Upgrading from AngularJS to Angular

Angular is the name for the Angular of today and tomorrow.

AngularJS is the name for all 1.x versions of Angular.

AngularJS applications are great. Always consider the business case before moving to Angular. An important part of that case is the time and effort to get there. This guide describes the built-in tools for efficiently migrating AngularJS projects over to the Angular platform, a piece at a time.

Some applications will be easier to upgrade than others, and there are many ways to make it easier for yourself. It is possible to prepare and align AngularJS applications with Angular even before beginning the upgrade process. These preparation steps are all about making the code more decoupled, more maintainable, and better aligned with modern development tools. That means in addition to making the upgrade easier, you will also improve the existing AngularJS applications.

One of the keys to a successful upgrade is to do it incrementally, by running the two frameworks side by side in the same application, and porting AngularJS components to Angular one by one. This makes it possible to upgrade even large and complex applications without disrupting other business, because the work can be done collaboratively and spread over a period of time. The `upgrade` module in Angular has been designed to make incremental upgrading seamless.

Preparation

There are many ways to structure AngularJS applications. When you begin to upgrade these applications to Angular, some will turn out to be much more easy to work with than others. There are a few key techniques and patterns that you can apply to future proof applications even before you begin the migration.

Follow the AngularJS Style Guide

The AngularJS Style Guide collects patterns and practices that have been proven to result in cleaner and more maintainable AngularJS applications. It contains a wealth of information about how to write and organize AngularJS code —and equally importantly— how **not** to write and organize AngularJS code.

Angular is a reimagined version of the best parts of AngularJS. In that sense, its goals are the same as the Style Guide for AngularJS: To preserve the good parts of AngularJS, and to avoid the bad parts. There is a lot more to Angular than that of course, but this does mean that *following the style guide helps make your AngularJS application more closely aligned with Angular.*

There are a few rules in particular that will make it much easier to do *an incremental upgrade* using the Angular `upgrade/static` module:

- The Rule of 1 states that there should be one component per file. This not only makes components easy to navigate and find, but will also allow us to migrate them between languages and frameworks one at a time. In this example application, each controller, component, service, and filter is in its own source file.
- The Folders-by-Feature Structure and Modularity rules define similar principles on a higher level of abstraction: Different parts of the application should reside in different directories and NgModules.

When an application is laid out feature per feature in this way, it can also be migrated one feature at a time. For applications that don't already look like this, applying the rules in the AngularJS style guide is a highly recommended preparation step. And this is not just for the sake of the upgrade - it is just solid advice in general!

Using a Module Loader

When you break application code down into one component per file, you often end up with a project structure with a large number of relatively small files. This is a much neater way to organize things than a small number of large files, but it doesn't work that well if you have to load all those files to the HTML page with `<script>` tags. Especially when you also have to maintain those tags in the correct order. That is why it is a good idea to start using a *module loader*.

Using a module loader such as SystemJS, Webpack, or Browserify allows us to use the built-in module systems of TypeScript or ES2015. You can use the `import` and `export` features that explicitly specify what code can and will be shared between different parts of the application. For ES5 applications you can use CommonJS style `require` and `module.exports` features. In both cases, the module loader will then take care of loading all the code the application needs in the correct order.

When moving applications into production, module loaders also make it easier to package them all up into production bundles with batteries included.

Migrating to TypeScript

If part of the Angular upgrade plan is to also take TypeScript into use, it makes sense to bring in the TypeScript compiler even before the upgrade itself begins. This means there is one less thing to learn and think about during the actual upgrade. It also means you can start using TypeScript features in your AngularJS code.

Since TypeScript is a superset of ECMAScript 2015, which in turn is a superset of ECMAScript 5, "switching" to TypeScript doesn't necessarily require anything more than installing the TypeScript compiler and renaming files from `*.js` to `*.ts`. But just doing that is not hugely useful or exciting, of course. Additional steps like the following can give us much more bang for the buck:

- For applications that use a module loader, TypeScript imports and exports (which are really ECMAScript 2015 imports and exports) can be used to organize code into modules.
- Type annotations can be gradually added to existing functions and variables to pin down their types and get benefits like build-time error checking, great autocompletion support and inline documentation.
- JavaScript features new to ES2015, like arrow functions, `let`s and `const`s, default function parameters, and destructuring assignments can also be gradually added to make the code more expressive.
- Services and controllers can be turned into *classes*. That way they'll be a step closer to becoming Angular service and component classes, which will make life easier after the upgrade.

Using Component Directives

In Angular, components are the main primitive from which user interfaces are built. You define the different portions of the UI as components and compose them into a full user experience.

You can also do this in AngularJS, using *component directives*. These are directives that define their own templates, controllers, and input/output bindings - the same things that Angular components define. Applications built with component directives are much easier to migrate to Angular than applications built with lower-level features like `ng-controller`, `ng-include`, and scope inheritance.

To be Angular compatible, an AngularJS component directive should configure these attributes:

- `restrict: 'E'` Components are usually used as elements.
- `scope: {}` - an isolate scope. In Angular, components are always isolated from their surroundings, and you should do this in AngularJS too.
- `bindToController: {}`. Component inputs and outputs should be bound to the controller instead of using the `$scope`.
- `controller` and `controllerAs`. Components have their own controllers.
- `template` or `templateUrl`. Components have their own templates.

Component directives may also use the following attributes:

- `transclude: true/{}`, if the component needs to transclude content from elsewhere.
- `require`, if the component needs to communicate with the controller of some parent component.

Component directives **should not** use the following attributes:

- `compile`. This will not be supported in Angular.

- `replace: true`. Angular never replaces a component element with the component template. This attribute is also deprecated in AngularJS.
- `priority` and `terminal`. While AngularJS components may use these, they are not used in Angular and it is better not to write code that relies on them.

An AngularJS component directive that is fully aligned with the Angular architecture may look something like this:

AngularJS 1.5 introduces the component API that makes it easier to define component directives like these. It is a good idea to use this API for component directives for several reasons:

- It requires less boilerplate code.
- It enforces the use of component best practices like `controllerAs`.
- It has good default values for directive attributes like `scope` and `restrict`.

The component directive example from above looks like this when expressed using the component API:

Controller lifecycle hook methods `$onInit()`, `$onDestroy()`, and `$onChanges()` are another convenient feature that AngularJS 1.5 introduces. They all have nearly exact equivalents in Angular, so organizing component lifecycle logic around them will ease the eventual Angular upgrade process.

Upgrading with ngUpgrade

The ngUpgrade library in Angular is a very useful tool for upgrading anything but the smallest of applications. With it you can mix and match AngularJS and Angular components in the same application and have them interoperate seamlessly. That means you don't have to do the upgrade work all at once, since there is a natural coexistence between the two frameworks during the transition period.

The end of life of AngularJS is December 31st, 2021. With this event, ngUpgrade is now in a feature complete state. We will continue publishing security and bug fixes for ngUpgrade at least until December 31st, 2023.

How ngUpgrade Works

One of the primary tools provided by ngUpgrade is called the `UpgradeModule`. This is a module that contains utilities for bootstrapping and managing hybrid applications that support both Angular and AngularJS code.

When you use ngUpgrade, 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. Both of these are the actual, fully featured versions of the frameworks. There is no emulation

going on, so you can expect to have all the features and natural behavior of both frameworks.

What happens on top of this is that components and services managed by one framework can interoperate with those from the other framework. This happens in three main areas: Dependency injection, the DOM, and change detection.

Dependency Injection Dependency injection is front and center in both AngularJS and Angular, but there are some key differences between the two frameworks in how it actually works.

AngularJS	Angular
Dependency injection tokens are always strings	Tokens can have different types.They are often classes.They may also be strings.
There is exactly one injector.Even in multi-module applications, everything is poured into one big namespace.	There is a tree hierarchy of injectors, with a root injector and an additional injector for each component.

Even accounting for these differences you can still have dependency injection interoperability. `upgrade/static` resolves the differences and makes everything work seamlessly:

- You can make AngularJS services available for injection to Angular code by *upgrading* them. The same singleton instance of each service is shared between the frameworks. In Angular these services will always be in the *root injector* and available to all components.
- You can also make Angular services available for injection to AngularJS code by *downgrading* them. Only services from the Angular root injector can be downgraded. Again, the same singleton instances are shared between the frameworks. When you register a downgraded service, you must explicitly specify a *string token* that you want to use in AngularJS.

Components and the DOM In the DOM of a hybrid ngUpgrade application are components and directives from both AngularJS and Angular. These components communicate with each other by using the input and output bindings of their respective frameworks, which ngUpgrade bridges together. They may also communicate through shared injected dependencies, as described above.

The key thing to understand about a hybrid application is that every element in the DOM is owned by exactly one of the two frameworks. The other framework ignores it. If an element is owned by AngularJS, Angular treats it as if it didn't exist, and vice versa.

So normally a hybrid application begins life as an AngularJS application, and it is AngularJS that processes the root template, for example, the index.html. Angular

then steps into the picture when an Angular component is used somewhere in an AngularJS template. The template of that component will then be managed by Angular, and it may contain any number of Angular components and directives.

Beyond that, you may interleave the two frameworks. You always cross the boundary between the two frameworks by one of two ways:

1. By using a component from the other framework: An AngularJS template using an Angular component, or an Angular template using an AngularJS component.
2. By transcluding or projecting content from the other framework. ngUpgrade bridges the related concepts of AngularJS transclusion and Angular content projection together.

Whenever you use a component that belongs to the other framework, a switch between framework boundaries occurs. However, that switch only happens to the elements in the template of that component. Consider a situation where you use an Angular component from AngularJS like this:

<a-component></a-component>

The DOM element `<a-component>` will remain to be an AngularJS managed element, because it is defined in an AngularJS template. That also means you can apply additional AngularJS directives to it, but *not* Angular directives. It is only in the template of the `<a-component>` where Angular steps in. This same rule also applies when you use AngularJS component directives from Angular.

Change Detection The `scope.$apply()` is how AngularJS detects changes and updates data bindings. After every event that occurs, `scope.$apply()` gets called. This is done either automatically by the framework, or manually by you.

In Angular things are different. While change detection still occurs after every event, no one needs to call `scope.$apply()` for that to happen. This is because all Angular code runs inside something called the Angular zone. Angular always knows when the code finishes, so it also knows when it should kick off change detection. The code itself doesn't have to call `scope.$apply()` or anything like it.

In the case of hybrid applications, the `UpgradeModule` bridges the AngularJS and Angular approaches. Here is what happens:

- Everything that happens in the application runs inside the Angular zone. This is true whether the event originated in AngularJS or Angular code. The zone triggers Angular change detection after every event.
- The `UpgradeModule` will invoke the AngularJS `$rootScope.$apply()` after every turn of the Angular zone. This also triggers AngularJS change detection after every event.

In practice, you do not need to call `$apply()`, regardless of whether it is in AngularJS or Angular. The `UpgradeModule` does it for us. You *can* still call `$apply()` so there is no need to remove such calls from existing code. Those calls just trigger additional AngularJS change detection checks in a hybrid application.

When you downgrade an Angular component and then use it from AngularJS, the inputs of the component will be watched using AngularJS change detection. When those inputs change, the corresponding properties in the component are set. You can also hook into the changes by implementing the OnChanges interface in the component, just like you could if it hadn't been downgraded.

Correspondingly, when you upgrade an AngularJS component and use it from Angular, all the bindings defined for `scope` (or `bindToController`) of the component directive will be hooked into Angular change detection. They will be treated as regular Angular inputs. Their values will be written to the scope (or controller) of the upgraded component when they change.

Using UpgradeModule with Angular *NgModules*

Both AngularJS and Angular have their own concept of modules to help organize an application into cohesive blocks of functionality.

Their details are quite different in architecture and implementation. In AngularJS, you add Angular assets to the `angular.module` property. In Angular, you create one or more classes adorned with an `NgModule` decorator that describes Angular assets in metadata. The differences blossom from there.

In a hybrid application you run both versions of Angular at the same time. That means that you need at least one module each from both AngularJS and Angular. You will import `UpgradeModule` inside the NgModule, and then use it for bootstrapping the AngularJS module.

For more information, see NgModules.

Bootstrapping hybrid applications

To bootstrap a hybrid application, you must bootstrap each of the Angular and AngularJS parts of the application. You must bootstrap the Angular bits first and then ask the `UpgradeModule` to bootstrap the AngularJS bits next.

In an AngularJS application you have a root AngularJS module, which will also be used to bootstrap the AngularJS application.

Pure AngularJS applications can be automatically bootstrapped by using an `ng-app` directive somewhere on the HTML page. But for hybrid applications, you manually bootstrap using the `UpgradeModule`. Therefore, it is a good preliminary step to switch AngularJS applications to use the manual JavaScript `angular.bootstrap` method even before switching them to hybrid mode.

Say you have an `ng-app` driven bootstrap such as this one:

You can remove the `ng-app` and `ng-strict-di` directives from the HTML and instead switch to calling `angular.bootstrap` from JavaScript, which will result in the same thing:

To begin converting your AngularJS application to a hybrid, you need to load the Angular framework. You can see how this can be done with SystemJS by following the instructions in Setup for Upgrading to AngularJS for selectively copying code from the QuickStart github repository.

You also need to install the `@angular/upgrade` package using `npm install @angular/upgrade --save` and add a mapping for the `@angular/upgrade/static` package:

Next, create an `app.module.ts` file and add the following `NgModule` class:

This bare minimum `NgModule` imports `BrowserModule`, the module every Angular browser-based application must have. It also imports `UpgradeModule` from `@angular/upgrade/static`, which exports providers that will be used for upgrading and downgrading services and components.

In the constructor of the `AppModule`, use dependency injection to get a hold of the `UpgradeModule` instance, and use it to bootstrap the AngularJS application in the `AppModule.ngDoBootstrap` method. The `upgrade.bootstrap` method takes the exact same arguments as angular.bootstrap:

NOTE: You do not add a `bootstrap` declaration to the `@NgModule` decorator, since AngularJS will own the root template of the application.

Now you can bootstrap `AppModule` using the `platformBrowserDynamic.bootstrapModule` method.

Congratulations! You're running a hybrid application! The existing AngularJS code works as before *and* you're ready to start adding Angular code.

Using Angular Components from AngularJS Code

Once you're running a hybrid app, you can start the gradual process of upgrading code. One of the more common patterns for doing that is to use an Angular component in an AngularJS context. This could be a completely new component or one that was previously AngularJS but has been rewritten for Angular.

Say you have an Angular component that shows information about a hero:

If you want to use this component from AngularJS, you need to *downgrade* it using the `downgradeComponent()` method. The result is an AngularJS *directive*, which you can then register in the AngularJS module:

By default, Angular change detection will also run on the component for everyAngularJS `$digest` cycle. If you want to only have change detection run whenthe inputs change, you can set `propagateDigest` to `false` when calling`downgradeComponent()`.

Because `HeroDetailComponent` is an Angular component, you must also add it to the `declarations` in the `AppModule`.

All Angular components, directives and pipes must be declared in an NgModule.

The net result is an AngularJS directive called `heroDetail`, that you can use like any other directive in AngularJS templates.

NOTE: This AngularJS is an element directive (`restrict: 'E'`) called `heroDetail`. An AngularJS element directive is matched based on its *name*. *The `selector` metadata of the downgraded Angular component is ignored.*

Most components are not quite this simple, of course. Many of them have *inputs and outputs* that connect them to the outside world. An Angular hero detail component with inputs and outputs might look like this:

These inputs and outputs can be supplied from the AngularJS template, and the `downgradeComponent()` method takes care of wiring them up:

Even though you are in an AngularJS template, **you are using Angular attribute syntax to bind the inputs and outputs**. This is a requirement for downgraded components. The expressions themselves are still regular AngularJS expressions.

Use kebab-case for downgraded component attributes

There is one notable exception to the rule of using Angular attribute syntax for downgraded components. It has to do with input or output names that consist of multiple words. In Angular, you would bind these attributes using camelCase:

[myHero]="hero" (heroDeleted)="handleHeroDeleted($event)"

But when using them from AngularJS templates, you must use kebab-case:

[my-hero]="hero" (hero-deleted)="handleHeroDeleted($event)"

The `$event` variable can be used in outputs to gain access to the object that was emitted. In this case it will be the `Hero` object, because that is what was passed to `this.deleted.emit()`.

Since this is an AngularJS template, you can still use other AngularJS directives on the element, even though it has Angular binding attributes on it. For example, you can easily make multiple copies of the component using `ng-repeat`:

Using AngularJS Component Directives from Angular Code

So, you can write an Angular component and then use it from AngularJS code. This is useful when you start to migrate from lower-level components and work your way up. But in some cases it is more convenient to do things in the opposite order: To start with higher-level components and work your way down. This too can be done using the `upgrade/static`. You can *upgrade* AngularJS component directives and then use them from Angular.

Not all kinds of AngularJS directives can be upgraded. The directive really has to be a *component directive*, with the characteristics described in the preparation guide above. The safest bet for ensuring compatibility is using the component API introduced in AngularJS 1.5.

An example of an upgradeable component is one that just has a template and a controller:

You can *upgrade* this component to Angular using the `UpgradeComponent` class. By creating a new Angular **directive** that extends `UpgradeComponent` and doing a `super` call inside its constructor, you have a fully upgraded AngularJS component to be used inside Angular. All that is left is to add it to the `declarations` array of `AppModule`.

Upgraded components are Angular **directives**, instead of **components**, because Angular is unaware that AngularJS will create elements under it. As far as Angular knows, the upgraded component is just a directive —a tag— and Angular doesn't have to concern itself with its children.

An upgraded component may also have inputs and outputs, as defined by the scope/controller bindings of the original AngularJS component directive. When you use the component from an Angular template, provide the inputs and outputs using **Angular template syntax**, observing the following rules:

	Binding definition	Template syntax
Attribute binding	`myAttribute: '@myAttribute'`	`<my-component myAttribute="value">`
Expression binding	`myOutput: '&myOutput'`	`<my-component (myOutput)="action()">`
One-way binding	`myValue: '<myValue'`	`<my-component [myValue]="anExpression">`
Two-way binding	`myValue: '=myValue'`	As a two-way binding:`<my-component [(myValue)]="anExpression">`Since most AngularJS two-way bindings actually only need a one-way binding in practice, `<my-component [myValue]="anExpression">` is often enough.

For example, imagine a hero detail AngularJS component directive with one input and one output:

You can upgrade this component to Angular, annotate inputs and outputs in the upgrade directive, and then provide the input and output using Angular

template syntax:

Projecting AngularJS Content into Angular Components

When you are using a downgraded Angular component from an AngularJS template, the need may arise to *transclude* some content into it. This is also possible. While there is no such thing as transclusion in Angular, there is a very similar concept called *content projection*. `upgrade/static` is able to make these two features interoperate.

Angular components that support content projection make use of an `<ng-content>` tag within them. Here is an example of such a component:

When using the component from AngularJS, you can supply contents for it. Just like they would be transcluded in AngularJS, they get projected to the location of the `<ng-content>` tag in Angular:

When AngularJS content gets projected inside an Angular component, it still remains in "AngularJS land" and is managed by the AngularJS framework.

Transcluding Angular Content into AngularJS Component Directives

Just as you can project AngularJS content into Angular components, you can *transclude* Angular content into AngularJS components, whenever you are using upgraded versions from them.

When an AngularJS component directive supports transclusion, it may use the `ng-transclude` directive in its template to mark the transclusion point:

If you upgrade this component and use it from Angular, you can populate the component tag with contents that will then get transcluded:

Making AngularJS Dependencies Injectable to Angular

When running a hybrid app, you may encounter situations where you need to inject some AngularJS dependencies into your Angular code. Maybe you have some business logic still in AngularJS services. Maybe you want access to built-in services of AngularJS like `$location` or `$timeout`.

In these situations, it is possible to *upgrade* an AngularJS provider to Angular. This makes it possible to then inject it somewhere in Angular code. For example, you might have a service called `HeroesService` in AngularJS:

You can upgrade the service using a Angular factory provider that requests the service from the AngularJS `$injector`.

Many developers prefer to declare the factory provider in a separate `ajs-upgraded-providers.ts` file so that they are all together, making it easier to reference them, create new ones and delete them once the upgrade is over.

It is also recommended to export the `heroesServiceFactory` function so that Ahead-of-Time compilation can pick it up.

NOTE: The 'heroes' string inside the factory refers to the AngularJS `HeroesService`. It is common in AngularJS applications to choose a service name for the token, for example "heroes", and append the "Service" suffix to create the class name.

You can then provide the service to Angular by adding it to the `@NgModule`:

Then use the service inside your component by injecting it in the component constructor using its class as a type annotation:

In this example you upgraded a service class. You can use a TypeScript type annotation when you inject it. While it doesn't affect how the dependency is handled, it enables the benefits of static type checking. This is not required though, and any AngularJS service, factory, or provider can be upgraded.

Making Angular Dependencies Injectable to AngularJS

In addition to upgrading AngularJS dependencies, you can also *downgrade* Angular dependencies, so that you can use them from AngularJS. This can be useful when you start migrating services to Angular or creating new services in Angular while retaining components written in AngularJS.

For example, you might have an Angular service called `Heroes`:

Again, as with Angular components, register the provider with the `NgModule` by adding it to the `providers` list of the module.

Now wrap the Angular `Heroes` in an *AngularJS factory function* using `downgradeInjectable()` and plug the factory into an AngularJS module. The name of the AngularJS dependency is up to you:

After this, the service is injectable anywhere in AngularJS code:

Lazy Loading AngularJS

When building applications, you want to ensure that only the required resources are loaded when necessary. Whether that be loading of assets or code, making sure everything that can be deferred until needed keeps your application running efficiently. This is especially true when running different frameworks in the same application.

Lazy loading is a technique that defers the loading of required assets and code resources until they are actually used. This reduces startup time and increases efficiency, especially when running different frameworks in the same application.

When migrating large applications from AngularJS to Angular using a hybrid approach, you want to migrate some of the most commonly used features first, and only use the less commonly used features if needed. Doing so helps you

ensure that the application is still providing a seamless experience for your users while you are migrating.

In most environments where both Angular and AngularJS are used to render the application, both frameworks are loaded in the initial bundle being sent to the client. This results in both increased bundle size and possible reduced performance.

Overall application performance is affected in cases where the user stays on Angular-rendered pages because the AngularJS framework and application are still loaded and running, even if they are never accessed.

You can take steps to mitigate both bundle size and performance issues. By isolating your AngularJS application to a separate bundle, you can take advantage of lazy loading to load, bootstrap, and render the AngularJS application only when needed. This strategy reduces your initial bundle size, defers any potential impact from loading both frameworks until absolutely necessary, and keeps your application running as efficiently as possible.

The steps below show you how to do the following:

- Setup a callback function for your AngularJS bundle.
- Create a service that lazy loads and bootstraps your AngularJS app.
- Create a routable component for AngularJS content
- Create a custom `matcher` function for AngularJS-specific URLs and configure the Angular `Router` with the custom matcher for AngularJS routes.

Create a service to lazy load AngularJS

As of Angular version 8, lazy loading code can be accomplished by using the dynamic import syntax `import('...')`. In your application, you create a new service that uses dynamic imports to lazy load AngularJS.

The service uses the `import()` method to load your bundled AngularJS application lazily. This decreases the initial bundle size of your application as you're not loading code your user doesn't need yet. You also need to provide a way to *bootstrap* the application manually after it has been loaded. AngularJS provides a way to manually bootstrap an application using the angular.bootstrap() method with a provided HTML element. Your AngularJS application should also expose a `bootstrap` method that bootstraps the AngularJS app.

To ensure any necessary teardown is triggered in the AngularJS app, such as removal of global listeners, you also implement a method to call the `$rootScope.destroy()` method.

Your AngularJS application is configured with only the routes it needs to render content. The remaining routes in your application are handled by the Angular Router. The exposed `bootstrap` method is called in your Angular application to bootstrap the AngularJS application after the bundle is loaded.

NOTE: After AngularJS is loaded and bootstrapped, listeners such as those wired up in your route configuration will continue to listen for route changes. To ensure listeners are shut down when AngularJS isn't being displayed, configure an `otherwise` option with the $routeProvider that renders an empty template. This assumes all other routes will be handled by Angular.

Create a component to render AngularJS content

In your Angular application, you need a component as a placeholder for your AngularJS content. This component uses the service you create to load and bootstrap your AngularJS application after the component is initialized.

When the Angular Router matches a route that uses AngularJS, the `AngularJSComponent` is rendered, and the content is rendered within the AngularJS `ng-view` directive. When the user navigates away from the route, the `$rootScope` is destroyed on the AngularJS application.

Configure a custom route matcher for AngularJS routes

To configure the Angular Router, you must define a route for AngularJS URLs. To match those URLs, you add a route configuration that uses the `matcher` property. The `matcher` allows you to use custom pattern matching for URL paths. The Angular Router tries to match on more specific routes such as static and variable routes first. When it doesn't find a match, it then looks at custom matchers defined in your route configuration. If the custom matchers don't match a route, it then goes to catch-all routes, such as a 404 page.

The following example defines a custom matcher function for AngularJS routes.

The following code adds a route object to your routing configuration using the `matcher` property and custom matcher, and the `component` property with `AngularJSComponent`.

When your application matches a route that needs AngularJS, the AngularJS application is loaded and bootstrapped, the AngularJS routes match the necessary URL to render their content, and your application continues to run with both AngularJS and Angular frameworks.

Using the Unified Angular Location Service

In AngularJS, the $location service handles all routing configuration and navigation, encoding and decoding of URLS, redirects, and interactions with browser APIs. Angular uses its own underlying `Location` service for all of these tasks.

When you migrate from AngularJS to Angular you will want to move as much responsibility as possible to Angular, so that you can take advantage of new APIs. To help with the transition, Angular provides the `LocationUpgradeModule`. This module enables a *unified* location service that shifts responsibilities from the AngularJS `$location` service to the Angular `Location` service.

To use the `LocationUpgradeModule`, import the symbol from `@angular/common/upgrade` and add it to your `AppModule` imports using the static `LocationUpgradeModule.config()` method.

// Other imports ... import { LocationUpgradeModule } from '@angular/common/upgrade';

@NgModule({ imports: [// Other NgModule imports... LocationUpgradeModule.config()] }) export class AppModule {}

The `LocationUpgradeModule.config()` method accepts a configuration object that allows you to configure options including the `LocationStrategy` with the `useHash` property, and the URL prefix with the `hashPrefix` property.

The `useHash` property defaults to `false`, and the `hashPrefix` defaults to an empty `string`. Pass the configuration object to override the defaults.

LocationUpgradeModule.config({ useHash: true, hashPrefix: '!' })

NOTE: See the `LocationUpgradeConfig` for more configuration options available to the `LocationUpgradeModule.config()` method.

This registers a drop-in replacement for the `$location` provider in AngularJS. Once registered, all navigation, routing broadcast messages, and any necessary digest cycles in AngularJS triggered during navigation are handled by Angular. This gives you a single way to navigate within both sides of your hybrid application consistently.

For usage of the `$location` service as a provider in AngularJS, you need to downgrade the `$locationShim` using a factory provider.

// Other imports ... import { $locationShim } from '@angular/common/upgrade'; import { downgradeInjectable } from '@angular/upgrade/static';

angular.module('myHybridApp', [...]) .factory('*location*′, *downgradeInjectable*(locationShim));

Once you introduce the Angular Router, using the Angular Router triggers navigations through the unified location service, still providing a single source for navigating with AngularJS and Angular.

PhoneCat Upgrade Tutorial

In this section, you'll learn to prepare and upgrade an application with `ngUpgrade`. The example application is Angular PhoneCat from the original AngularJS tutorial, which is where many of us began our Angular adventures. Now you'll see how to bring that application to the brave new world of Angular.

During the process you'll learn how to apply the steps outlined in the preparation guide. You'll align the application with Angular and also start writing in TypeScript.

This tutorial is based on the 1.5.x version of the `angular-phonecat` tutorial, which is preserved in the 1.5-snapshot branch of the repository. To follow along, clone the angular-phonecat repository, check out the `1.5-snapshot` branch and apply the steps as you go.

In terms of project structure, this is where the work begins:

```
angular-phonecat
```

```
<div class='file'>
  bower.json
</div>
<div class='file'>
  karma.conf.js
</div>
<div class='file'>
  package.json
</div>
<div class='file'>
  app
</div>
<div class='children'>
  <div class='file'>
    core
  </div>
  <div class='children'>
    <div class='file'>
      checkmark
    </div>
    <div class='children'>
      <div class='file'>
        checkmark.filter.js
      </div>
      <div class='file'>
        checkmark.filter.spec.js
      </div>
    </div>
    <div class='file'>
      phone
    </div>
    <div class='children'>
      <div class='file'>
        phone.module.js
      </div>
      <div class='file'>
        phone.service.js
      </div>
```

```
    <div class='file'>
      phone.service.spec.js
    </div>
  </div>
  <div class='file'>
    core.module.js
  </div>
</div>
<div class='file'>
  phone-detail
</div>
<div class='children'>
  <div class='file'>
    phone-detail.component.js
  </div>
  <div class='file'>
    phone-detail.component.spec.js
  </div>
  <div class='file'>
    phone-detail.module.js
  </div>
  <div class='file'>
    phone-detail.template.html
  </div>
</div>
<div class='file'>
  phone-list
</div>
<div class='children'>
  <div class='file'>
    phone-list.component.js
  </div>
  <div class='file'>
    phone-list.component.spec.js
  </div>
  <div class='file'>
    phone-list.module.js
  </div>
  <div class='file'>
    phone-list.template.html
  </div>
</div>
<div class='file'>
  img
</div>
<div class='children'>
```

```
    <div class='file'>
       ...
    </div>
  </div>
  <div class='file'>
    phones
  </div>
  <div class='children'>
    <div class='file'>
       ...
    </div>
  </div>
  <div class='file'>
    app.animations.js
  </div>
  <div class='file'>
    app.config.js
  </div>
  <div class='file'>
    app.css
  </div>
  <div class='file'>
    app.module.js
  </div>
  <div class='file'>
    index.html
  </div>
</div>
<div class='file'>
  e2e-tests
</div>
<div class='children'>
  <div class='file'>
    protractor-conf.js
  </div>
  <div class='file'>
    scenarios.js
  </div>
</div>
```

This is actually a pretty good starting point. The code uses the AngularJS 1.5 component API and the organization follows the AngularJS Style Guide, which is an important preparation step before a successful upgrade.

- Each component, service, and filter is in its own source file, as per the Rule of 1.

- The `core`, `phone-detail`, and `phone-list` modules are each in their own subdirectory. Those subdirectories contain the JavaScript code as well as the HTML templates that go with each particular feature. This is in line with the Folders-by-Feature Structure and Modularity rules.
- Unit tests are located side-by-side with application code where they are easily found, as described in the rules for Organizing Tests.

Switching to TypeScript

Since you're going to be writing Angular code in TypeScript, it makes sense to bring in the TypeScript compiler even before you begin upgrading.

You'll also start to gradually phase out the Bower package manager in favor of NPM, installing all new dependencies using NPM, and eventually removing Bower from the project.

Begin by installing TypeScript to the project.

npm i typescript –save-dev

Install type definitions for the existing libraries that you're using but that don't come with prepackaged types: AngularJS, AngularJS Material, and the Jasmine unit test framework.

For the PhoneCat app, we can install the necessary type definitions by running the following command:

npm install @types/jasmine @types/angular @types/angular-animate @types/angular-aria @types/angular-cookies @types/angular-mocks @types/angular-resource @types/angular-route @types/angular-sanitize –save-dev

If you are using AngularJS Material, you can install the type definitions via:

npm install @types/angular-material –save-dev

You should also configure the TypeScript compiler with a `tsconfig.json` in the project directory as described in the TypeScript Configuration guide. The `tsconfig.json` file tells the TypeScript compiler how to turn your TypeScript files into ES5 code bundled into CommonJS modules.

Finally, you should add some npm scripts in `package.json` to compile the TypeScript files to JavaScript (based on the `tsconfig.json` configuration file):

“scripts”: { “tsc”: “tsc”, “tsc:w”: “tsc -w”, . . .

Now launch the TypeScript compiler from the command line in watch mode:

npm run tsc:w

Keep this process running in the background, watching and recompiling as you make changes.

Next, convert your current JavaScript files into TypeScript. Since TypeScript is a super-set of ECMAScript 2015, which in turn is a super-set of ECMAScript 5, you can switch the file extensions from `.js` to `.ts` and everything will work just like it did before. As the TypeScript compiler runs, it emits the corresponding `.js` file for every `.ts` file and the compiled JavaScript is what actually gets executed. If you start the project HTTP server with `npm start`, you should see the fully functional application in your browser.

Now that you have TypeScript though, you can start benefiting from some of its features. There is a lot of value the language can provide to AngularJS applications.

For one thing, TypeScript is a superset of ES2015. Any application that has previously been written in ES5 —like the PhoneCat example has— can with TypeScript start incorporating all of the JavaScript features that are new to ES2015. These include things like `let`s and `const`s, arrow functions, default function parameters, and destructuring assignments.

Another thing you can do is start adding *type safety* to your code. This has actually partially already happened because of the AngularJS typings you installed. TypeScript are checking that you are calling AngularJS APIs correctly when you do things like register components to Angular modules.

But you can also start adding *type annotations* to get even more out of type system of TypeScript. For instance, you can annotate the checkmark filter so that it explicitly expects booleans as arguments. This makes it clearer what the filter is supposed to do.

In the `Phone` service, you can explicitly annotate the `$resource` service dependency as an `angular.resource.IResourceService` - a type defined by the AngularJS typings.

You can apply the same trick to the route configuration file of the application in `app.config.ts`, where you are using the location and route services. By annotating them accordingly TypeScript can verify you're calling their APIs with the correct kinds of arguments.

The AngularJS 1.x type definitions you installed are not officially maintained by the Angular team, but are quite comprehensive. It is possible to make an AngularJS 1.x application fully type-annotated with the help of these definitions.

If this is something you wanted to do, it would be a good idea to enable the `noImplicitAny` configuration option in `tsconfig.json`. This would cause the TypeScript compiler to display a warning when there is any code that does not yet have type annotations. You could use it as a guide to inform us about how close you are to having a fully annotated project.

Another TypeScript feature you can make use of is *classes*. In particular, you can turn component controllers into classes. That way they'll be a step closer

to becoming Angular component classes, which will make life easier once you upgrade.

AngularJS expects controllers to be constructor functions. That is exactly what ES2015/TypeScript classes are under the hood, so that means you can just plug in a class as a component controller and AngularJS will happily use it.

Here is what the new class for the phone list component controller looks like:

What was previously done in the controller function is now done in the class constructor function. The dependency injection annotations are attached to the class using a static property `$inject`. At runtime this becomes the `PhoneListController.$inject` property.

The class additionally declares three members: The array of phones, the name of the current sort key, and the search query. These are all things you have already been attaching to the controller but that weren't explicitly declared anywhere. The last one of these isn't actually used in the TypeScript code since it is only referred to in the template, but for the sake of clarity you should define all of the controller members.

In the Phone detail controller, you'll have two members: One for the phone that the user is looking at and another for the URL of the currently displayed image:

This makes the controller code look a lot more like Angular already. You're all set to actually introduce Angular into the project.

If you had any AngularJS services in the project, those would also be a good candidate for converting to classes, since like controllers, they're also constructor functions. But you only have the `Phone` factory in this project, and that is a bit special since it is an `ngResource` factory. So you won't be doing anything to it in the preparation stage. You'll instead turn it directly into an Angular service.

Installing Angular

Having completed the preparation work, get going with the Angular upgrade of PhoneCat. You'll do this incrementally with the help of ngUpgrade that comes with Angular. By the time you're done, you'll be able to remove AngularJS from the project completely, but the key is to do this piece by piece without breaking the application.

The project also contains some animations. You won't upgrade them in this version of the guide. Turn to the Angular animations guide to learn about that.

Install Angular into the project, along with the SystemJS module loader. Take a look at the results of the upgrade setup instructions and get the following configurations from there:

- Add Angular and the other new dependencies to `package.json`
- The SystemJS configuration file `systemjs.config.js` to the project root directory.

Once these are done, run:

npm install

Soon you can load Angular dependencies into the application inside `index.html`, but first you need to do some directory path adjustments. You'll need to load files from `node_modules` and the project root instead of from the `/app` directory as you've been doing to this point.

Move the `app/index.html` file to the project root directory. Then change the development server root path in `package.json` to also point to the project root instead of `app`:

"start": "http-server ./ -a localhost -p 8000 -c-1",

Now you're able to serve everything from the project root to the web browser. But you do *not* want to have to change all the image and data paths used in the application code to match the development setup. For that reason, you'll add a `<base>` tag to `index.html`, which will cause relative URLs to be resolved back to the `/app` directory:

Now you can load Angular using SystemJS. You'll add the Angular polyfills and the SystemJS configuration to the end of the `<head>` section, and then you'll use `System.import` to load the actual application:

You also need to make a couple of adjustments to the `systemjs.config.js` file installed during upgrade setup.

Point the browser to the project root when loading things through SystemJS, instead of using the `<base>` URL.

Install the `upgrade` package using `npm install @angular/upgrade --save` and add a mapping for the `@angular/upgrade/static` package.

Creating the *AppModule*

Now create the root `NgModule` class called `AppModule`. There is already a file named `app.module.ts` that holds the AngularJS module. Rename it to `app.module.ajs.ts` and update the corresponding script name in the `index.html` as well. The file contents remain:

Now create a new `app.module.ts` with the minimum `NgModule` class:

Bootstrapping a hybrid PhoneCat

Next, you'll bootstrap the application as a *hybrid application* that supports both AngularJS and Angular components. After that, you can start converting the individual pieces to Angular.

The application is currently bootstrapped using the AngularJS `ng-app` directive attached to the `<html>` element of the host page. This will no longer work in the hybrid application. Switch to the ngUpgrade bootstrap method instead.

First, remove the `ng-app` attribute from `index.html`. Then import `UpgradeModule` in the `AppModule`, and override its `ngDoBootstrap` method:

You are bootstrapping the AngularJS module from inside `ngDoBootstrap`. The arguments are the same as you would pass to `angular.bootstrap` if you were manually bootstrapping AngularJS: the root element of the application; and an array of the AngularJS 1.x modules that you want to load.

Finally, bootstrap the `AppModule` in `app/main.ts`. This file has been configured as the application entrypoint in `systemjs.config.js`, so it is already being loaded by the browser.

Now you're running both AngularJS and Angular at the same time. That is pretty exciting! You're not running any actual Angular components yet. That is next.

Why declare *angular* as *angular.IAngularStatic*?

`@types/angular` is declared as a UMD module, and due to the way UMD typings work, once you have an ES6 `import` statement in a file all UMD typed modules must also be imported using `import` statements instead of being globally available.

AngularJS is currently loaded by a script tag in `index.html`, which means that the whole app has access to it as a global and uses the same instance of the `angular` variable. If you used `import * as angular from 'angular'` instead, you'd also have to load every file in the AngularJS application to use ES2015 modules in order to ensure AngularJS was being loaded correctly.

This is a considerable effort and it often isn't worth it, especially since you are in the process of moving your code to Angular. Instead, declare `angular` as `angular.IAngularStatic` to indicate it is a global variable and still have full typing support.

Manually create a UMD bundle for your Angular application

Starting with Angular version 13, the distribution format no longer includes UMD bundles.

If your use case requires the UMD format, use `rollup` to manually produce a bundle from the flat ES module files.

1. Use `npm` to globally install `rollup`

   npm i -g rollup

2. Output the version of `rollup` and verify the installation was successful

   rollup -v

3. Create the `rollup.config.js` configuration file for `rollup` to use the global `ng` command to reference all of the Angular framework exports.

   1. Create a file named `rollup.config.js`

2. Copy the following content into `rollup.config.js`

   export default { input: 'node_modules/@angular/core/fesm2015/core.js', output: { file: 'bundle.js', format: 'umd', name: 'ng' } }

4. Use `rollup` to create the `bundle.js` UMD bundle using settings in `rollup.config.js`

   rollup -c rollup.config.js

The `bundle.js` file contains your UMD bundle. For an example on GitHub, see UMD Angular bundle.

Upgrading the Phone service

The first piece you'll port over to Angular is the `Phone` service, which resides in `app/core/phone/phone.service.ts` and makes it possible for components to load phone information from the server. Right now it is implemented with ngResource and you're using it for two things:

- For loading the list of all phones into the phone list component.
- For loading the details of a single phone into the phone detail component.

You can replace this implementation with an Angular service class, while keeping the controllers in AngularJS land.

In the new version, you import the Angular HTTP module and call its `HttpClient` service instead of `ngResource`.

Re-open the `app.module.ts` file, import and add `HttpClientModule` to the `imports` array of the `AppModule`:

Now you're ready to upgrade the Phone service itself. Replace the ngResource-based service in `phone.service.ts` with a TypeScript class decorated as `@Injectable`:

The `@Injectable` decorator will attach some dependency injection metadata to the class, letting Angular know about its dependencies. As described by the Dependency Injection Guide, this is a marker decorator you need to use for classes that have no other Angular decorators but still need to have their dependencies injected.

In its constructor the class expects to get the `HttpClient` service. It will be injected to it and it is stored as a private field. The service is then used in the two instance methods, one of which loads the list of all phones, and the other loads the details of a specified phone:

The methods now return observables of type `PhoneData` and `PhoneData[]`. This is a type you don't have yet. Add a simple interface for it:

`@angular/upgrade/static` has a `downgradeInjectable` method for the purpose of making Angular services available to AngularJS code. Use it to plug in

the `Phone` service:

Here is the full, final code for the service:

Notice that you're importing the `map` operator of the RxJS `Observable` separately. Do this for every RxJS operator.

The new `Phone` service has the same features as the original, `ngResource`-based service. Because it is an Angular service, you register it with the `NgModule` providers:

Now that you are loading `phone.service.ts` through an import that is resolved by SystemJS, you should **remove the <script> tag** for the service from `index.html`. This is something you'll do to all components as you upgrade them. Simultaneously with the AngularJS to Angular upgrade you're also migrating code from scripts to modules.

At this point, you can switch the two components to use the new service instead of the old one. While you `$inject` it as the downgraded `phone` factory, it is really an instance of the `Phone` class and you annotate its type accordingly:

Now there are two AngularJS components using an Angular service! The components don't need to be aware of this, though the fact that the service returns observables and not promises is a bit of a giveaway. In any case, what you've achieved is a migration of a service to Angular without having to yet migrate the components that use it.

You could use the `toPromise` method of `Observable` to turn those observables into promises in the service. In many cases that reduce the number of changes to the component controllers.

Upgrading Components

Upgrade the AngularJS components to Angular components next. Do it one component at a time while still keeping the application in hybrid mode. As you make these conversions, you'll also define your first Angular *pipes*.

Look at the phone list component first. Right now it contains a TypeScript controller class and a component definition object. You can morph this into an Angular component by just renaming the controller class and turning the AngularJS component definition object into an Angular `@Component` decorator. You can then also remove the static `$inject` property from the class:

The `selector` attribute is a CSS selector that defines where on the page the component should go. In AngularJS you do matching based on component names, but in Angular you have these explicit selectors. This one will match elements with the name `phone-list`, just like the AngularJS version did.

Now convert the template of this component into Angular syntax. The search controls replace the AngularJS `$ctrl` expressions with the two-way `[(ngModel)]` binding syntax of Angular:

Replace the `ng-repeat` of the list with an `*ngFor` as described in the Template Syntax page. Replace the `ng-src` of the image tag with a binding to the native `src` property.

No Angular *filter* or *orderBy* filters The built-in AngularJS `filter` and `orderBy` filters do not exist in Angular, so you need to do the filtering and sorting yourself.

You replaced the `filter` and `orderBy` filters with bindings to the `getPhones()` controller method, which implements the filtering and ordering logic inside the component itself.

Now you need to downgrade the Angular component so you can use it in AngularJS. Instead of registering a component, you register a `phoneList` *directive*, a downgraded version of the Angular component.

The `as angular.IDirectiveFactory` cast tells the TypeScript compiler that the return value of the `downgradeComponent` method is a directive factory.

The new `PhoneListComponent` uses the Angular `ngModel` directive, located in the `FormsModule`. Add the `FormsModule` to `NgModule` imports and declare the new `PhoneListComponent` since you downgraded it:

Remove the <script> tag for the phone list component from `index.html`.

Now set the remaining `phone-detail.component.ts` as follows:

This is similar to the phone list component. The new wrinkle is the `RouteParams` type annotation that identifies the `routeParams` dependency.

The AngularJS injector has an AngularJS router dependency called `$routeParams`, which was injected into `PhoneDetails` when it was still an AngularJS controller. You intend to inject it into the new `PhoneDetailsComponent`.

Unfortunately, AngularJS dependencies are not automatically available to Angular components. You must upgrade this service using a factory provider to make `$routeParams` an Angular injectable. Do that in a new file called `ajs-upgraded-providers.ts` and import it in `app.module.ts`:

Convert the phone detail component template into Angular syntax as follows:

There are several notable changes here:

- You've removed the `$ctrl.` prefix from all expressions.
- You've replaced `ng-src` with property bindings for the standard `src` property.
- You're using the property binding syntax around `ng-class`. Though Angular does have a very similar `ngClass` as AngularJS does, its value is not magically evaluated as an expression. In Angular, you always specify

in the template when the value of an attribute is a property expression, as opposed to a literal string.

- You've replaced `ng-repeat`s with `*ngFor`s.
- You've replaced `ng-click` with an event binding for the standard `click`.
- You've wrapped the whole template in an `ngIf` that causes it only to be rendered when there is a phone present. You need this because when the component first loads, you don't have `phone` yet and the expressions will refer to a non-existing value. Unlike in AngularJS, Angular expressions do not fail silently when you try to refer to properties on undefined objects. You need to be explicit about cases where this is expected.

Add `PhoneDetailComponent` component to the `NgModule` *declarations*:

You should now also remove the phone detail component <script> tag from `index.html`.

Add the *CheckmarkPipe* The AngularJS directive had a `checkmark` *filter*. Turn that into an Angular **pipe**.

There is no upgrade method to convert filters into pipes. You won't miss it. It is easy to turn the filter function into an equivalent Pipe class. The implementation is the same as before, repackaged in the `transform` method. Rename the file to `checkmark.pipe.ts` to conform with Angular conventions:

Now import and declare the newly created pipe and remove the filter <script> tag from `index.html`:

AOT compile the hybrid app

To use AOT with a hybrid app, you have to first set it up like any other Angular application, as shown in the Ahead-of-time Compilation chapter.

Then change `main-aot.ts` to bootstrap the `AppComponentFactory` that was generated by the AOT compiler:

You need to load all the AngularJS files you already use in `index.html` in `aot/index.html` as well:

These files need to be copied together with the polyfills. The files the application needs at runtime, like the `.json` phone lists and images, also need to be copied.

Install `fs-extra` using `npm install fs-extra --save-dev` for better file copying, and change `copy-dist-files.js` to the following:

And that is all you need to use AOT while upgrading your app!

Adding The Angular Router And Bootstrap

At this point, you've replaced all AngularJS application components with their Angular counterparts, even though you're still serving them from the AngularJS router.

Add the Angular router Angular has an all-new router.

Like all routers, it needs a place in the UI to display routed views. For Angular that is the `<router-outlet>` and it belongs in a *root component* at the top of the applications component tree.

You don't yet have such a root component, because the application is still managed as an AngularJS app. Create a new `app.component.ts` file with the following `AppComponent` class:

It has a template that only includes the `<router-outlet>`. This component just renders the contents of the active route and nothing else.

The selector tells Angular to plug this root component into the `<phonecat-app>` element on the host web page when the application launches.

Add this `<phonecat-app>` element to the `index.html`. It replaces the old AngularJS `ng-view` directive:

Create the *Routing Module* A router needs configuration whether it is the AngularJS or Angular or any other router.

The details of Angular router configuration are best left to the Routing documentation which recommends that you create a `NgModule` dedicated to router configuration (called a *Routing Module*).

This module defines a `routes` object with two routes to the two phone components and a default route for the empty path. It passes the `routes` to the `RouterModule.forRoot` method which does the rest.

A couple of extra providers enable routing with "hash" URLs such as `#!/phones` instead of the default "push state" strategy.

Now update the `AppModule` to import this `AppRoutingModule` and also the declare the root `AppComponent` as the bootstrap component. That tells Angular that it should bootstrap the application with the *root* `AppComponent` and insert its view into the host web page.

You must also remove the bootstrap of the AngularJS module from `ngDoBootstrap()` in `app.module.ts` and the `UpgradeModule` import.

And since you are routing to `PhoneListComponent` and `PhoneDetailComponent` directly rather than using a route template with a `<phone-list>` or `<phone-detail>` tag, you can do away with their Angular selectors as well.

Generate links for each phone You no longer have to hardcode the links to phone details in the phone list. You can generate data bindings for the `id` of each phone to the `routerLink` directive and let that directive construct the appropriate URL to the `PhoneDetailComponent`:

See the Routing page for details.

Use route parameters The Angular router passes route parameters differently. Correct the `PhoneDetail` component constructor to expect an injected `ActivatedRoute` object. Extract the `phoneId` from the `ActivatedRoute.snapshot.params` and fetch the phone data as before:

You are now running a pure Angular application!

Say Goodbye to AngularJS

It is time to take off the training wheels and let the application begin its new life as a pure, shiny Angular app. The remaining tasks all have to do with removing code - which of course is every programmer's favorite task!

The application is still bootstrapped as a hybrid app. There is no need for that anymore.

Switch the bootstrap method of the application from the `UpgradeModule` to the Angular way.

If you haven't already, remove all references to the `UpgradeModule` from `app.module.ts`, as well as any factory provider for AngularJS services, and the `app/ajs-upgraded-providers.ts` file.

Also remove any `downgradeInjectable()` or `downgradeComponent()` you find, together with the associated AngularJS factory or directive declarations.

You may also completely remove the following files. They are AngularJS module configuration files and not needed in Angular:

- `app/app.module.ajs.ts`
- `app/app.config.ts`
- `app/core/core.module.ts`
- `app/core/phone/phone.module.ts`
- `app/phone-detail/phone-detail.module.ts`
- `app/phone-list/phone-list.module.ts`

The external typings for AngularJS may be uninstalled as well. The only ones you still need are for Jasmine and Angular polyfills. The `@angular/upgrade` package and its mapping in `systemjs.config.js` can also go.

npm uninstall @angular/upgrade –save npm uninstall @types/angular @types/angular-animate @types/angular-cookies @types/angular-mocks @types/angular-resource @types/angular-route @types/angular-sanitize –save-dev

Finally, from `index.html`, remove all references to AngularJS scripts and jQuery. When you're done, this is what it should look like:

That is the last you'll see of AngularJS! It has served us well but now it is time to say goodbye.

Appendix: Upgrading PhoneCat Tests

Tests can not only be retained through an upgrade process, but they can also be used as a valuable safety measure when ensuring that the application does not break during the upgrade. E2E tests are especially useful for this purpose.

E2E Tests

The PhoneCat project has both E2E Protractor tests and some Karma unit tests in it. Of these two, E2E tests can be dealt with much more easily: By definition, E2E tests access the application from the *outside* by interacting with the various UI elements the application puts on the screen. E2E tests aren't really that concerned with the internal structure of the application components. That also means that, although you modify the project quite a bit during the upgrade, the E2E test suite should keep passing with just minor modifications. You didn't change how the application behaves from the user's point of view.

During TypeScript conversion, there is nothing to do to keep E2E tests working. But when you change the bootstrap to that of a Hybrid app, you must make a few changes.

Update the `protractor-conf.js` to sync with hybrid applications:

ng12Hybrid: true

When you start to upgrade components and their templates to Angular, you'll make more changes because the E2E tests have matchers that are specific to AngularJS. For PhoneCat you need to make the following changes in order to make things work with Angular:

Previous code	New code	Notes
`by.repeater('phone in $ctrl.phones').column('phone.name')`	`by.css('.phones .name')`	The repeater matcher relies on AngularJS `ng-repeat`
`by.repeater('phone in $ctrl.phones')`	`by.css('.phones li')`	The repeater matcher relies on AngularJS `ng-repeat`
`by.model('$ctrl.query')`	`by.css('input')`	The model matcher relies on AngularJS `ng-model`

Previous code	New code	Notes
`by.model('$ctrl.orderProp')`	`by.css('select')`	The model matcher relies on AngularJS `ng-model`
`by.binding('$ctrl.phone.name')`	`by.css('h1')`	The binding matcher relies on AngularJS data binding

When the bootstrap method is switched from that of `UpgradeModule` to pure Angular, AngularJS ceases to exist on the page completely. At this point, you need to tell Protractor that it should not be looking for an AngularJS application anymore, but instead it should find *Angular apps* from the page.

Replace the `ng12Hybrid` previously added with the following in `protractor-conf.js`:

useAllAngular2AppRoots: true,

Also, there are a couple of Protractor API calls in the PhoneCat test code that are using the AngularJS `$location` service under the hood. As that service is no longer present after the upgrade, replace those calls with ones that use the generic URL APIs of WebDriver instead. The first of these is the redirection spec:

And the second is the phone links spec:

Unit Tests

For unit tests, on the other hand, more conversion work is needed. Effectively they need to be *upgraded* along with the production code.

During TypeScript conversion no changes are strictly necessary. But it may bea good idea to convert the unit test code into TypeScript as well.

For instance, in the phone detail component spec, you can use ES2015 features like arrow functions and block-scoped variables and benefit from the type definitions of the AngularJS services you're consuming:

Once you start the upgrade process and bring in SystemJS, configuration changes are needed for Karma. You need to let SystemJS load all the new Angular code, which can be done with the following kind of shim file:

The shim first loads the SystemJS configuration, then the test the support libraries of Angular, and then the spec files of the application themselves.

Karma configuration should then be changed so that it uses the application root dir as the base directory, instead of `app`.

Once done, you can load SystemJS and other dependencies, and also switch the configuration for loading application files so that they are *not* included to the page by Karma. You'll let the shim and SystemJS load them.

Since the HTML templates of Angular components will be loaded as well, you must help Karma out a bit so that it can route them to the right paths:

The unit test files themselves also need to be switched to Angular when their production counterparts are switched. The specs for the checkmark pipe are probably the most straightforward, as the pipe has no dependencies:

The unit test for the phone service is a bit more involved. You need to switch from the mocked-out AngularJS `$httpBackend` to a mocked-out Angular Http backend.

For the component specs, you can mock out the `Phone` service itself, and have it provide canned phone data. You use the component unit testing APIs of Angular for both components.

Finally, revisit both of the component tests when you switch to the Angular router. For the details component, provide a mock of Angular `ActivatedRoute` object instead of using the AngularJS `$routeParams`.

And for the phone list component, a few adjustments to the router make the `RouteLink` directives work.

@reviewed 2021-11-02