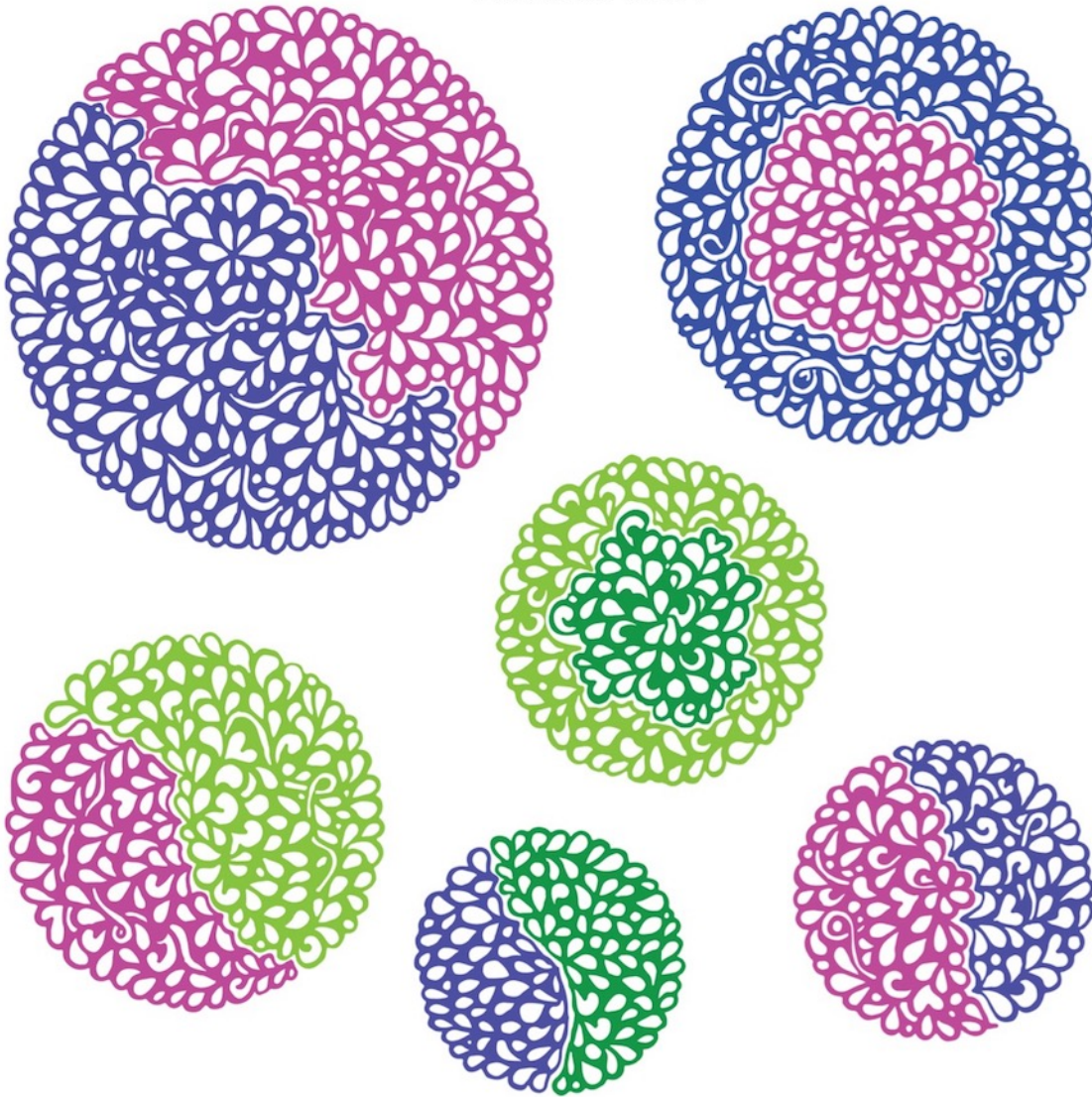


UPGRADING ANGULAR APPLICATIONS

VICTOR SAVKIN



Angular consulting for enterprise customers,
from core contributors

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.

© 2017 Victor Savkin and Nrwl.io

Contents

Introduction	1
Chapter 1: Preparation	3
Angular Style	3
Writing AngularJS Applications in the Angular Style	3
New Development in the Angular Style	5
Changing Existing Code to the Angular Style	5
Chapter 2: NgUpgrade in Depth	7
How to Use it Best	7
How it Works	7
Bootstrapping	9
Dependency Injection	12
Component Integration	14
Code Samples	23
Summary	23
Chapter 3: Upgrade Shell Strategy	24
Implementation	24
Summary	25
Chapter 4: Two Approaches to Upgrading Angular Apps	26
Vertical Slicing	26
Horizontal Slicing	29
Comparing Horizontal and Vertical Slicing	31

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.io¹ 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:

```
1 angular.component('myComponent', {
2   bindings: {
3     myInput: '<',
4     myOutput: '&'
5   },
6   template: `
7     <h2>{{$ctrl.input}}!</h2>
8     <button ng-click="$ctrl.onEvent()">Trigger Action!</button>
9   `,
10  controller: function() {
11    this.onEvent = () => {
12      this.myOutput('some data');
13    };
14  }
15 });
```

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.

```
1 class MyComponent {
2   myInput: string;
3   myOutput: (s: string) => void;
4
5   onEvent() {
6     this.myOutput('some data');
7   }
8 }
9
10 angular.component('myComponent', {
11   bindings: {
12     myInput: '<',
13     myOutput: '&'
14   },
15   template: `
16     <h2>{{$ctrl.input}}!</h2>
17     <button ng-click="$ctrl.onEvent()">Trigger Action!</button>
18   `,
19   controller: MyComponent
20 });
```

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.

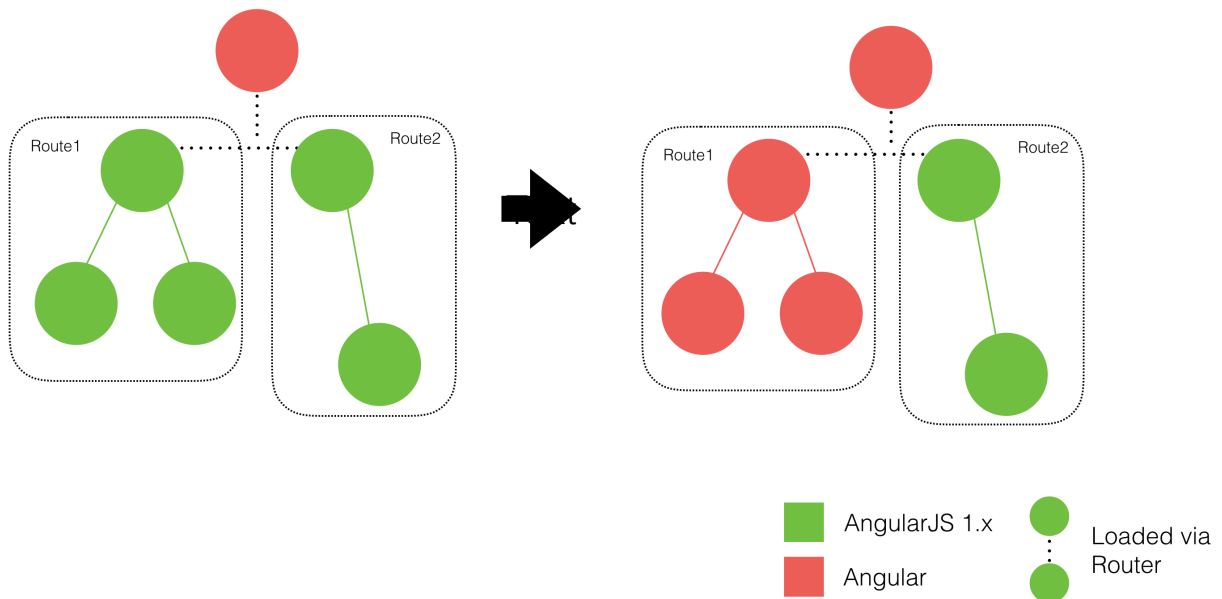
Changing Existing Code to the Angular Style

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.

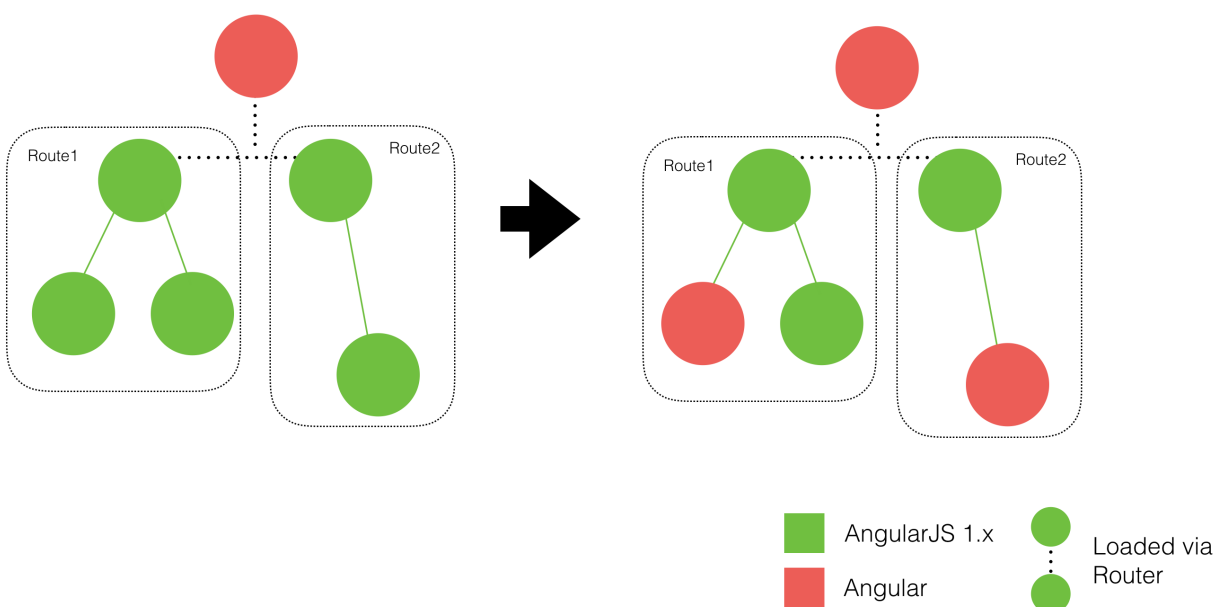
Two Approaches to Upgrade

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

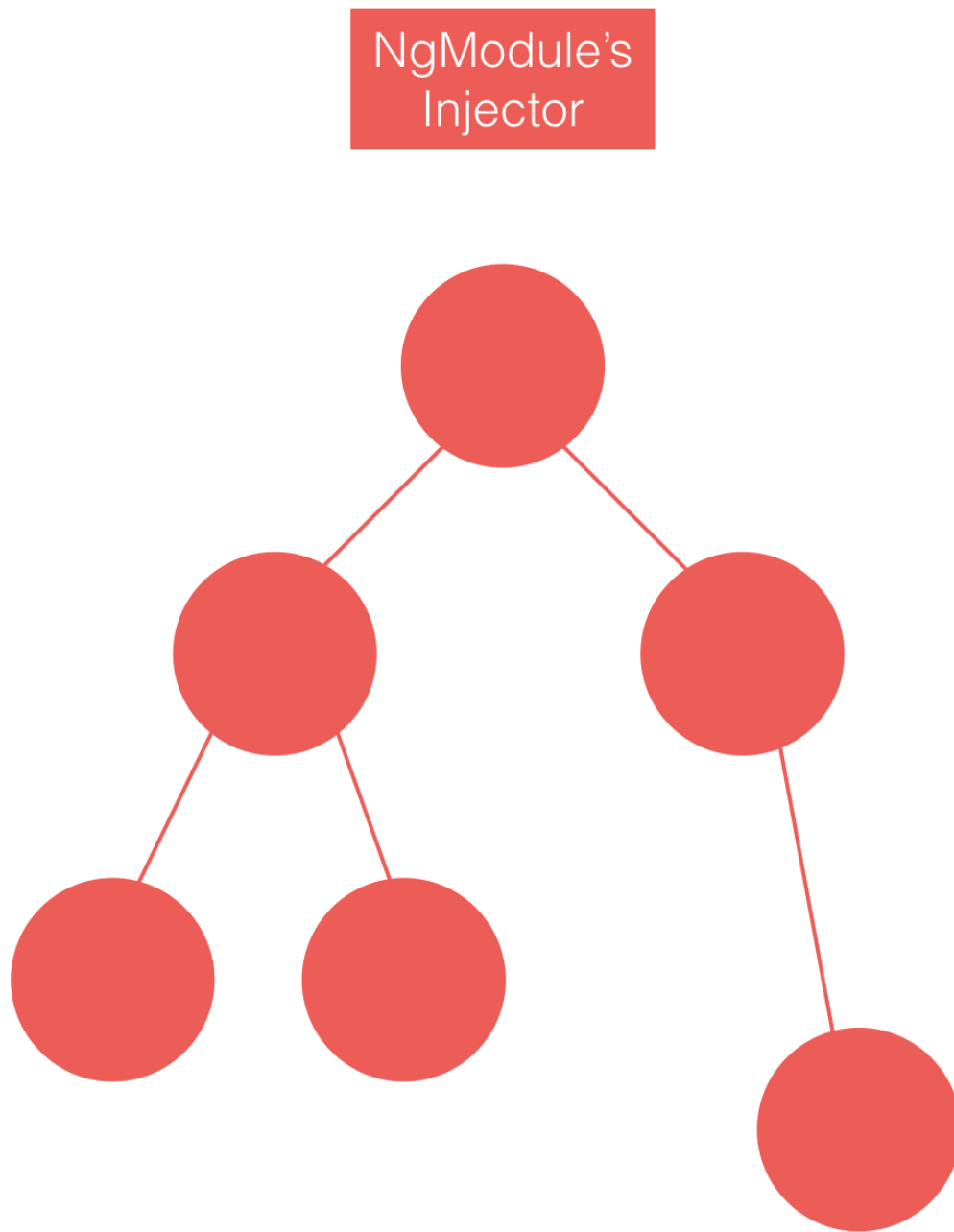
NgUpgrade is a library put together by the Angular team, which we can use in our applications to mix and match AngularJS and Angular components and bridge the AngularJS and Angular dependency injection systems. In this chapter we will look at what it is and how it works.

How to Use it Best

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

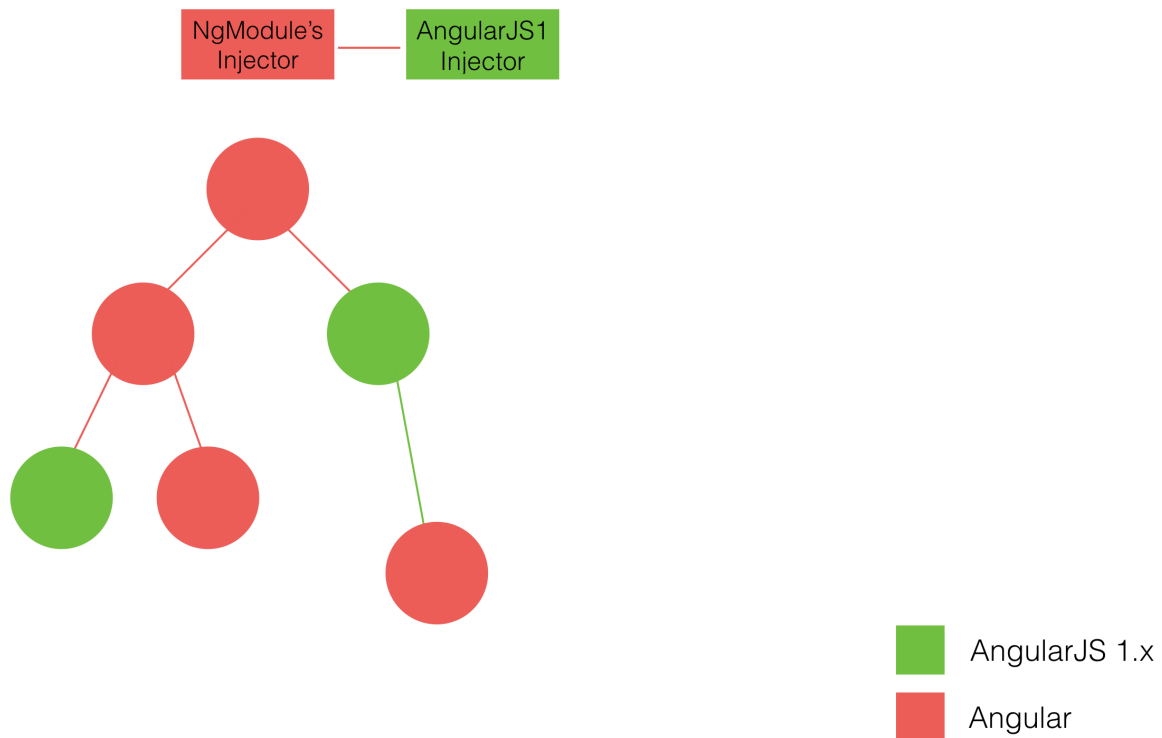
How it Works

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.

```

1  @Component({
2    selector: 'app-root',
3    templateUrl: './app.component.html'
4  })
5  export class AppComponent {}
6  // Access global AngularJS 1.x object
7  const m = angular.module('AngularJsModule', []);
8  m.directive('appRoot', downgradeComponent({component: AppComponent}));
9

```

```
10 @NgModule({
11   declarations: [
12     AppComponent
13   ],
14   imports: [
15     BrowserModule,
16     UpgradeModule
17   ]
18 })
19 export class AppModule {
20   constructor(private upgrade: UpgradeModule) {}
21
22   ngDoBootstrap() {
23     this.upgrade.bootstrap(document.body, ['AngularJsModule']);
24   }
25 }
```

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'`.

```
1 import 'angular';
2 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:

```
1 import { UpgradeModule, setAngularJSGlobal } from '@angular/upgrade/static';
2
3 @NgModule({
4   declarations: [
5     AppComponent
6   ],
7   imports: [
8     BrowserModule,
9     UpgradeModule
10  ]
11 })
12 class AppModule {
13   constructor(private upgrade: UpgradeModule) { }
14
15   ngDoBootstrap() {
16     requirejs(['angular', 'angularJsApp'], (angular, app) => {
17       setAngularJSGlobal(angular);
18       this.upgrade.bootstrap(document, ['AngularJsAppModule']);
19     });
20   }
21 }
```

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:

```
1 resolve: {
2   alias: {
3     "@angular/upgrade/static": "@angular/upgrade/bundles/upgrade-static.umd.js"
4   }
5 }
```

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:

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

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

```
1 function needsAngularJsInjectableFactory($injector) {
2   return `needsAngularJsInjectable got ${$injector.get('angularJsInjectable')}`;
3 }
4
5 @NgModule({
6   imports: [
7     BrowserModule,
8     UpgradeModule
9   ],
10  providers: [
11    {
12      provide: 'needsAngularJsInjectable',
13      useFactory: needsAngularJsInjectableFactory,
14      deps: ['$injector'] // $injector is provided by UpgradeModule
```

```

15     }
16   ]
17 })
18 export class AppModule {
19   constructor(private upgrade: UpgradeModule) {}
20
21   ngDoBootstrap() {
22     this.upgrade.bootstrap(document.body, ['AngularJsModule']);
23     console.log(this.upgrade.injector.get('needsAngularJsInjectable'));
24   }
25 }

```

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.

```

1  @NgModule({
2    imports: [
3      BrowserModule,
4      UpgradeModule
5    ],
6    providers: [
7      {
8        provide: 'needsAngularJsInjectable',
9        useFactory: needsAngularJsInjectableFactory,
10       deps: ['$injector'] // $injector is provided by UpgradeModule
11     }
12   ]
13 })
14 export class AppModule {
15   constructor(private upgrade: UpgradeModule) {}
16
17   ngDoBootstrap() {
18     console.log(this.upgrade.injector.get('needsAngularJsInjectable')); // throws
19     this.upgrade.bootstrap(document.body, ['AngularJsModule']);
20   }
21 }

```

Accessing Angular Injectables in AngularJS

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


```

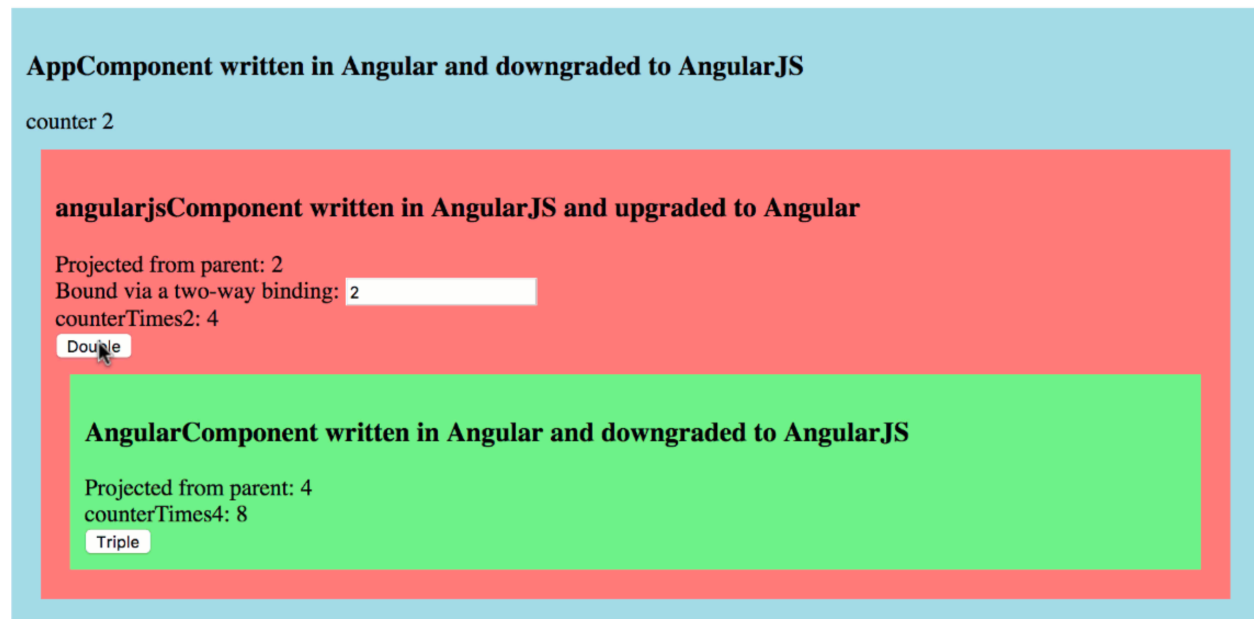
1  import { downgradeInjectable, UpgradeModule } from '@angular/upgrade/static';
2
3  export class AngularInjectable {
4    get value() { return 'angularInjectable-value'; }
5  }
6
7  const m = angular.module('AngularJsModule', []);
8  m.factory('angularInjectable', downgradeInjectable(AngularInjectable));
9  m.factory('needsAngularInjectable', (angularInjectable: AngularInjectable) => `n\
10 eedsAngularInjectable [got ${angularInjectable.value}]`);
11
12 @NgModule({
13   imports: [
14     BrowserModule,
15     UpgradeModule
16   ],
17   providers: [
18     AngularInjectable
19   ]
20 })
21 export class AppModule {
22   constructor(private upgrade: UpgradeModule) {}
23
24   ngDoBootstrap() {
25     this.upgrade.bootstrap(document.body, ['AngularJsModule']);
26     console.log(this.upgrade.$injector.get('needsAngularInjectable')); // 'angula\
27 rInjectable-value'
28   }
29 }

```

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.

```

```
16 // Note that this is @Directive and not a @Component.
17 @Directive({selector: 'angularjs-component'})
18 export class AngularJSComponent extends UpgradeComponent {
19   constructor(ref: ElementRef, inj: Injector) {
20     super('angularjsComponent', ref, inj);
21   }
22 }
23 m.component('angularjsComponent', {
24   template: `
25     angularjsComponent written in AngularJS and upgraded to Angular
26     <angular-component></angular-component>
27   `
28 });
29
30 // An Angular component downgraded to AngularJS
31 @Component({
32   selector: 'angular-component',
33   template: `
34     AngularComponent written in Angular and downgraded to AngularJS
35   `
36 })
37 export class AngularComponent {
38 }
39 m.directive('angularComponent', downgradeComponent({component: AngularComponent})\
40 );
41
42
43 @NgModule({
44   declarations: [
45     AppComponent,
46     AngularJSComponent,
47     AngularComponent
48   ],
49   // All downgraded components have to be listed here.
50   entryComponents: [
51     AppComponent,
52     AngularComponent
53   ],
54   imports: [
55     BrowserModule,
56     UpgradeModule
57   ]
58 })
```

```

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.

```
1  const m = angular.module('AngularJsModule', []);
2
3  @Component({
4    selector: 'app-root',
5    template: `
6      AppComponent written in Angular and downgraded to AngularJS:
7      counter {{counter}}
8      <angularjs-component [counterTimes2]="counter * 2" (multiply)="multiplyCount\
9  er($event)">
10     </angularjs-component>
11   `
12 })
13 export class AppComponent {
14   counter = 1;
15
16   multiplyCounter(n: number): void {
17     this.counter *= n;
18   }
19 }
20 m.directive('appRoot', downgradeComponent({component: AppComponent}));
21
22
23 @Directive({selector: 'angularjs-component'})
24 export class AngularJSComponent extends UpgradeComponent {
25   @Input() counterTimes2: number;
26   @Output() multiply: EventEmitter<number>;
27
28   constructor(ref: ElementRef, inj: Injector) {
29     super('angularjsComponent', ref, inj);
30   }
31 }
32 m.component('angularjsComponent', {
33   bindings: {
34     counterTimes2: '<',
35     multiply: '&'
36   },
37   template: `
38     angularjsComponent written in AngularJS and upgraded to Angular
39     counterTimes2: {{$ctrl.counterTimes2}}
40     <button ng-click="$ctrl.multiply(2)">Double</button>
41     <angular-component [counter-times-4]="$ctrl.counterTimes2 * 2" (multiply)="$\
42     ctrl.multiply($event)">
```

```

43     </angular-component>
44     `
45   });
46
47   @Component({
48     selector: 'angular-component',
49     template: `
50       AngularComponent written in Angular and downgraded to AngularJS:
51       counterTimes4: {{counterTimes4}}
52       <button (click)="multiply.next(3)">Triple</button>
53     `
54   })
55   export class AngularComponent {
56     @Input() counterTimes4: number;
57     @Output() multiply = new EventEmitter();
58   }
59   m.directive('angularComponent', downgradeComponent({ component: AngularComponent\
60     }));

```

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.

```

1  export class AngularComponent {
2    @Input() counterTimes4: number;
3    @Output() multiply = new EventEmitter();
4  }

```

And we can bind to them in the AngularJS context.

```

1  <angular-component [counter-times-4]="$ctrl.counterTimes2 * 2" (multiply)="$ctrl\
2  .multiply($event)">
3  </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 list the inputs and outputs in two places.

First in the Angular directive extending UpgradeComponent.

```

1  @Directive({selector: 'angularjs-component'})
2  export class AngularJSComponent extends UpgradeComponent {
3    @Input() counterTimes2: number;
4    @Output() multiply: EventEmitter<number>; // Do not create an instance of Even\
5    tEmitter here
6
7    constructor(ref: ElementRef, inj: Injector) {
8      super('angularjsComponent', ref, inj);
9    }
10 }

```

And then in the AngularJS component itself.

```

1  m.component('angularjsComponent', {
2    bindings: {
3      counterTimes2: '<', // < corresponds to @Input
4      multiply: '&' // & corresponds to @Output
5    },
6    template: `
7      ...
8    `
9  });

```

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.

```

1  @Component({
2    selector: 'app-root',
3    template: `
4      AppComponent written in Angular and downgraded to AngularJS:
5      counter {{counter}}
6      <angularjs-component [(twoWay)]="counter">
7      </angularjs-component>
8    `
9  })
10 export class AppComponent {
11 }
12 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 it is called reprojection.


```

1  @Component({
2    selector: 'app-root',
3    template: `
4      AppComponent written in Angular and downgraded to AngularJS
5      <angularjs-component>
6        Projected from parent
7      </angularjs-component>
8    `
9  })
10 export class AppComponent {}
11 m.directive('appRoot', downgradeComponent({component: AppComponent}));
12
13
14 @Directive({selector: 'angularjs-component'})
15 export class AngularJSComponent extends UpgradeComponent {
16   constructor(ref: ElementRef, inj: Injector) {
17     super('angularjsComponent', ref, inj);
18   }
19 }
20 m.component('angularjsComponent', {
21   template: `
22     angularjsComponent written in AngularJS and upgraded to Angular
23     <ng-transclude></ng-transclude>
24     <angular-component>
25       Projected from parent
26     </angular-component>
27   `
28 });
29
30
31 @Component({
32   selector: 'angular-component',
33   template: `
34     AngularComponent written in Angular and downgraded to AngularJS:
35     <ng-content></ng-content>
36   `
37 })
38 export class AngularComponent {
39 }
40 m.directive('angularComponent', downgradeComponent({ component: AngularComponent\
41 }));

```

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:

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

First, we remove the bootstrap call.

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

Next, we define an Angular module importing UpgradeModule.

```
1 @NgModule({
2   imports: [
3     BrowserModule,
4     UpgradeModule,
5   ],
6   bootstrap: [AppComponent],
7   declarations: [AppComponent]
8 })
9 class AppModule {}
```

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

```
1 @Component({
2   selector: 'app-component',
3   template: `<div class="ng-view"></div>`,
4 })
5 class AppComponent implements OnInit {
6   constructor(private upgrade: UpgradeModule) { }
7
8   ngOnInit() {
9     this.upgrade.bootstrap(document, ['AngularJsAppModule']);
10  }
11 }
```

We use the injected `UpgradeModule` to bootstrap the existing AngularJS application in `ngOnInit`. For the upgrade to work properly, `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.

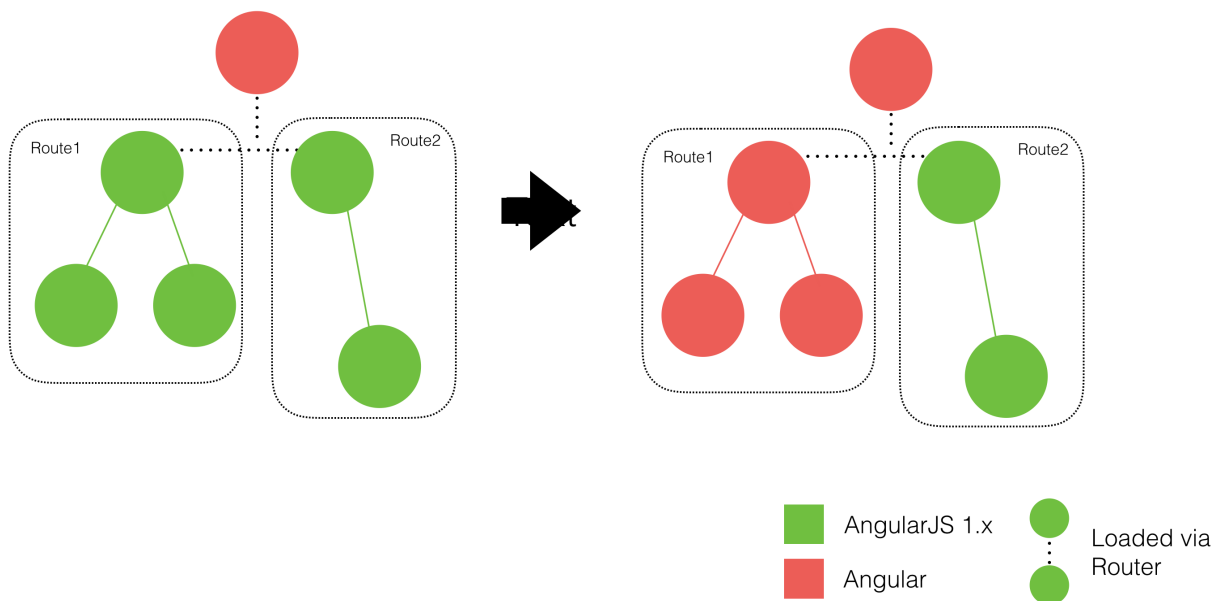
Chapter 4: Two Approaches to Upgrading Angular Apps

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.

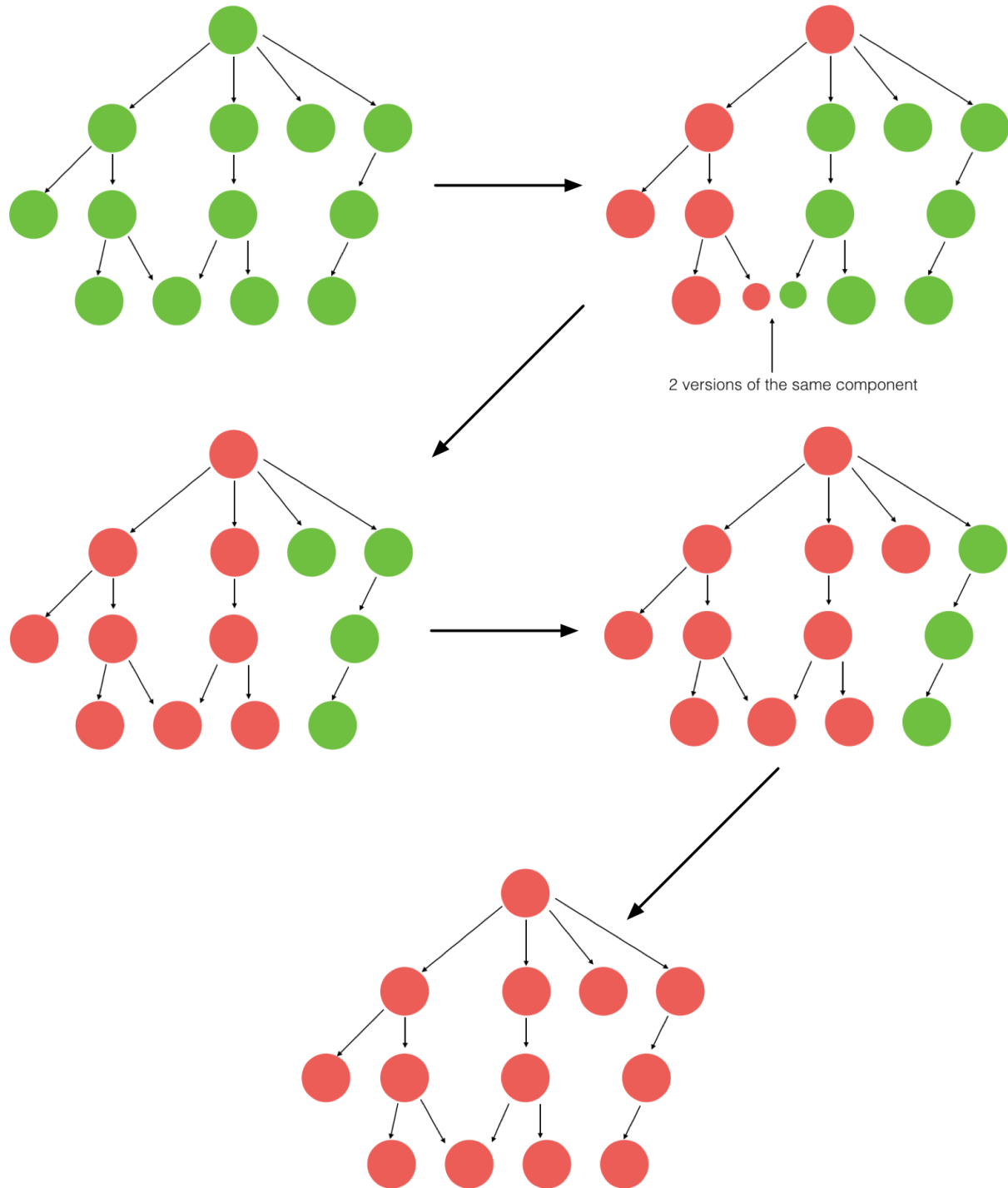
Even though, it is not feasible to rewrite the whole application at once, it is often possible to do it route by route, screen by screen, or feature by feature. In which case the whole route is either written in AngularJS or in Angular.



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



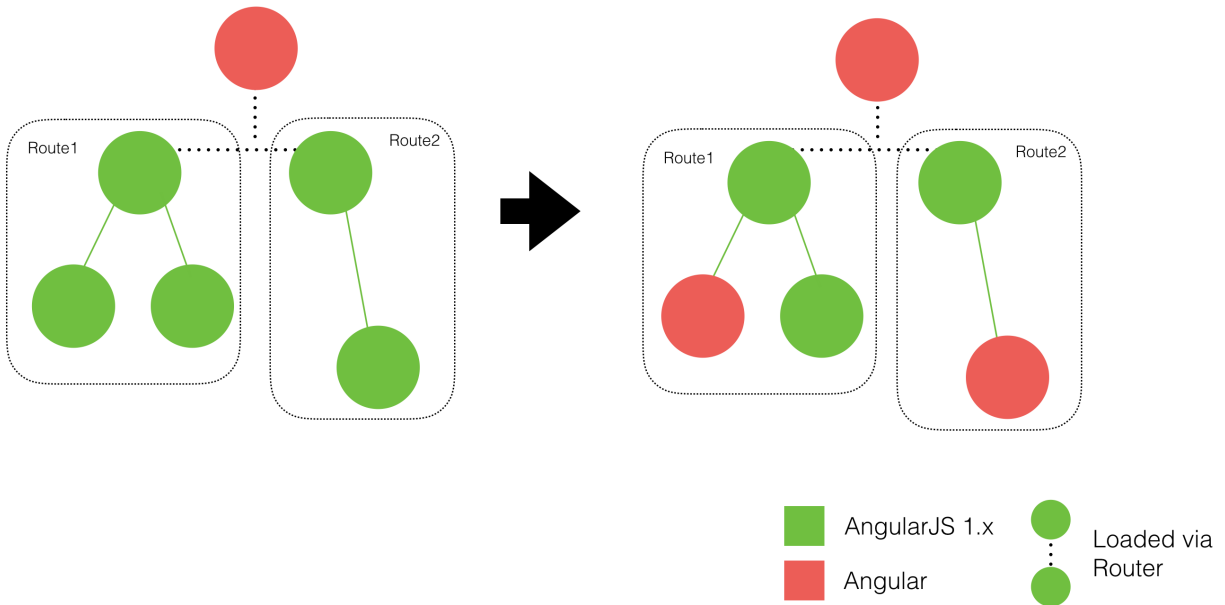
We can visualize this strategy as follows:



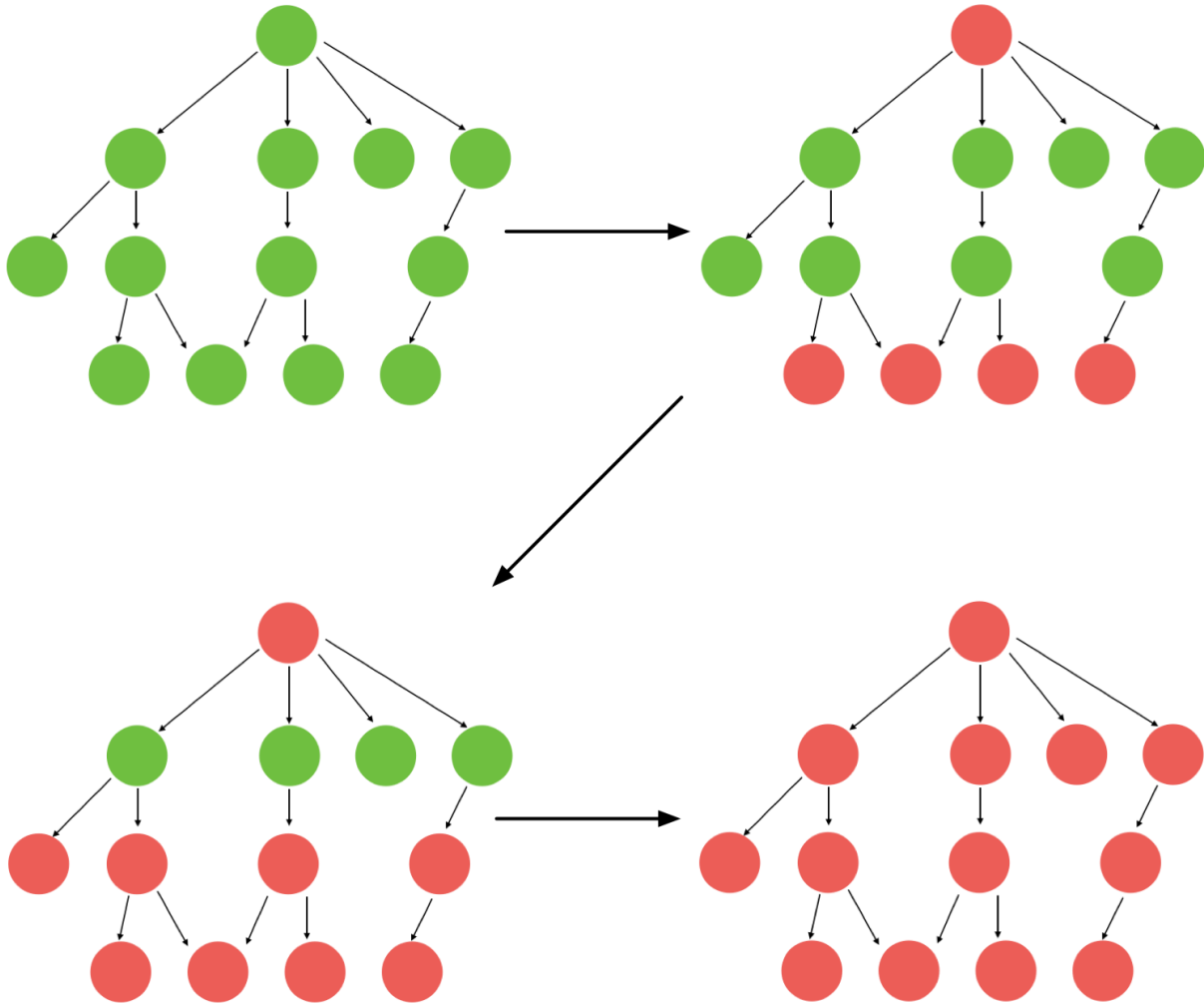
As you can see, while upgrading the second route, we had to duplicate one of the components. For a while we had two copies of that component: one written in AngularJS and the other in Angular. Such duplication is one of this approach's downside.

Horizontal Slicing

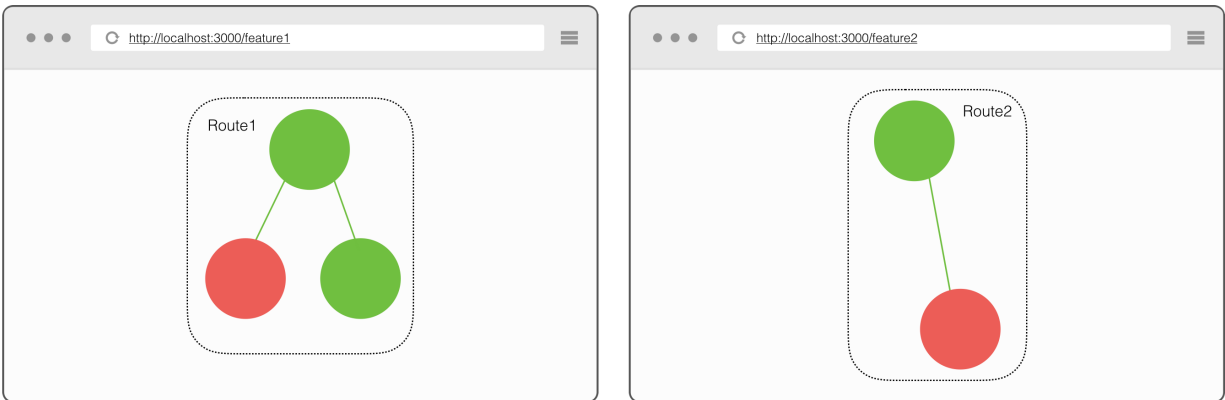
Horizontal slicing is the opposite strategy.



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.