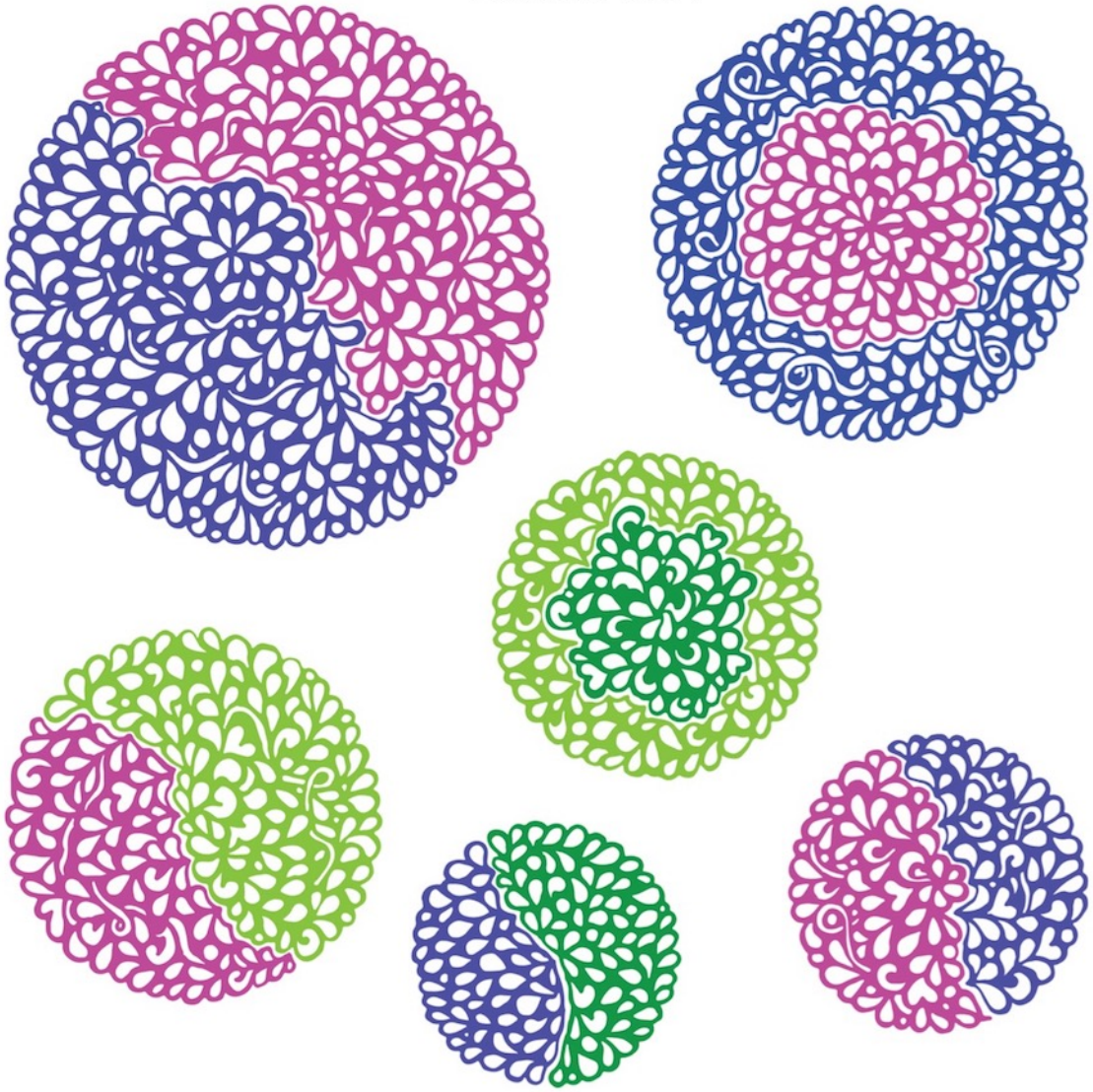
UPGRADING ANGULAR APPLICATIONS

VICTOR SAVKIN



Upgrading Angular Applications

Victor Savkin and Nrwl.io

This book is for sale at http://leanpub.com/ngupgrade

This version was published on 2017-05-25

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Introduction

Nrwl.io - Angular consulting for enterprise customers, from core team members

Victor is a co-founder of Nrwl, a company providing Angular consulting for enterprise customers, from core team members. Visit nrwl.io[1] for more information.

About the Author

Victor Savkin is a core contributor to the Angular project. He has been on the Angular team since the inception of Angular 2.x and developed dependency injection, change detection, forms, and the router. After founding Nrwl Victor has been working with many companies helping them upgrade their AngularJS applications.

What is this book about?

Many organizations have large AngularJS 1.x applications deployed to production. These applications may be built by multiple teams from different lines of business. They may use different routers and state management strategies. Rewriting such applications all at once, or big bang migrations, is not just impractical, it is often impossible, and mainly for business reasons. We need to do gradually.

[1] http://nrwl.io

To help with the upgrade process the Angular team built NgUpgrade, a library for mix-and-matching AngularJS and Angular components. The question now is how we can use this library in the most advantageous way? What patterns and strategies should we use?

The book explores NgUpgrade in depth, including the mental model, implementation, subtleties of the API. It also talks about different strategies for upgrading large AngularJS applications.

AngularJS and Angular

This confuses a lot of folks, so let's clarify it right away.

For a while we referred to the new version of the Angular framework as Angular 2. So we would always talk about upgrading from Angular 1 to Angular 2. Since the core team embraced semantic versioning, this stopped making sense. We already have Angular 4, and Angular 5 is right around the corner. So what the team decided to do instead is to refer to all 1.x versions of the framework as AngularJS, and refer to all the versions starting with 2.x as Angular. So this book is about upgrading from AngularJS to Angular.

Finally, let's refer to the application being upgraded as hybrid.

Chapter 1: Preparation

Before we embark on introducing Angular into our code base, it is good idea to do some preparatory work to make this process easier. Should we switch to TypeScript? Should we write AngularJS code in a certain way? These are the questions we are going to explore in this chapter.

Angular Style

AngularJS and Angular share some core fundamentals: they use HTML, dependency injection, and components. So in many ways the two frameworks are very similar, but, at the same time, they use these capabilities differently.

- Angular applications are built of components. The framework also supports directives, which we can use to augment the behavior of elements or components, but comparing to AngularJS they aren't used nearly as often.
- Components in Angular are encapsulated, so the data does not gets inherited from the parent's scope. Instead, we have to use inputs and outputs to pass the data in an out.
- Angular still compiles HTML, but it is no longer done in an ad-hoc fashion. We only compile components, and we do it ahead of time, as part of our build process.
- Angular is built on top of TypeScript and clearly separates the static parts of our applications (stored in decorators) from the dynamic parts.

Writing AngularJS Applications in the Angular Style

We can write AngularJS applications in the Angular style (outlined above), especially if we use AngularJS 1.5+, which added `angular.component` and the `$onInit()`, `$onDestroy()`, and `$onChanges()` life-cycle events. This is what it looks like:

```
angular.component('myComponent', {
 bindings: {
   myInput: '<',
   myOutput: '&'
 },
 template: `
   <h2>{{$ctrl.input}}!</h2>
   <button ng-click="$ctrl.onEvent()">Trigger Action!</button>
 `,
 controller: function() {
   this.onEvent = () => {
     this.myOutput('some data');
   };
 }
});
```

As you can see, we do not use `compile` or `link`. We also use isolated scope and explicitly pass data in and out.

We can go even further and rewrite our controller as a class.

```
class MyComponent {
 myInput: string;
 myOutput: (s: string) => void;

 onEvent() {
   this.myOutput('some data');
 }
}

angular.component('myComponent', {
 bindings: {
   myInput: '<',
   myOutput: '&'
 },
 template: `
   <h2>{{$ctrl.input}}!</h2>
   <button ng-click="$ctrl.onEvent()">Trigger Action!</button>
 `,
 controller: MyComponent
});
```

New Development in the Angular Style

Often while upgrading an application, we keep developing new features for it in AngularJS. It is a good idea to do new development in the Angular style to make the coming upgrade process a little bit easier.

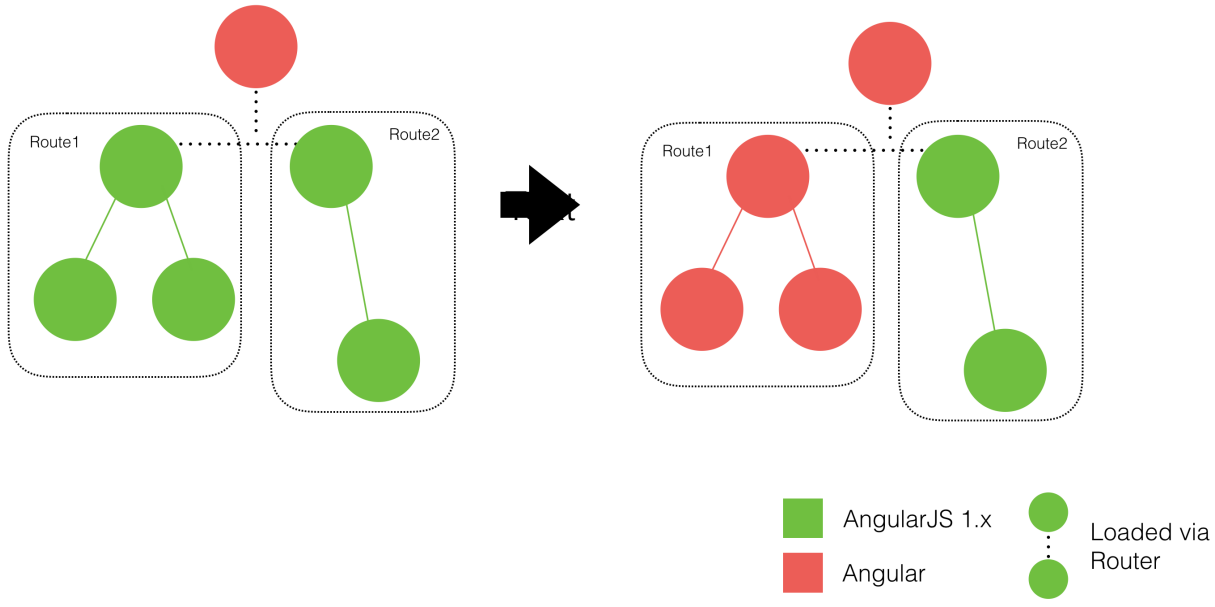
Changing Existing Code to the Angular Style

Whether you should update your existing code to the Angular style is a harder question, and the answer for it depends on what approach to upgrade you will choose.

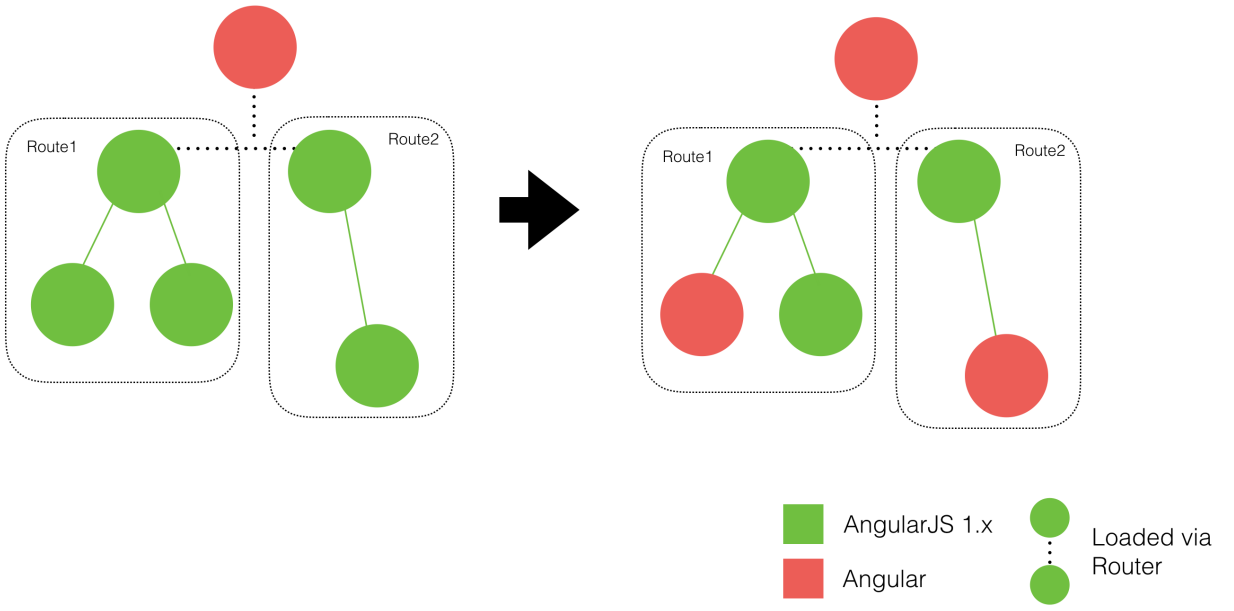
Two Approaches to Upgrade

Horizontal slicing and vertical slicing are the two main approaches to upgrade (see Chapter 4 for more information).

When using vertical slicing, we upgrade our application route by route, or screen by screen.



When using horizontal slicing, we start with upgrading reusable components, and work our way up to the root of the component tree.



The main downside of vertical slicing is that it is coarse-grained. The upside is that it works well with pretty much any AngularJS application out there, regardless of how it is written. So we don't have to "prepare" our AngularJS application when using this strategy as the number of integration points (the places where we have to upgrade or downgrade components) between the two frameworks is minimal.

When using horizontal slicing, the number of integration points is much larger. As a result, it is not a bad idea to update the components we want to upgrade first to the Angular style. Otherwise, we won't be able to use NgUpgrade to upgrade those components.

Should I switch to TypeScript?

This is a business decision that highly depends on the structure of your organization. If we have multiple teams working on the same project, and your team embarks on upgrading a part of this project, switching to TypeScript is not a good option. By doing this we impose a new technology on many teams who might not be even thinking about the upgrade. If, however, there is one team working on the project, it might be a good idea to give it a try.

Should I update the whole codebase to the Angular style before upgrading to Angular?

I don't think this is ever a good idea. There is just no good business reason to do it. Even if we use horizontal slicing, which I would discourage, we still need to update only certain components. And even then I would not update all of them at once, but as we move forward with the upgrade process.

Chapter 2: NgUpgrade in Depth

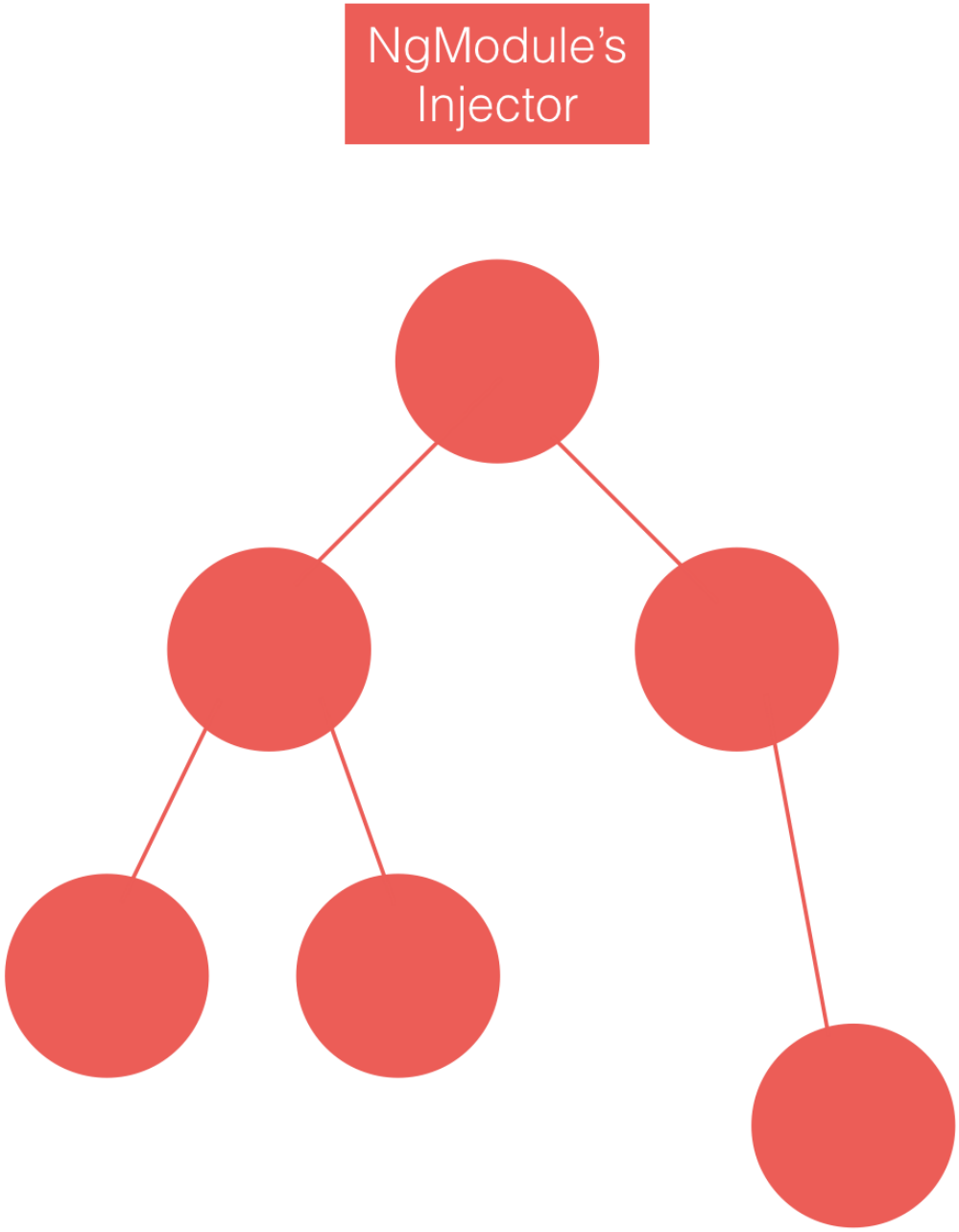
NgUpgrade is a library put together by the Angular team, which we can use in our applications to mix and match AngularJS and Angular components and bridge the AngularJS and Angular dependency injection systems. In this chapter we will look at what it is and how it works.

How to Use it Best

This chapter only talks about the mechanics of of the library, and not how to use it in the best way. That's covered in the rest of the book.

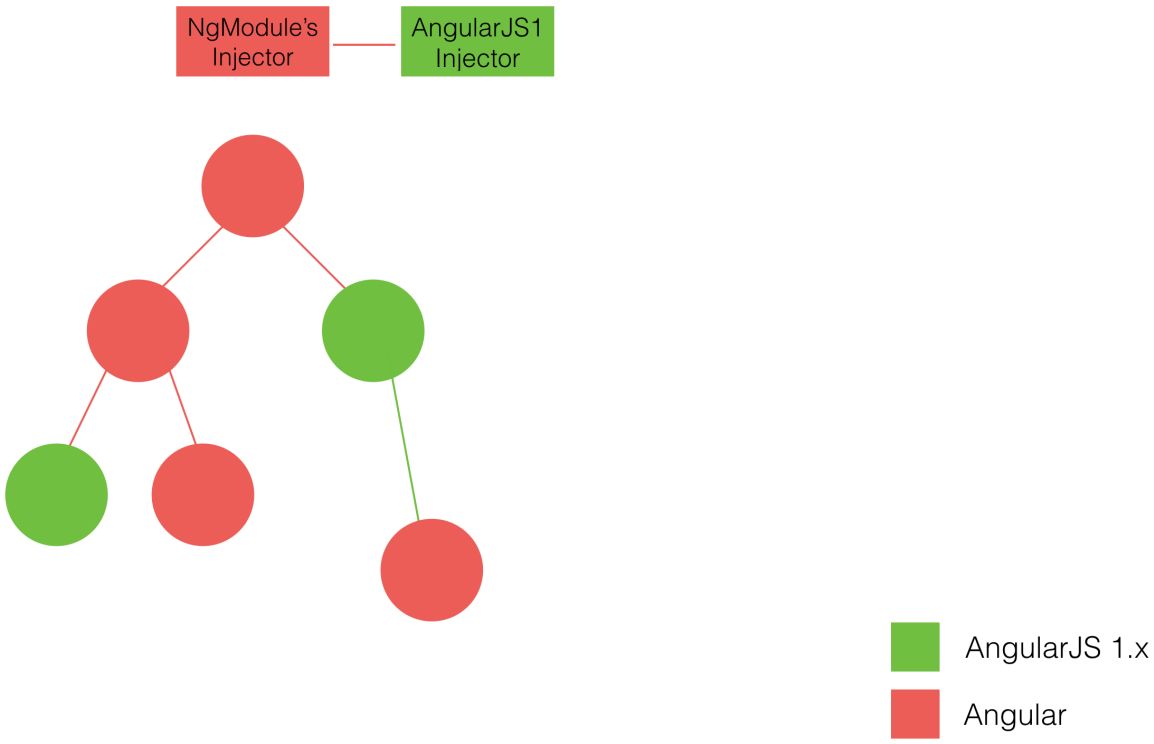
How it Works

An Angular application is a tree of components, with each of them having an injector. Plus there is an injector for the NgModule of the application. When trying to resolve a dependency for a particular component, the framework will first try to get it from the component tree. And if the dependency cannot be found, Angular will get it from the NgModule's injector.



If the application uses lazy loading, the diagram will look more complex, but we will cover this use case in the chapter about lazy loading and NgUpgrade.

Using NgUpgrade we can bootstrap an existing AngularJS application from an Angular application. And we do it in such a way that we can mix-and-match components written in the two frameworks and bridge the two DI systems. We can consider such an application a “hybrid application”.



Bootstrapping

The easiest way to bootstrap such a hybrid application is to have an NgModule without any bootstrap components. Instead it defines `ngDoBootstrap` where we use the injected `UpgradeModule` to bootstrap the AngularJS application.

```
@Component({
 selector: 'app-root',
 templateUrl: './app.component.html'
})
export class AppComponent {}
// Access global AngularJS 1.x object
const m = angular.module('AngularJsModule', []);
m.directive('appRoot', downgradeComponent({component: AppComponent}));

```

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

 ngDoBootstrap() {
   this.upgrade.bootstrap(document.body, ['AngularJsModule']);
 }
}
```

This is a good default because components upgraded from AngularJS to Angular require an AngularJS ancestor component, and this way of bootstrapping guarantees that. This, however, does not work in certain situations. For instance, it won't work if we load and bootstrap the AngularJS application lazily, only when the user navigates to a certain route. In this case we need to bootstrap our hybrid differently, by doing it in a lazy-loaded component.

UpgradeModule.bootstrap

`UpgradeModule.bootstrap` has the same signature as `angular.bootstrap`. And if we look at the implementation, we will see that it actually calls `angular.bootstrap` under the hood, but it does it with a few tweaks:

- It makes sure `angular.bootstrap` runs in the right zone.
- It adds an extra module that sets up AngularJS to be visible in Angular and vice versa.
- It adapts the testability APIs to make Protractor work with hybrid apps.

Capturing AngularJS and Lazy Loading

One thing that may not be obvious is that importing `@angular/upgrade/static` captures `window.angular`. And that's why we have to import the AngularJS framework before we import '@angular/upgrade/static'.

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

Otherwise we'll see the `AngularJS v1.x is not loaded` error.

This works well for simple applications, but is problematic for complex enterprise applications, where, for instance, AngularJS can be loaded lazily via `requirejs`. To make it work there, we can manually reset AngularJS, as follows:

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

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

ngDoBootstrap() {
  requirejs(['angular', 'angularJsApp'], (angular, app) => {
   setAngularJSGlobal(angular);
   this.upgrade.bootstrap(document, ['AngularJsAppModule']);
  });
}
}
```

Using '@angular/upgrade/static'or '@angular/upgrade'?

For historical reasons NgUpgrade has two entry points: '@angular/upgrade' and '@angular/upgrade/static'. Use '@angular/upgrade/static'. It provides better error reporting and works in the AOT mode.

Fixing module resolution

Depending on our application's build setup, we may need to point the bundler to the right UMD bundle. For instance, this is how you do it for webpack:

```
resolve: {
 alias: {
   "@angular/upgrade/static": "@angular/upgrade/bundles/upgrade-static.umd.js"
 }
}
```

We have learned how to bootstrap a hybrid application. Now let's see how we can bridge the AngularJS and Angular dependency injection systems.

Dependency Injection

An important part of the upgrade process is moving services (or to use a better word, "injectables") from AngularJS to Angular. Usually we don't have to make any of the changes to the injectables themselves–we just need to make sure they are properly wired up in the DI system. This often requires us to either access AngularJS injectables in Angular or access Angular injectables in AngularJS.

Accessing AngularJS Injectables in Angular

Say our AngularJS application has the following injectable:

```
const m = angular.module('AngularJsModule', []);
m.value('angularJsInjectable', 'angularJsInjectable-value');
```

We can access it in the Angular part of the application via `$injector`, like this:

```
function needsAngularJsInjectableFactory($injector) {
 return `needsAngularJsInjectable got ${$injector.get('angularJsInjectable')}`;
}

@NgModule({
 imports: [
   BrowserModule,
   UpgradeModule
 ],
 providers: [
   {
     provide: 'needsAngularJsInjectable',
     useFactory: needsAngularJsInjectableFactory,
     deps: ['$injector'] // $injector is provided by UpgradeModule
```

```
    }
  ]
})
export class AppModule {
  constructor(private upgrade: UpgradeModule) {}

  ngDoBootstrap() {
    this.upgrade.bootstrap(document.body, ['AngularJsModule']);
    console.log(this.upgrade.injector.get('needsAngularJsInjectable'));
  }
}
```

`UpgradeModule` brings in `$injector` (the injector of our AngularJS application), and that's why we can access it in `AppModule` or any of its children.

Note that `$injector` gets defined by the `upgrade.bootstrap` call. If we try to get it before calling `bootstrap`, we will see the `Cannot read property 'get' of undefined` error.

```
@NgModule({
  imports: [
    BrowserModule,
    UpgradeModule
  ],
  providers: [
    {
      provide: 'needsAngularJsInjectable',
      useFactory: needsAngularJsInjectableFactory,
      deps: ['$injector'] // $injector is provided by UpgradeModule
    }
  ]
})
export class AppModule {
  constructor(private upgrade: UpgradeModule) {}

  ngDoBootstrap() {
    console.log(this.upgrade.injector.get('needsAngularJsInjectable')); // throws
    this.upgrade.bootstrap(document.body, ['AngularJsModule']);
  }
}
```

Accessing Angular Injectables in AngularJS

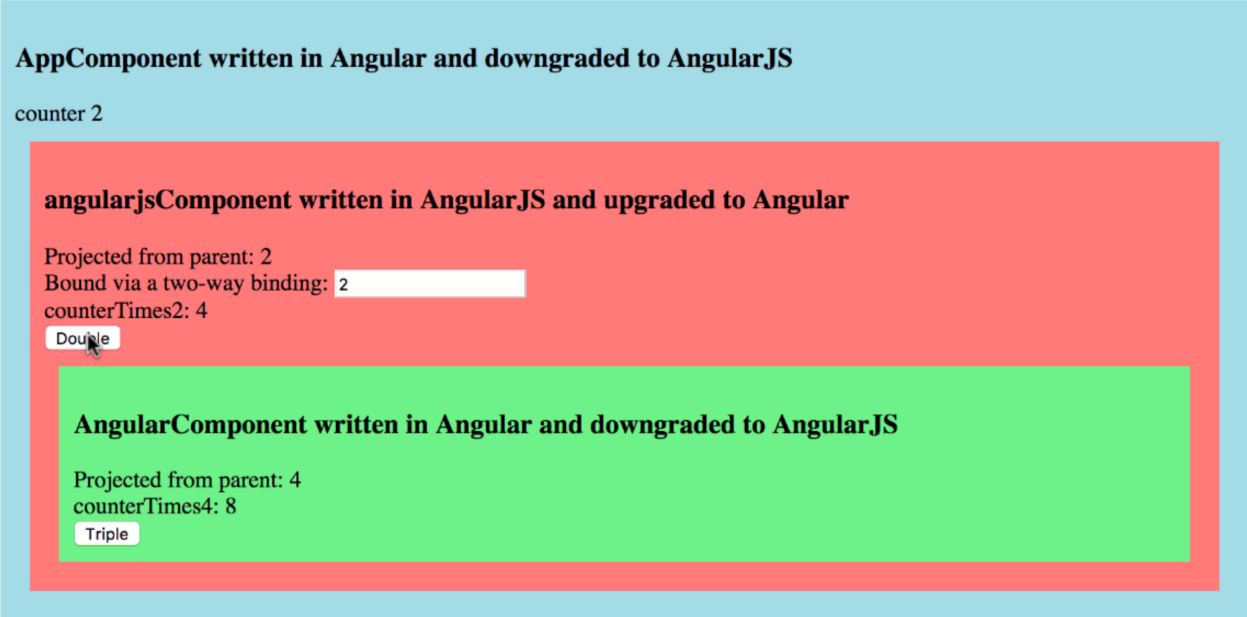
We can also make Angular injectables available in the AngularJS part of our application, like this:

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

export class AngularInjectable {
 get value() { return 'angularInjectable-value'; }
}

const m = angular.module('AngularJsModule', []);
m.factory('angularInjectable', downgradeInjectable(AngularInjectable));
m.factory('needsAngularInjectable', (angularInjectable: AngularInjectable) => `n\
eedsAngularInjectable [got ${angularInjectable.value}]`);

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

 ngDoBootstrap() {
   this.upgrade.bootstrap(document.body, ['AngularJsModule']);
   console.log(this.upgrade.$injector.get('needsAngularInjectable')); // 'angula\
rInjectable-value'
 }
}
```

There is more going on here. This is because the Angular DI system allows us to use any token (e.g., types) to express dependencies between injectables. But in AngularJS we can only use strings for that. By doing this `m.factory('angularInjectable', downgradeInjectable(AngularInjectable))` we map `AngularInjectable` to the `'angularInjectable'` string. We then use it to instantiate `needsAngularInjectable`.

Component Integration

Another important capability provided by NgUpgrade is being able to mix-and-match AngularJS and Angular components.To demonstrate how it works, we'll look at the following example.



There are three components in this example: `AppComponent` > `angularJSComponent` > `AngularComponent`.

- `AppComponent` is written in Angular and is downgraded to AngularJS.
- `angularJSComponent` is written in AngularJS and is upgraded to Angular.
- `AngularComponent` is written in Angular and is downgraded to AngularJS.

Let's start with a simple case where the components do not pass any data and do not re-project any nodes.

```
1  const m = angular.module('AngularJsModule', []);
2
3  // The root component of the application, downgraded to AngularJS.
4  @Component({
5    selector: 'app-root',
6    template: `
7      AppComponent written in Angular and downgraded to AngularJS'
8      <angularjs-component></angularjs-component>
9    `
10 })
11 export class AppComponent {}
12 m.directive('appRoot', downgradeComponent({component: AppComponent}));
13
14
15 // An AngularJS component upgraded to Angular.
```

```
// Note that this is @Directive and not a @Component.
@Directive({selector: 'angularjs-component'})
export class AngularJSComponent extends UpgradeComponent {
 constructor(ref: ElementRef, inj: Injector) {
   super('angularjsComponent', ref, inj);
 }
}
m.component('angularjsComponent', {
  template: `
    angularjsComponent written in AngularJS and upgraded to Angular
    <angular-component></angular-component>
  `
});

// An Angular component downgraded to AngularJS
@Component({
  selector: 'angular-component',
  template: `
   AngularComponent written in Angular and downgraded to AngularJS
  `
})
export class AngularComponent {
}
m.directive('angularComponent', downgradeComponent({component: AngularComponent}\
));


@NgModule({
  declarations: [
    AppComponent,
    AngularJSComponent,
    AngularComponent
  ],
  // All downgraded components have to be listed here.
  entryComponents: [
    AppComponent,
    AngularComponent
  ],
  imports: [
    BrowserModule,
    UpgradeModule
  ]
```

```
58 })
59 export class AppModule {
60  constructor(private upgrade: UpgradeModule) {}
61
62  ngDoBootstrap() {
63    this.upgrade.bootstrap(document.body, ['AngularJsModule']);
64  }
65 }
```

Let's look at this example in detail.

First, to use an Angular component in an AngularJS context we need register it using the `downgradeComponent` function, like this:

```
1 m.directive('appRoot', downgradeComponent({component: AppComponent}));
```

This creates an AngularJS directive with the `appRoot` selector. This directive will use `AppComponent` to render its template. Because of this indirection, we need to register `AppComponent` as an entry component.

So the `<app-root>` element itself is owned by AngularJS, which means that we can apply other AngularJS directives to it. Its template, however, is rendered using Angular.

The `downgradeComponent` function sets everything up in such a way that the AngularJS bindings will be hooked up with the inputs and outputs of AppComponent.

Extending `UpgradeComponent` allows to upgrade an AngularJS component.

```
1 @Directive({selector: 'angularjs-component'})
2 export class AngularJSComponent extends UpgradeComponent {
3   constructor(ref: ElementRef, inj: Injector) {
4     // The first argument passed is the AngularJS component we want to upgrade.
5     super('angularjsComponent', ref, inj);
6   }
7 }
```

To ensure the necessary guarantees, NgUpgrade allows only for certain directives to be upgraded. So if we have a directive defining `compile`, `terminal`, `replace`, or `link.post`, we will have to wrap it into an AngularJS component.

Inputs and Outputs

Now, let's make components interact via inputs and outputs.

```
const m = angular.module('AngularJsModule', []);

@Component({
  selector: 'app-root',
  template: `
    AppComponent written in Angular and downgraded to AngularJS:
    counter {{counter}}
    <angularjs-component [counterTimes2]="counter * 2" (multiply)="multiplyCount\
er($event)">
    </angularjs-component>
  `
})
export class AppComponent {
  counter = 1;

  multiplyCounter(n: number): void {
    this.counter *= n;
  }
}
m.directive('appRoot', downgradeComponent({component: AppComponent}));


@Directive({selector: 'angularjs-component'})
export class AngularJSComponent extends UpgradeComponent {
  @Input() counterTimes2: number;
  @Output() multiply: EventEmitter<number>;

  constructor(ref: ElementRef, inj: Injector) {
    super('angularjsComponent', ref, inj);
  }
}
m.component('angularjsComponent', {
  bindings: {
    counterTimes2: '<',
    multiply: '&'
  },
  template: `
    angularjsComponent written in AngularJS and upgraded to Angular
    counterTimes2: {{$ctrl.counterTimes2}}
    <button ng-click="$ctrl.multiply(2)">Double</button>
    <angular-component [counter-times-4]="$ctrl.counterTimes2 * 2" (multiply)="$\
ctrl.multiply($event)">
```

```
    </angular-component>
  `
});

@Component({
  selector: 'angular-component',
  template: `
    AngularComponent written in Angular and downgraded to AngularJS:
    counterTimes4: {{counterTimes4}}
    <button (click)="multiply.next(3)">Triple</button>
  `
})
export class AngularComponent {
  @Input() counterTimes4: number;
  @Output() multiply = new EventEmitter();
}
m.directive('angularComponent', downgradeComponent({ component: AngularComponent\
 }));
```

Let's go through the changes.

Adding inputs and outputs to a downgraded component does not require any extra configuration– we just need to add the properties themselves.

```
export class AngularComponent {
  @Input() counterTimes4: number;
  @Output() multiply = new EventEmitter();
}
```

And we can bind to them in the AngularJS context.

```
<angular-component [counter-times-4]="$ctrl.counterTimes2 * 2" (multiply)="$ctrl\
.multiply($event)">
</angular-component>
```

Note that as with Angular templates, we use square brackets and parenthesis. But in opposite to Angular, we have to hyphenize property names.

When upgrading components we have to to list the inputs and outputs in two places.

First in the Angular directive extending `UpgradeComponent`.

```
@Directive({selector: 'angularjs-component'})
export class AngularJSComponent extends UpgradeComponent {
  @Input() counterTimes2: number;
  @Output() multiply: EventEmitter<number>; // Do not create an instance of Even\
tEmitter here

  constructor(ref: ElementRef, inj: Injector) {
    super('angularjsComponent', ref, inj);
  }
}
```

And then in the AngularJS component itself.

```
m.component('angularjsComponent', {
  bindings: {
    counterTimes2: '<', // < corresponds to @Input
    multiply: '&' // & corresponds to @Output
  },
  template: `
    ...
  `
});
```

Two-Way Bindings

AngularJS and Angular implement two-way binding behavior differently. AngularJS has a special two-way binding capability, whereas Angular simply uses an Input/Output pair. NgUpgrade bridges the two.

```
@Component({
  selector: 'app-root',
  template: `
    AppComponent written in Angular and downgraded to AngularJS:
    counter {{counter}}
    <angularjs-component [(twoWay)]="counter">
    </angularjs-component>
  `
})
export class AppComponent {
}
m.directive('appRoot', downgradeComponent({component: AppComponent}));
```

```
13
14
15 @Directive({selector: 'angularjs-component'})
16 export class AngularJSComponent extends UpgradeComponent {
17
18   // We need to declare these two properties.
19   // [(twoWay)]="counter" is the same as
20   // [twoWay]="counter" (twoWayChange)="counter=$event"
21   @Input() twoWay: number;
22   @Output() twoWayChange: EventEmitter<number>;
23
24   constructor(ref: ElementRef, inj: Injector) {
25     super('angularjsComponent', ref, inj);
26   }
27 }
28 m.component('angularjsComponent', {
29   bindings: {
30     twoWay: '='
31   },
32   template: `
33     angularjsComponent written in AngularJS and upgraded to Angular
34     Bound via a two-way binding: <input ng-model="$ctrl.twoWay">
35   `
36 });
```

Bindings and Change Detection

Change detection works differently in AngularJS and Angular. In AngularJS `$scope.apply` triggers a change detection run, also known as a digest cycle. In Angular, we no longer have `$scope.apply`. Instead, the framework relies on Zone.js, which will trigger a change detection run on every browser event.

Since a hybrid application is an Angular application, it uses Zone.js, and we don't need to worry about `$scope.apply`.

Angular also provides strict guarantees to make the order of checks predictable. A hybrid application preserves these guarantees.

Transclusion/Reprojection

Both AngularJS and Angular provide ways to project nodes from the content DOM into the view DOM. In AngularJS it is called transclusion. In Angular is is called reprojection.

```
@Component({
  selector: 'app-root',
  template: `
    AppComponent written in Angular and downgraded to AngularJS
    <angularjs-component>
      Projected from parent
    </angularjs-component>
  `
})
export class AppComponent {}
m.directive('appRoot', downgradeComponent({component: AppComponent}));


@Directive({selector: 'angularjs-component'})
export class AngularJSComponent extends UpgradeComponent {
  constructor(ref: ElementRef, inj: Injector) {
    super('angularjsComponent', ref, inj);
  }
}
m.component('angularjsComponent', {
  template: `
    angularjsComponent written in AngularJS and upgraded to Angular
    <ng-transclude></ng-transclude>
    <angular-component>
      Projected from parent
    </angular-component>
  `
});


@Component({
  selector: 'angular-component',
  template: `
    AngularComponent written in Angular and downgraded to AngularJS:
    <ng-content></ng-content>
  `
})
export class AngularComponent {
}
m.directive('angularComponent', downgradeComponent({ component: AngularComponent\
 }));
```

For simple cases like this, everything works the way you would expect. You use `<ng-transclude></ng-`

`transclude>` in AngularJS and `<ng-content></ng-content>` in Angular. Multi-slot reprojection still has some issue, which hopefully will be ironed out soon.

Code Samples

- The code samples illustrating the bridging of the two dependency injection systems.[2]
- The code samples illustrating the component integration.[3]

Summary

NgUpgrade is a library we can use to mix and match AngularJS and Angular components and bridge the AngularJS and Angular dependency injection systems. In this chapter we looked at how it works and how we can use it.

[2] https://github.com/vsavkin/upgrade-book-examples/tree/master/ngupgrade/mixing-di
[3] https://github.com/vsavkin/upgrade-book-examples/tree/master/ngupgrade/mixing-components

Chapter 3: Upgrade Shell Strategy

In this chapter we will look at the strategy used in most upgrade projects called “Upgrade Shell”.

When applying the Upgrade Shell strategy, we take an AngularJS application, and replace its root component with a new Angular component. If the application does not have a root component, we introduce one.

Implementation

Let’s look at the simplest way to implement this strategy.

We start with our existing AngularJS application defined and bootstrapped as follows:

```
angular.module('AngularJSAppModule', [deps]).component(...).service(...);
angular.bootstrap(document, ['AngularJSAppModule']);
```

First, we remove the bootstrap call.

```
angular.module('AngularJSAppModule', [deps]).component(...).service(...);
// angular.bootstrap(document, ['AngularJSAppModule']); - No longer needed
```

Next, we define an Angular module importing `UpgradeModule`.

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

Then we define a root component rendering a single element with the `ng-view` class applied.

```
@Component({
 selector: 'app-component',
 template: `<div class="ng-view"></div>`,
})
class AppComponent implements OnInit {
 constructor(private upgrade: UpgradeModule) { }

 ngOnInit() {
   this.upgrade.bootstrap(document, ['AngularJsAppModule']);
 }
}
```

We use the injected `UpgradeModule` to bootstrap the existing AngularJS application in `ngOnInit`. For the upgrade to work property, `upgrade.bootstrap` has to be called inside the Angular zone, and doing it in `ngOnInit` achieves that.

With this setup in place, the order of events during bootstrap will look as follows:

- Angular application bootstraps.
- AppComponent gets created.
- The bootstrap gets called during the Angular change detection cycle, and hence in the Angular zone.
- AngularJS application bootstraps.
- AngularJS router kicks in and inserts its view into the `ng-view`.

Summary

This strategy is a good first step of upgrading an app. It takes five minutes to implement. And what we are getting at the end is technically a new Angular application, even though the meat of the app is still written in AngularJS.

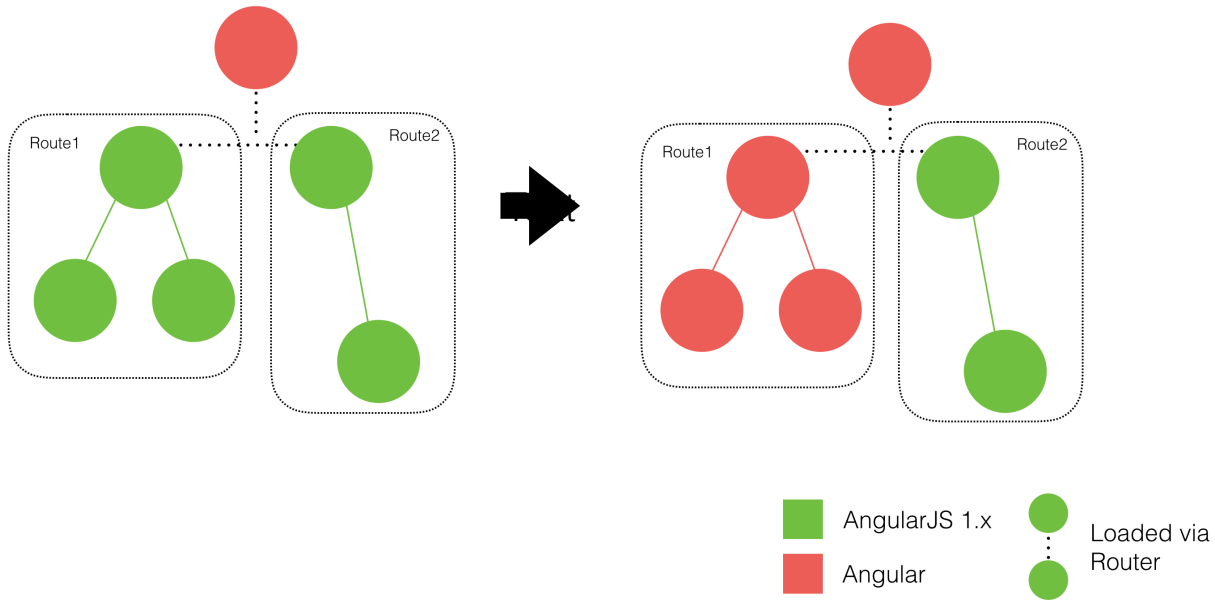
Chapter 4: Two Approaches to Upgrading Angular Apps

In this chapter we will look at two main ways to approach upgrade: Vertical Slicing and Horizontal Slicing.

Vertical Slicing

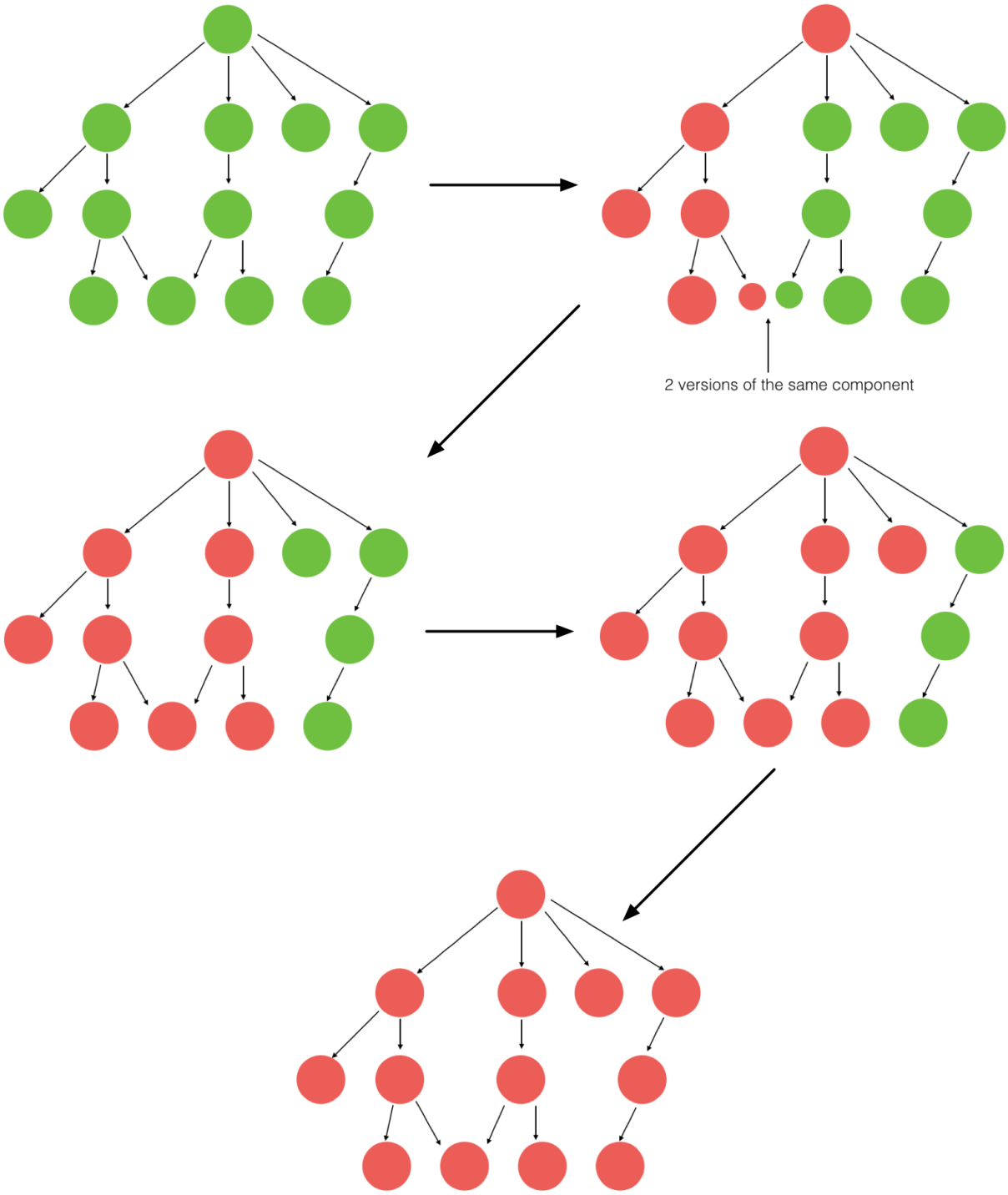
Once we have wrapped our AngularJS app into a shell (see here), we can start upgrading the rest of the application. “Vertical Slicing” is one approach we can use.

Even though, it is not feasible to rewrite the whole application at once, it is often possible to do it route by route, screen by screen, or feature by feature. In which case the whole route is either written in AngularJS or in Angular.



In other words, everything we see on the screen is either written in AngularJS or in Angular.

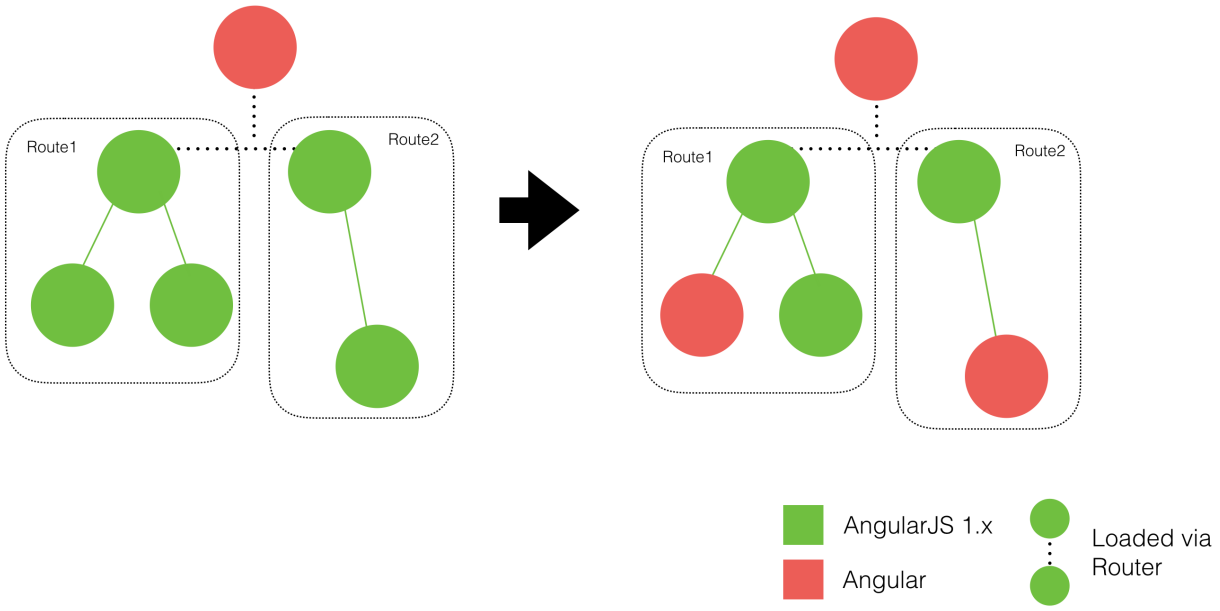
We can visualize this strategy as follows:



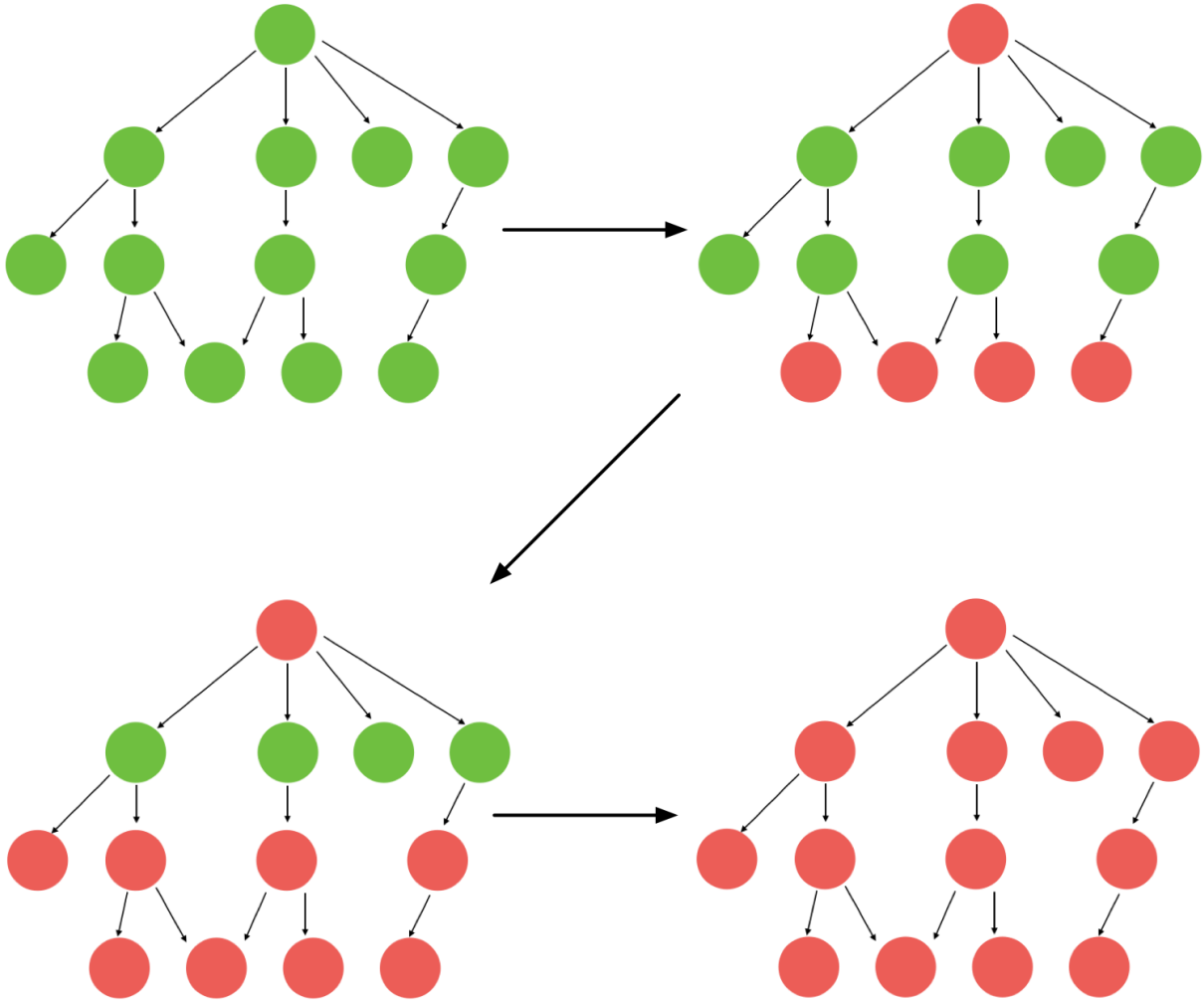
As you can see, while upgrading the second route, we had to duplicate one of the components. For a while we had two copies of that component: one written in AngularJS and the other in Angular. Such duplication is one of this approach's downside.

Horizontal Slicing

Horizontal slicing is the opposite strategy.



We start with upgrading reusable components like inputs and date pickers. Then we upgrade the components using those. And we keep doing it until we work our way up to the root.



One major implication of this approach is that is that no matter what screen we open, we will have two frameworks acting at the same time.

Comparing Horizontal and Vertical Slicing

The major upside of the Vertical Slicing strategy is that we deal with one framework at a time. This means that our application is easier to debug and easier to understand.

Second, when using Vertical Slicing our upgrade process is encapsulated to a single route. This can be extremely important in large organizations with multiple teams working on the same project. Being able to do your work independently means that we won't have to coordinate our work with other teams.

Finally, Vertical Slicing allows us to load NgUpgrade and AngularJS lazily, only when we need to render legacy routes. This makes our applications smaller and faster.

On the other hand, the Horizontal Slicing approach has one major advantage: it is finer grained. We can upgrade a component and ship it to production in a day, whereas upgrading a route may take months.

Chapter 5: Managing URL

Most AngularJS applications use some sort of router, so do most Angular applications. This means that upgrading an application always involves coordinating the two routers and managing the URL.

In this chapter we will look at why it is a hard problem, what kind of setups application typically have, and a few strategies that can make solving this problem easier.

Why Managing URL is Hard?

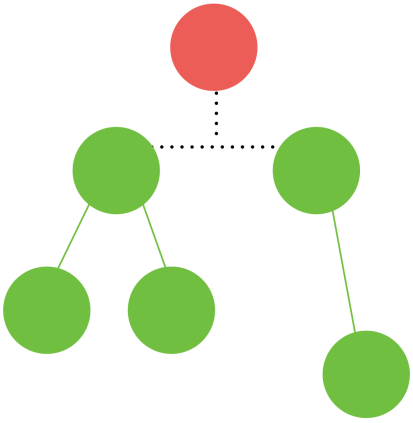
The URL, or to be more specific window.location, is a global and mutable resource. And as such, managing it is tricky. In particular, when multiple frameworks with multiple routers interact with it. Since we often have multiple routers active during the upgrade process, we need to know how to do it right.

Single Ownership and Mixed Ownership

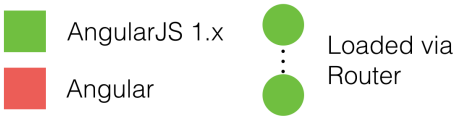
There are two URL-managing setups we can use during upgrade: single ownership and mixed ownership.

Single Ownership

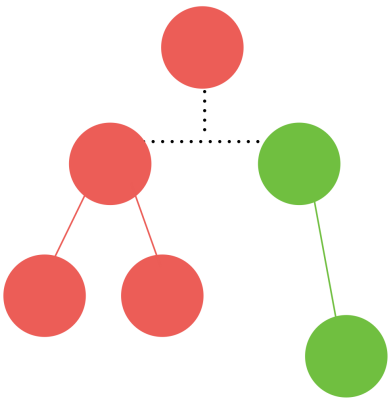
Say we have an application with the following four routes:



http://localhost:3000/
http://localhost:3000/feature1/sub1
http://localhost:3000/feature1/sub2
http://localhost:3000/feature2



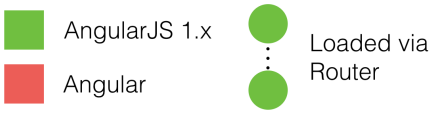
Now, say we used the "Vertical Slicing" strategy (see here) to upgrade a few of those routes to Angular.



http://localhost:3000/
http://localhost:3000/feature1/sub1
http://localhost:3000/feature1/sub2
http://localhost:3000/feature2

managed by Angular Router

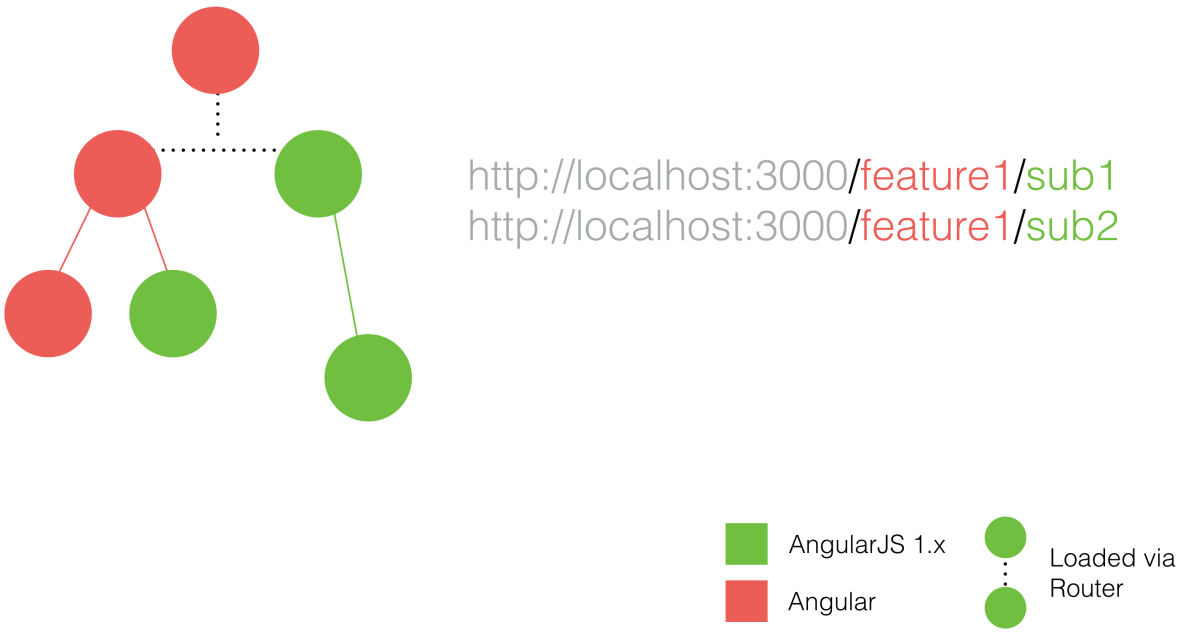
managed by AngularJS router



In the single-ownership setup, the features upgraded to Angular are managed by the Angular router. And the ones still in AngularJS are managed by the AngularJS router (or the UI router). In other words, every route has a single owner: either the AngularJS Router or the new Angular Router.

Mixed Ownership

In the mixed-ownership setup one URL can be managed by both the AngularJS Router and the Angular Router. One section is managed by AngularJS and the other one by Angular.



It often happens when we want to show a dialog written in AngularJS, whereas the rest of our application has already been upgraded.

Single Ownership or Mixed Ownership?

If possible, we should use the single ownership setup. With it we will always have clear transitions between the upgraded and legacy parts of our application. Only one router can update the URL at a time, so there will be no concurrency-related issues.

Now we've learned about the two main strategies, let's see how we can implement them.

Sibling Outlets

Sibling Outlets is one of the most useful strategies we can employ during the upgrade of an application using multiple routers. There are many ways to implement it. In this chapter we will look at the simplest one, which hopefully will give you an idea on how to do it for your application.

We start by making our top-level component as simple as possible: it has a router-outlet Angular directive and an ng-view AngularJS directive. In other words, we have two sibling router outlets: one for Angular and the other one for AngularJS.

```
@Component({
  selector: 'app-component',
  template: `
    <router-outlet></router-outlet>
    <div class="ng-view"></div>
  `
})
class AppComponent { }
```

When using single-ownership setup, only one outlet is active at a time–the other one is empty. In the mixed-ownership setup, both of them can be active simultaneously.

Configuring Angular

Next, we define the routes for the upgraded feature in the Angular router configuration.

```
@NgModule({
  imports: [
    BrowserModule,
    UpgradeModule,
    RouterModule.forRoot([
      { path: '', pathMatch: 'full', component: HomeComponent },
      { path: 'feature1/sub1', component: Feature1Sub1Component },
      { path: 'feature1/sub2', component: Feature1Sub2Component }
    ])
  ],
  bootstrap: [AppComponent],
  declarations: [AppComponent, HomeComponent, Feature1Sub1Component, Feature1Sub\
2Component]
})
class AppModule {}
```

Finally, we need to tell the router to only handle URLs starting with ‘feature1’, so it doesnâ€™t error when it sees “legacy” URLs. There are two main ways to do it: by overriding a UrlHandlingStrategy and by using an empty-path “sink” route.

Overriding UrlHandlingStrategy

We can provide a custom URL handling strategy to tell the Angular router which URLs it should process. When it sees a URL that does not match the criteria, it will unload all the components emptying the root router outlet.

```
class CustomHandlingStrategy implements UrlHandlingStrategy {
  shouldProcessUrl(url) { return url.toString().startsWith("/feature1") || url.t\
oString() === "/"; }
  extract(url) { return url; }
  merge(url, whole) { return url; }
}

@NgModule({
  imports: [
    BrowserModule,
    UpgradeModule,
    RouterModule.forRoot([
      { path: '', pathMatch: 'full', component: HomeComponent },
      { path: 'feature1/sub1', component: Feature1Sub1Component },
      { path: 'feature1/sub2', component: Feature1Sub2Component }
    ])
  ],
  providers: [
    { provide: UrlHandlingStrategy, useClass: CustomHandlingStrategy }
  ],
  bootstrap: [AppComponent],
  declarations: [AppComponent, HomeComponent, Feature1Sub1Component, Feature1Sub\
2Component]
})
class AppModule {}
```

Implementing a Sink Route

The Angular router processes its routes in order. So if we put an empty-path route at the end of the list, it will match any URL that is not matched by upgraded routes. And if we make it render an empty component, we will achieve a similar result to the one above.

```
@Component({selector: 'empty', template: ''})
class EmptyComponent {}

@NgModule({
  imports: [
    BrowserModule,
    UpgradeModule,
    RouterModule.forRoot([
      { path: '', pathMatch: 'full', component: HomeComponent },
      { path: 'feature1/sub1', component: Feature1Sub1Component },
```

```
11        { path: 'feature1/sub2', component: Feature1Sub2Component },
12        { path: '', component: EmptyComponent }
13      ])
14    ],
15    bootstrap: [AppComponent],
16    declarations: [AppComponent, HomeComponent, Feature1Sub1Component, Feature1Sub\
17  2Component, EmptyComponent]
18  })
19  class AppModule {}
```

We will look at sink routes more in further chapters, when we show how to load the AngularJS part of our application lazily.

This is what our Angular configuration looks like. Now let's see what we need to do to configure the AngularJS part of the app.

Configuring AngularJS

We still configure the legacy routes using the $routeProvider.

```
1  angular.config(($routeProvider) => {
2    $routeProvider
3      .when('/feature2', {template : '<feature2></feature2>'})
4      .otherwise({template : ''});
5  });
```

As you can see the setup above uses a sink route, but we can handle upgraded routes in a way that is similar to providing a custom URL handling strategy. For instance, if we use the UI router, we can subscribe to '$stateChangeStart' and call 'preventDefault' when navigating to the upgraded part of the application.

Summary

Most AngularJS and Application applications use some sort of router, this means that upgrading an application always involves coordinating the two routers and managing the URL. This can be tricky because the URL is a global and mutable resource.

In this chapter we looked at the two URL-managing setups we can use during upgrade: single ownership and mixed ownership. In the single-ownership setup, every route has a single owner: either the AngularJS Router or the new Angular Router. In the mixed-ownership setup, the same route can be handled by both. We talked about which one we should use and when.

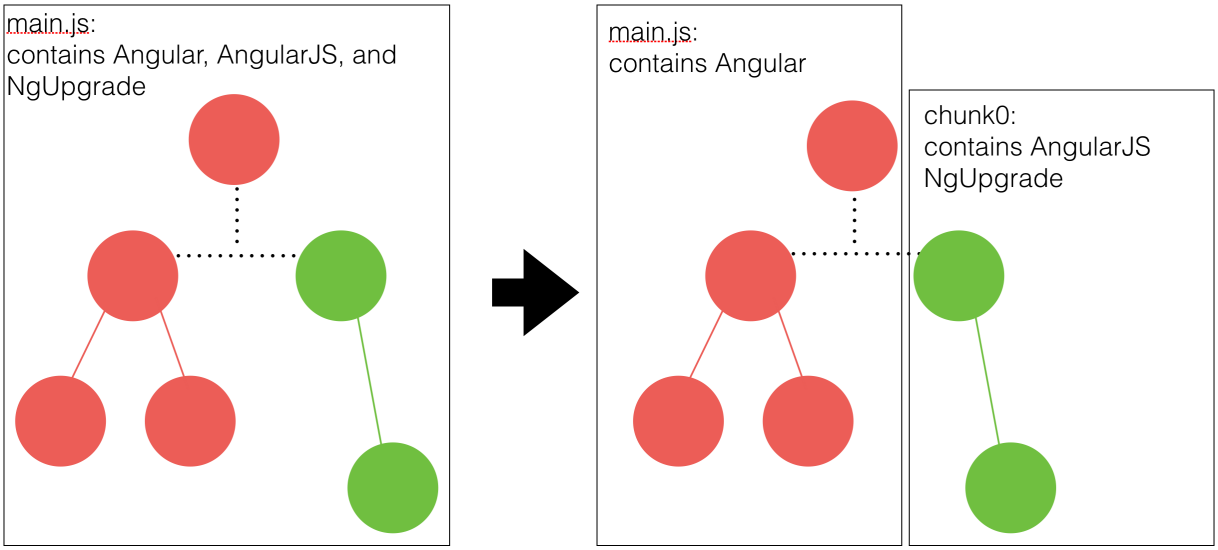
Then we looked at the Sibling Outlets strategy we often use to upgrade applications using multiple routers.

Chapter 6: Lazy Loading AngularJS Applications

One of the best things about the Angular router is its excellent support for lazy loading. We can use it to make the initial bundle of our application as small as possible by only shipping to the client what is needed to render the application's first screen.

This book is about running our applications in the hybrid mode, where we use both AngularJS and Angular. One implication of this is that we have ship both of the frameworks to the client. We, however, don't have to do it in the initial bundle.

In this chapter I will show how to set up a hybrid app such that AngularJS, NgUpgrade, and the existing AngularJS code are loaded lazily.



Dual Router Setup

Suppose our application has the following four routes, where the Angular application handles `/angular_a` and `/angular_b`, and the existing AngularJS application (via the UI-router) handles `/angularjs_a` and `/angularjs_b`.

```
/angular_a
/angular_b
/angularjs_a
/angularjs_b
```

Let's also suppose that our application's entry point is `/angular_a`.

Our goal is to treat this application as a regular Angular application and not worry about AngularJS, NgUpgrade or the UI-router until the user navigates to `/angularjs_a` or `/angularjs_b`. Only at that point we want to load the needed AngularJS code. Let's see how we can do it.

Sibling Outlets

```
@Component({
 selector: 'app-root',
 template: `
   <router-outlet></router-outlet>
   <div ui-view></div>
 `
})
export class AppComponent {}

@NgModule({
 declarations: [
   AppComponent
 ],
 imports: [
   BrowserModule,
   RouterModule.forRoot([
     //...
   ])
 ],
 providers: [],
 bootstrap: [AppComponent]
})
export class AppModule { }
```

As you can see the root component implements the Sibling Outlets strategy. It has a `<router-outlet>` Angular directive and an `<div ui-view>` AngularJS directive. In other words, it two sibling router outlets: one for Angular and the other one for AngularJS. Note that `<div ui-view>` will remain a simple DOM element until AngularJS is loaded–only then the ui-view directive will be place there.

AngularJS Application

To illustrate this example, let's sketch out a simple AngularJS application using the UI-router.

```
export const module = angular.module('AngularJSApp', ['ui.router']);

module.config(($locationProvider, $stateProvider) => {
  // use history api instead of URL fragment
  $locationProvider.html5Mode(true);

  $stateProvider.state('angularjs_a', {
    url: '/angularjs_a',
    template: `
      AngularJS A!

      <a href="/angular_a">Go to Angular A</a>
      <a href="/angular_b">Go to Angular B</a>
      Go to AngularJS A
      <a href="/angularjs_b">Go to Angular JS B</a>
    `
 });

 $stateProvider.state('angularjs_b', {
    url: '/angularjs_b',
    template: `
      AngularJS B!

      <a href="/angular_a">Go to Angular A</a>
      <a href="/angular_b">Go to Angular B</a>
      <a href="/angularjs_a">Go to Angular JS A</a>
      Go to AngularJS B
    `
  });
});
```

To make this AngularJS application ready for the hybrid app, let's add a sink state to capture unmatched URLs, which, we assume, will be handled by the Angular application.

```
export const module = angular.module('AngularJSApp', ['ui.router']);

module.config(($locationProvider, $stateProvider) => {
  //...

  $stateProvider.state('sink', {
    url: '/*path',
    template: ''
  });
});
```

Angular Application

Next, let's sketch out an Angular application.

```
@Component({
  template: `
    Angular A!

    Go to Angular A
    <a routerLink="/angular_b">Go to Angular B</a>
    <a routerLink="/angularjs_a">Go to AngularJS A</a>
    <a routerLink="/angularjs_b">Go to AngularJS B</a>
  `
})
export class AngularAComponent {}

@Component({
  template: `
    Angular B!

    <a routerLink="/angular_a">Go to Angular A</a>
    Go to Angular B
    <a routerLink="/angularjs_a">Go to AngularJS A</a>
    <a routerLink="/angularjs_b">Go to AngularJS B</a>
  `
})
export class AngularBComponent {}

@NgModule({
  declarations: [
```

```
    AppComponent,
    AngularAComponent,
    AngularBComponent
  ],
  imports: [
    BrowserModule,
    RouterModule.forRoot([
      {path: '', redirectTo: 'angular_a', pathMatch: 'full'},
      {path: 'angular_a', component: AngularAComponent},
      {path: 'angular_b', component: AngularBComponent}
    ])
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

And similar to the AngularJS application, let's define a sink route to capture unmatched URLs.

```
@NgModule({
  declarations: [
    AppComponent,
    AngularAComponent,
    AngularBComponent
  ],
  imports: [
    BrowserModule,
    RouterModule.forRoot([
      {path: '', redirectTo: 'angular_a', pathMatch: 'full'},
      {path: 'angular_a', component: AngularAComponent},
      {path: 'angular_b', component: AngularBComponent},
      {path: '', loadChildren: './angularjs.module#AngularJSModule'}
    ])
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

The Angular application assumes that unmatched URLs are handled by AngularJS. For this to work we need to load AngularJS first, and that's what the `{path: '', loadChildren: './angularjs.module#AngularJSModule'}` route does.

Loading AngularJS

`AngularJSModule` is a thin Angular wrapper around our existing AngularJS application.

```
import {Component, NgModule} from '@angular/core';
import {RouterModule} from '@angular/router';

import {module} from './angularjsapp';
import {UpgradeModule} from '@angular/upgrade/static';
import {setUpLocationSync} from '@angular/router/upgrade';

@Component({template: ``})
export class EmptyComponent {}

@NgModule({
  declarations: [
    EmptyComponent
  ],
  imports: [
    UpgradeModule,
    RouterModule.forChild([
      {path: '**', component: EmptyComponent}
    ])
  ]
})
export class AngularJSModule {
  // The constructor is called only once, so we bootstrap the application
  // only once, when we first navigate to the legacy part of the app.
  constructor(upgrade: UpgradeModule) {
    upgrade.bootstrap(document.body, [module.name]);
    setUpLocationSync(upgrade);
  }
}
```

To make the Angular router happy, we render an empty component. More importantly, we bootstrap the AngularJS application and set up the location synchronization.

Overview

Now when we have looked at all the pieces separately, let's see how they work together.

When the user loads the page, the Angular application will get bootstrapped: The Angular router will redirect to `'/angular_a'`, will instantiate `AngularAComponent` and will place it into `<router-outlet>` Defined in AppComponet. Note we haven't loaded AngularJS, NgUpgrade, or the existing AngularJS application yet, and navigating from `'/angular_a'` to `'/angular_b'` and back does not change that.

When the user navigates to `'/angularjs_a'`, the Angular router will match the `{path: '', loadChildren: './angularjs.module#AngularJSModule'}` route, which will load `AngularJSModule`. The bundle chunk containing the module will also contain `AngularJS`, `NgUpgrade`, and the existing AngularJS application. Once loaded, the module will call `upgrade.bootstrap`. This will trigger the UI-router, which will match the `angularjs_a` state and will place it into `<div ui-view>`. At the same time the Angular router will place an `EmptyComponent` into `<router-outlet>`. When the user navigates from `'/angularjs_a'` to `'/angularjs_b'` and back, the Angular router will keep matching the sink route, and the UI-router will update the `<div ui-view>`.

When the user navigates from `'/angularjs_a'` to `'/angular_a'`, the UI-router will match its sink route and will place an empty template into `<div ui-view>`. The `setUpLocationSync` helper will notify the Angular router about the URL change. The Angular router will match the URL and will place `AngularAComponent` into `<router-outlet>`. Note, the AngularJS application keeps running–it does not unloaded (it is almost impossible to unload a real-world `AngularJS` application, so we have to keep it in memory). When the user navigates from `'/angular_a'` to `'/angular_b'` and back, the UI-router will keep matching its sink route, and the Angular router will update `<router-outlet>`.

Finally, when the user goes back to `'/angularjs_a'`, the Angular router will match its sink route. And the UI-router, which is already running, will match the appropriate state. This time we haven't had to load or bootstrap the AngularJS application because it is only done once.

Preloading

As it stand right now, we will load `AngularJS` when and only when the user navigates to a route handled by `AngularJS`. This makes the initial load fast, but makes the user's first navigation to `'/angularjs_a'` slow. We can fix it by enabling preloading.

```
1  import {PreloadAllModules, RouterModule} from '@angular/router';
2
3  @NgModule({
4    declarations: [
5      AppComponent,
6      AngularAComponent,
7      AngularBComponent
8    ],
9    imports: [
10     BrowserModule,
```

```
    RouterModule.forRoot([
      {path: '', redirectTo: 'angular_a', pathMatch: 'full'},
      {path: 'angular_a', component: AngularAComponent},
      {path: 'angular_b', component: AngularBComponent},
      {path: '', loadChildren: './angularjs.module#AngularJSModule'}
    ], {
      enableTracing: true,
      preloadingStrategy: PreloadAllModules // ADD THIS!
    })
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

With this in place, the router will preload `AngularJSModule` (and, as a result, `AngularJS`, `NgUpgrade`, and the existing AngularJS app) in the background while the user is interacting with our application. You can learn more about this in the Angular Router: Preloading Modules[4] article.

More interestingly, the router will instantiate `AngularJSModule`, which in turn will bootstrap the AngularJS application. This means that the application will be instantiated in the background as well.

And that's how we get the best of both worlds: our initial bundle is small and subsequent navigations are instant.

Code Samples

- The code samples showing how to load AngularJS lazily.[5]

[4] https://vsavkin.com/angular-router-preloading-modules-ba3c75e424cb

[5] https://github.com/vsavkin/upgrade-book-examples/tree/master/lazyloading