**Table of Contents**

1. Our Starting Point
2. Installing Angular & ngUpgrade
3. Setting up ngUpgrade
4. Rewrite & Downgrade Your First Component
5. Let's check that it works.
6. Where to Go Next

## Our Starting Point

To get started with ngUpgrade, your 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

(If you're lost on any of that, we cover it all in parts 1 and 2 of the course.)

For now, though, take a minute to clone or fork the course sample project on GitHub (don’t forget to run npm install). Checkout this commit to see our starting point:

## Installing Angular & ngUpgrade

### Installing Dependencies

We're ready to install Angular, ngUpgrade, and all of the peer dependencies. In the sample project, go ahead and update your package.json dependencies array so it looks 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"

}

(We're going to use Angular 5 in this series, even though the sample project uses version 4. Don't sweat it - the steps are identical.)

We could put all of these packages in one long command in the terminal with the save flag, but I want to take the time to explain to you what each of these packages are.

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 actually 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 stuff, 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.

After all of our Angular packages come our polyfill packages that are dependencies of 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.

*One thing to watch out for:* Sometimes, there are conflicts involving the version of TypeScript you're using. This can be due to RxJS, the Angular compiler, or Webpack. If you start getting weird compilation errors, do some research to find out of any of those need a specific version range of TypeScript for the version you're using.

Without further ado, open your terminal, cd into the public folder of the project, and run npm install (you're welcome to install and use Yarn if you'd prefer!). You should see that all of your packages were installed.

We're now ready to make our application a hybrid application by dual-booting both AngularJS and Angular.

## #Setting up ngUpgrade

To set up ngUpgrade, we need to do a series of steps to allow AngularJS and Angular to run alongside of each other.

### Step 1: Remove 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, just open index.html and remove that data-ng-app tag. (if you're using strict DI in your own app, you'll remove ng-strict-di as well in this step.) Your index.html file should look like this now:

<html>

<head>

<title>Amazing, Inc. Order System</title>

</head>

<body>

<navigation></navigation>

<div class="container" ng-view></div>

</body>

</html>

### Step 2: Change the AngularJS Module

Now we need to make some changes in AngularJS module. Open up 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 (which we'll make in a second).

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

And finally, we need to export our constant:

export default MODULE\_NAME;

You can check out the finished AngularJS module at this stage here.

### Step 3: Create 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 called app.module.ts. Now for the first time, you're about to see a pattern that's going to become very familiar to you throughout your upgrade: making and exporting a class, decorating it with an annotation, and importing all of the dependencies.

In our new app module, let's create a class named AppModule:

export class AppModule {

}

Now let's add our first annotation (also called a 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 {

}

If you're following along in an editor like Visual Studio Code, you'll see that TypeScript is mad at us because it doesn't know what NgModule is. This is because we need to import it from the Angular core library. Above our decorator, we can fix this with:

import { NgModule } from '@angular/core';

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. (These imports are different than the TypeScript imports at the top of our file.) Right now, we need the BrowserModule and the UpgradeModule:

import { NgModule } from '@angular/core';

@NgModule({

imports: [

BrowserModule,

UpgradeModule

]

})

export class AppModule { }

Of course, 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 actually 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.

We've got the basic scaffolding of our root module for Angular set up and we're ready to do the bootstrapping itself.

### Step 4: Bootstrap 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. The next thing we'll do is override the doBootstrap function. After the constructor, type:

ngDoBootstrap(){

}

Next, we'll use the UpgradeModule's bootstrap function. It actually 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. Lastly, it adapts the testability APIs, so that Protractor will work with hybrid apps, which is super important.

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.

Are you thinking, "Wait, where did that moduleName come from?" Great instinct! 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});

}

}

This is going to be a pattern that's going to become very familiar to you over time, so if you're a little confused right now, just hang in there and keep on going. You're doing great!

### Step 5: Create 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';

Why platformBrowserDynamic instead of just platformBrowser? 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 actually 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.

(In the course we spend an entire module setting up AOT compiling for production.)

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);

Now that we've got that set up, we just need to change our Webpack entry point in our config.

### Step 6: Update Webpack

Hopefully, this process of bootstrapping a hybrid application is starting to make sense to you. 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're now ready to change our Webpack config so that it's starting with our main.ts file and not one of our app module files. Open up webpack.common.js (it's under the webpack-configs folder). Under module.exports for entry, we'll change our app root to main.ts:

entry: {

app: './src/main.ts',

}

### Let's See It in Action

Now, we're ready to actually see our hybrid application in action. You can run the dev server by opening a terminal and running these commands:

cd server

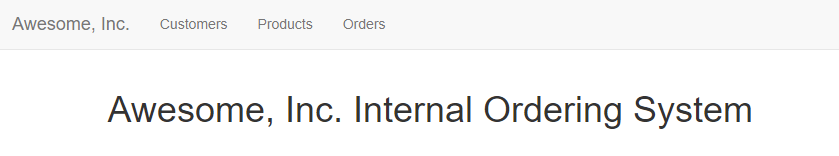
npm start

cd ../public

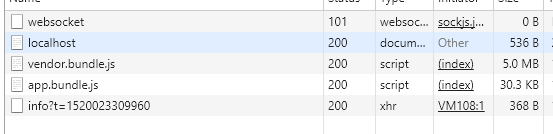
npm run dev

You should see that Webpack is loading and that our TypeScript is compiled successfully.

Let's go check out the browser at localhost:9000. You can see that our application still runs on our dev server!



You might see a couple of warnings in the console about core-js depending on your version, but don't worry about them, they won't affect us. You can also open the network tab and see the vendor bundle and app bundle:



The vendor bundle is absolutely huge, and that's because 1) we're running Webpack dev server, which means it's not minifying anything, 2) we're running Angular in dynamic compiling, so it's shipping the compiler code to the browser as well. We'll fix this downstream when we talk about AOT compiling, but we can actually navigate around here and see that all of our data is still loading. Awesome.

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

## #Rewrite & Downgrade Your First Component

### Step 1: Rewrite 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. The approach I typically take 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.

Let's start 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.

Now we need to rewrite our home component as an Angular class. Rewriting an AngularJS component to an Angular component is not that bad, especially since this one is pretty simple. 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';

}

Now let's clean up what's inside the class. We no longer need the declaration of vmsince 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 actually 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';

}

}

Note: If you're working on an application that will use the AOT compiler, you'll want to use *templateUrl* instead what we're doing here and make some changes to Webpack. This is totally fine for JIT and the development server, though.

### Step 2: Downgrade 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 availble to Angular. We'll cover that in another article. Luckily, downgrading is super easy.

#### Register and Downgrade the Component

First, we need to do two things at the top of our file along with our imports. We need to import the downgradeComponent function from the Angular upgrade library declare a variable called angular so we can register this component on our AngularJS module. This looks like this:

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

Downgrading the component is actually very simple. 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);

##### Remove from AngularJS Module

Now we have a downgraded Angular component that's available to our AngularJS application. You might be wondering why I registered that directive here at the bottom of this file instead of importing and registering it in our AngularJS module TypeScript file. The end goal is to actually get rid of that file altogether once all of our application is converted, so I want to gradually remove things from that file and then eventually delete it altogether when we uninstall AngularJS. This works great for sample applications or rapid migrations (more on that in a second).

Go ahead and open up app.module.ajs.ts and remove the import of homeComponent on line 12 and the component registration on line 37.

##### A Quick Note on AOT Compiling

This method of downgrading -- registering the downgraded component in the component file and removing it from the AngularJS module file -- works perfectly well for development or if you plan on quickly rewriting your application before you deploy. However, the Angular AOT compiler for production won't work with this method. Instead, it wants all of our downgraded registrations in the AngularJS module. Here in this sample project, we don't need to worry about that, but I'd be remiss if I didn't mention it for use in the real world.

The downgrade is identical, but instead you'd:

1. Import downgradeComponent in app.module.ajs.ts (you've already got angular in there so you don't need to declare it).
2. Change the import of homeComponent to import { HomeComponent } from './home/home.component'; since we switched to a named export.
3. Change the component registration to the exact same directive registration shown above.

You can read more about setting up ngUpgrade for AOT in this article I wrote, as well as in Course 3 of Upgrading AngularJS (there's a whole module that lays it out step-by-step).

### Step 3: Update the Template

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, the template for homeComponent is obviously stuper simple. We just need to remove $ctrl on line two. The template looks like this now:

<div class="row">

<h1>{{title}}</h1>

</div>

Now we have a fully functional downgraded home component in our hybrid application!

### Step 4: Add to the Angular App Module

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

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

I like to keep these imports separate from the imports up at the top.

Now, we need to add HomeComponent to our Angular application. All Angular components must be added to a declarations array of our NgModule. So, after line 12 in our options object, 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

]

And we're done!

## #Let's check that it works.

You did it! Let's run those same commands as before and make sure our application is still working. Here are those commands again:

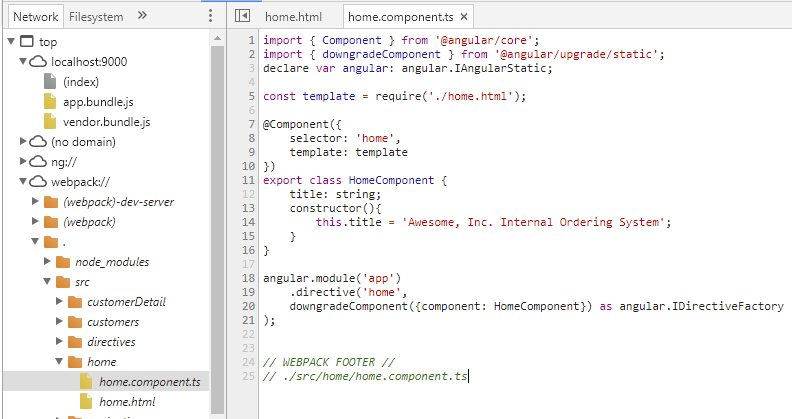
cd server

npm start

cd ../public

npm run dev

Head back over to localhost:9000. Lo and behold, you can see that our home component is actually loading in the browser as a rewritten Angular component! You can even go look at the Sources tab of Chrome devtools just to be positive. Open up webpack://, scroll down to ./src/home/home.component.ts, and sure enough, there it is!



Way to go!

## #Where to Go Next

Here's what you've accomplished in this guide:

1. Installed Angular and ngUpgrade
2. Set up an Angular module
3. Bootstrapped Angular and AngularJS
4. Updated Webpack
5. Rewritten and downgraded your first component