
Your Recommendation
Please follow these preparation steps before migrating with ngUpgrade.
  * App contains 19 JavaScript files that need to be converted to TypeScript.
      To learn more, visit https://angular.io/guide/upgrade#migrating-to-typescript

Files that contain AngularJS patterns and need to be modified:
1.  angular-phonecat-step-14/app/app.animations.js --> Modifications necessary:  JavaScript
2.  angular-phonecat-step-14/app/app.config.js --> Modifications necessary:  JavaScript
3.  angular-phonecat-step-14/app/app.module.js --> Modifications necessary:  JavaScript
4.  angular-phonecat-step-14/app/core/checkmark/checkmark.filter.js --> Modifications necessary:  JavaS
cript
5.  angular-phonecat-step-14/app/core/checkmark/checkmark.filter.spec.js --> Modifications necessary:
JavaScript
6.  angular-phonecat-step-14/app/core/core.module.js --> Modifications necessary:  JavaScript
7.  angular-phonecat-step-14/app/core/phone/phone.module.js --> Modifications necessary:  JavaScript
8.  angular-phonecat-step-14/app/core/phone/phone.service.js --> Modifications necessary:  JavaScript
9.  angular-phonecat-step-14/app/core/phone/phone.service.spec.js --> Modifications necessary:  JavaScr
ipt
10.  angular-phonecat-step-14/app/phone-detail/phone-detail.component.js --> Modifications necessary:
JavaScript, AngularJS component
11.  angular-phonecat-step-14/app/phone-detail/phone-detail.component.spec.js --> Modifications necessa
ry:  JavaScript
12.  angular-phonecat-step-14/app/phone-detail/phone-detail.module.js --> Modifications necessary:  Jav
aScript
13.  angular-phonecat-step-14/app/phone-list/phone-list.component.js --> Modifications necessary:  Java
Script, AngularJS component
14.  angular-phonecat-step-14/app/phone-list/phone-list.component.spec.js --> Modifications necessary:
 JavaScript
15.  angular-phonecat-step-14/app/phone-list/phone-list.module.js --> Modifications necessary:  JavaScr
ipt
16.  angular-phonecat-step-14/e2e-tests/protractor.conf.js --> Modifications necessary:  JavaScript
17.  angular-phonecat-step-14/e2e-tests/scenarios.js --> Modifications necessary:  JavaScript
18.  angular-phonecat-step-14/karma.conf.js --> Modifications necessary:  JavaScript
19.  angular-phonecat-step-14/scripts/private/old/ScrapeData.js --> Modifications necessary:  JavaScrip
t

Head to ngMigration-Forum to understand this migration approach.
https://github.com/angular/ngMigration-Forum/wiki
```

6.2 NgUpgrade

ngUpgrade is the official tool that allows Angular.JS developers to migrate the application little by little. It lets Angular run side-by-side along with the AngularJS code for as long as it is needed to slowly upgrade the application.

The steps that we plan on going through is as follows:

To get started with ngUpgrade, the application needs to meet a few prerequisites:

1. Code organized by feature (not by type) and every file contains only one item (like a directive or service)
2. TypeScript set up
3. Using a module bundler (most people use Webpack)
4. Using AngularJS 1.5+ with controllers replaced by components

The steps that we follow for carrying out the upgrade are mentioned in the appendices*.

7. Conclusion

The process of migration from Angular.js to Angular is gradual but is important in order to get continuous support. The forums around the Angular are being maintained regularly offering continuous updates and support. We are still in the process of setting up our system well in order to carry out the migration but with investigation we have laid the map and have steps that we are following in order to carry out the migration. This is a great learning experience as it has helped me read the systems well and I got a chance to investigate the existing code-base well and contribute in the discussions around the process.

8. Appendix

The details of the migration using ngupgrade are provided here:

*6.2.1. Installing Angular & ngUpgrade

After installing ngUpgrade, the dependencies look like this:

```
"dependencies": {
    "@angular/common": "^5.2.5",
    "@angular/compiler": "^5.2.5",
    "@angular/core": "^5.2.5",
    "@angular/forms": "^5.2.5",
    "@angular/platform-browser": "^5.2.5",
    "@angular/platform-browser-dynamic": "^5.2.5",
    "@angular/router": "^5.2.5",
    "@angular/upgrade": "^5.2.5",
    "angular": "1.6.6",
    "angular-route": "1.6.6",
    "bootstrap": "3.3.7",
    "core-js": "^2.5.3",
    "jquery": "^2.2.4",
    "lodash": "4.17.4",
    "moment": "~2.17.1",
    "reflect-metadata": "^0.1.12",
    "rxjs": "^5.5.6",
    "zone.js": "^0.8.20"
}
```

First are our libraries under the `@angular` namespace:

- `@angular/common`: These are the commonly needed services, pipes, and directives for Angular. This package also contains the new `HttpClient` as of version 4.3, so we no longer need `@angular/http`.
- `@angular/compiler`: This is Angular's template compiler. It takes the templates and converts them into the code that makes your application run and render. You almost never need to interact with it.
- `@angular/core`: These are the critical runtime parts of Angular needed by every application. This has things like the metadata decorators (e.g. `Component`, `Injectable`), all the dependency injection, and the component life-cycle hooks like `OnInit`.

- @angular/forms: This is just everything we need with forms, whether template or reactive.
- @angular/platform-browser: This is everything dom and browser related, especially pieces that help render the dom. This is the package that includes bootstrapStatic, which is the method that we use for bootstrapping our applications for production builds.
- @angular/platform-browser-dynamic: This package includes providers and another bootstrap method for applications that compile templates on the client. This is the package that we use for bootstrapping during development and we'll cover switching between the two in another video.
- @angular/router: As you might guess, this is just the router for Angular.
- @angular/upgrade: This is the ngUpgrade library, which allows us to migrate our AngularJS application to Angular.
- core-js patches the global context or the window with certain features of ES6 or ES2015.
- reflect-metadata is a polyfill library for the annotations that Angular uses in its classes.
- rxjs: This is the library that includes all of the observables that we'll use for handling our data.
- zone.js is a polyfill for the Zone specification, which is part of how Angular manages change detection.

After this step, we dual boot the application with Angular as well as the Angular.js.

*6.2.2. Setting Up ngUpgrade

In order to set it up, we need to follow a series of steps for both the angular and the angular.js. These are as follows

Step 1: Removing Bootstrap from index.html

The first thing we need to do is remove our bootstrap directive from index.html. This is how AngularJS normally gets started up at page load, but we're going to

bootstrap it through Angular using ngUpgrade. So we just open index.html and remove that data-ng-app tag.

Step 2: Changing the AngularJS Module

Now we need to make some changes in the AngularJS module. We open app.ts. The first thing we need to do is rename app.ts to app.module.ajs.ts to reflect that it's the module for AngularJS. It's kind of a lengthy name, but in Angular we want to have our type in our file name. Here we're using app.module and then we're adding that ajs to specify that it's for AngularJS instead of our root app.module for Angular.

As the app is now, we're just using AngularJS, so we have all of our import statements here and we're registering everything on our Angular module. However, now what we're going to do is export this module and import it into our new Angular module to get it up and running. So, on line 28 let's create a string constant of our app name:

```
const MODULE_NAME = 'app';
```

Then we'll replace our app string with module name in our Angular.module declaration:

```
angular.module(MODULE_NAME, ['ngRoute'])
// component and service registrations continue here
```

Copy

And finally, we need to export our constant:

```
export default MODULE_NAME;
```

Step 3: Creating the Angular App Module

Our AngularJS module is ready to go, so we're now ready to make our Angular module. We'll then import our AngularJS module so we can manually bootstrap it

here. That's what let's the two frameworks run together, and enables ngUpgrade to bridge the gap between them.
The first thing we need to do is create a new file at the same level as our AngularJS module named app.module.ts. Here, we will be making and exporting a class, decorating it with an annotation, and importing all of the dependencies.

In our new app module, we create a class named AppModule:

```
export class AppModule {
}
```

Now, we will add our first annotation also called as decorator. An annotation is just a bit of metadata that Angular uses when building our application. Above our new class, we'll use the NgModule annotation and pass in an options object:

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

Now, in our options object for ngModule, we need to pass an array of imports. The imports array specifies other NgModules that this NgModule will depend on. Right now, we need the BrowserModule and the UpgradeModule:
import { NgModule } from '@angular/core';

```
@NgModule({
  imports: [
    BrowserModule,
    UpgradeModule
  ]
})
export class AppModule { }
```

We don't have those imported either at the top of our file, so we need to do that too. After our first import, we can add:

```
import { BrowserModule } from '@angular/platform-browser';
import { UpgradeModule } from '@angular/upgrade/static';
```

There's an UpgradeModule in both upgrade and upgrade/static. We want to use the static one because it provides better error reporting and works with AOT (ahead-of-time) compiling.

Step 4: Bootstrapping in the Angular Module

To bootstrap our application, the first thing we need to do is inject UpgradeModule using a constructor function:

```
constructor(private upgrade: UpgradeModule){
}
```

We don't need to do anything in our constructor function. Now we will override the doBootstrap function. After the constructor we type:

```
ngDoBootstrap(){
}
```

Next, we'll use the Upgrade Modules bootstrap function. It has the same signature as the Angular bootstrap function, but it does a couple extra things for us. First, it makes sure that Angular and AngularJS run in the correct zones, and then it sets up an extra module that allows AngularJS to be visible in Angular and Angular to be visible in AngularJS. It adapts the testability APIs, so that Protractor will work with hybrid apps.

```
ngDoBootstrap(){
                this.upgrade.bootstrap(document.documentElement,  [moduleName],
{strictDi: true});
}
```

We're first passing in our document element and then our AngularJS module inside an array. Lastly, just so you can see an example of this, we're adding a config object so we can switch on strict dependency injection.
You may be wondering where the moduleName came from. We need to import it up with our other import statements:

```
import moduleName from './app.module.ajs';
```

Here's what our completed app.module.ts file looks like now:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { UpgradeModule } from '@angular/upgrade/static';
import moduleName from './app.module.ajs';

@NgModule({
   imports: [
      BrowserModule,
      UpgradeModule
   ]
})
export class AppModule {
   constructor(private upgrade: UpgradeModule) { }

   ngDoBootstrap(){
                this.upgrade.bootstrap(document.documentElement,   [moduleName],
{strictDi: true});
   }}
```

Step 5: Creating main.ts

Now that we've got our AngularJS module and our Angular module set up, we need an entry point that's going to bring these two together and get our application running. Let's create a new file under our src folder called main.ts.

In main.ts, we need to import a few things, tell Angular which version of AngularJS to load, and then tell it to bootstrap our Angular module. First, we need to import two polyfill libraries and Angular's platformBrowserDynamic function:

```
import 'zone.js';
import 'reflect-metadata';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
```

Angular has two ways to compile: a dynamic option and a static option. In the dynamic option (known as just-in-time, or JIT), the Angular compiler compiles the application in the browser and then launches the app. The static option (known as ahead-of-time, or AOT) produces a much smaller application that launches faster. This is because the Angular compiler runs ahead of time as part of the build process. We're just going to be using the JIT method here along with the Webpack dev server.

Now we need to import both our Angular and AngularJS modules, as well as a method that tells Angular which version of AngularJS to use:

```
import { setAngularLib } from '@angular/upgrade/static';
import * as angular from 'angular';
import { AppModule } from './app.module';
```

Now to finish this off, we just need to call setAngularLib and pass in our version of AngularJS, and we need to call platformBrowserDynamic and tell it to bootstrap our app module. The finished file looks like this:

```
import 'zone.js';
import 'reflect-metadata';
```

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { setAngularLib } from '@angular/upgrade/static';
import * as angular from 'angular';
import { AppModule } from './app.module';

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

Step 6: Updating Webpack

We have a main.ts file that's our entry point, which sets up our AngularJS library and bootstraps our Angular module. Then, our Angular module bootstraps our AngularJS module. That's what let's both frameworks run alongside each other.

We now change Webpack config so that it's starting with our main.ts file and not one of our app module files. Open webpack.common.js,Under module.exports for entry, we'll change our app root to main.ts:

```
entry: {
        app: './src/main.ts', }
```

We now have Angular and AngularJS running alongside each other, which means we've successfully set up our hybrid application. That means we're ready to start upgrading our application piece by piece.

*6.2.3. Rewriting & Downgrading Your First Component

Step 1: Rewriting the Component

We've got our application bootstrapped and running in hybrid mode, so we're ready to get started with migrating each piece of our application. One common approach is to pick a route and then start from the bottom up to rewrite each piece, starting with whatever has the least dependencies. This allows us to iteratively

upgrade our application so that every point along the way, we have something that's deployable to production.

We begin with the home route because that's an easy one with just the home component. We'll first rename our home component to home.component.ts. We need to rewrite our home component as an Angular class. The first thing we need to do is import component from the Angular core library at the top of our file:

```
import { Component } from '@angular/core'
```

The next thing we'll do is convert our function homeComponentController to a class. We can also capitalize it and remove the controller at the end of the name, so that it's just called HomeComponent. Lastly, let's get rid of the parenthesis. It looks like this now:

```
class HomeComponent {
    var vm = this;
    vm.title = 'Awesome, Inc. Internal Ordering System';
}
```

We clean up the inside of class. We no longer need the declaration of vm since we're using a class. We can also add a property of title as a string, and move setting the title to a constructor function. Our class looks like this now:

```
class HomeComponent {
    title: string;
    constructor(){
        title = 'Awesome, Inc. Internal Ordering System';
    }
}
```

We also need to export this class and then delete that export default line.
Now we need to apply the Component metadata decorator that we imported to tell Angular that this is a component. We can replace the home component object with the component decorator and an options object:

```
@Component({
}
```

The first option of our component decorator is the selector. This is just the HTML tag that we'll use to reference this component, which will just be 'home'. Note that in Angular, the selector is a string literal. This is different than in AngularJS, where we would name the component in camel case, and then it would translate to an HTML tag with hyphens. Here, we're going to put exactly the tag that we want to use. In this case, we're just keeping it to 'home', so it doesn't matter too much. After that, we'll specify our template, just like we did with AngularJS, so I'll just say template: template. And believe it or not, that's all there is to it. Our finished component looks like this:

```
import { Component } from '@angular/core';

const template = require('./home.html');

@Component({
    selector: 'home',
    template: template
})
export class HomeComponent {
    title: string;
    constructor(){
        this.title = 'Awesome, Inc. Internal Ordering System';
    }
}
```

Step 2: Downgrading the Component for AngularJS

We now need to use the ngUpgrade library to "downgrade" this component. "Downgrading" means to make an Angular component or service available to

AngularJS. “Upgrading,” on the other hand, means to make an AngularJS component or service available to Angular.

We need to import the downgradeComponent function from the Angular upgrade library and declare a variable called angular so we can register this component on our AngularJS module. This looks like:

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

Downgrading the component is fairly straightforward. Down at the bottom of our component, we’ll register this component as a directive. We’ll pass in our directive name, which is just home, the same as our selector in this case. Then after that, we’ll pass in the downgradeComponent function from ngUpgrade. This function converts our Angular component into an AngularJS directive. Finally, we’ll cast this object as angular.IDirectiveFactory. The finished registration looks like this:

```
app.module('app')
    .directive('home', downgradeComponent({component: HomeComponent} as angular.IDirectiveFactory);
```

Now we have a downgraded Angular component that’s available to our AngularJS application. The end goal is to get rid of that file altogether once all of our application is converted, so we want to gradually remove things from that file and then eventually delete it altogether when we uninstall AngularJS.

After a component is updated, we need to be sure to update its template so it complies with the new Angular syntax. In this case, there are only minimal changes you must make to homeComponent. We just need to remove $ctrl on line two. The template looks like this now:

```
<div class="row">
   <h1>{{title}}</h1>
</div>
```

Step 4: Add to the Angular App Module

Let's add our new Angular component to our Angular module. We open app.module.ts. First, we need to just import our home component after all of our other imports:

```
import { HomeComponent } from './home/home.component';
```

Now, we need to add HomeComponent to our Angular application. All Angular components must be added to a declarations array of our NgModule. We'll add a new array called declarations and add our component:

```
declarations: [
    HomeComponent
]
```

We also need to create an entryComponents array and add our HomeComponent to that. All downgraded components must be added to this entryComponents array. We'll add it after our declarations:

```
entryComponents: [
    HomeComponent
]
```

9. References

- https://angular.io/guide/upgrade
- https://www.javatpoint.com/angular-7-architecture
- https://angular-academy.com/angular-architecture-best-practices/
- https://docs.angularjs.org/guide/concepts