MENU

UPGRADING ANGULAR APPS USING NGUPGRADE

*by **Pascal Precht** on Oct 24, 2015, last updated on Dec 18, 2016*
20 minute read

Contents are based on Angular version **>= 2.x** unless explicitly stated differently.

Upgrading an existing AngularJS application to Angular >= 2.x is surely one of the most interesting topics when it comes to Angular. A long time it has been unclear what a dedicated upgrade path will actually look like, since Angular was still in alpha state and APIs weren't stable yet, which makes it hard to "assume" where things will go and what's the best way to get there.

Earlier this year however, the Angular team has made an official announcement in which they talk about what are the available upgrade strategies and what things of both frameworks have to interoperate in order to run them side-by-side on the same website. While the blog post is rather a kind of birds-eye view where no code is shown, a dedicated design document has been created that gives a more concrete idea on what the APIs will look like.

Meanwhile, first implementations of `ngUpgrade` have landed in the code base and it's time to start digging into it. In this article we're going to explore what we can do to prepare for an upgrade, and of course how we eventually use `ngUpgrade` to upgrade our Angular application.

TABLE OF CONTENTS

Why upgrade?

One thing that seems to be a bit left out when people get scared that they can't upgrade to Angular >= 2.x for various reasons, is to think about if an upgrade is needed in the first place. Of course, Angular >= 2.x is the next major version of the framework and it will surely be the version we want to go with when building web applications in the future.

However, that doesn't mean that our existing Angular 1 applications aren't good enough anymore to survive the next generation of the web. It's not that once Angular 2.x is out, our Angular 1 applications immediately stop working. We still have a massive amount of websites out there that seem to be outdated, old and slow. Believe it or not, even if those websites are over 10 years old, they still work. That's the nice thing about the web, nothing is as backwards compatible because no one wants to break the web, right?

So before we think about going through the process upgrading we should really ask ourselves why we want to do it and if the applications we've built so far are really in a state that they need this upgrade.

Nonetheless there are some very strong arguments why one wants to upgrade and here are just a few of them:

- **Better Performance** - Angular comes with a way faster change detection, template precompilation, faster bootstrap time, view caching and plenty other things that make the framework freakin' fast.
- **Server-side Rendering** - The next version of Angular has been split up into two parts, an application layer and a render layer. This enables us to run Angular in other environments than the browser like Web Workers or even servers.
- **More powerful Templating** - The new template syntax is statically analyzable as discussed here, removes many directives and integrates better with Web Components and other elements.
- **Better Ecosystem** - Of course, at the time of writing this article, this is not true. But the Angular ecosystem will eventually be better and more interesting to us in the future.

There are surely more reasons why Angular >= 2.x is better than Angular 1.x, but keep in mind that we're talking about the motivation for upgrade here. Let's talk about how we can prepare for an actual upgrade.

Changes in Angular

In order to upgrade, we need to understand in what ways Angular >= 2.x is different. Unfortunately, covering the bigger differences between both version of the framework is out of the scope of this article. However, if you're entirely new to Angular, you might want to checkout our project Exploring Angular and get your feet wet.

Here's a list of changes that are crucial when thinking about upgrading:

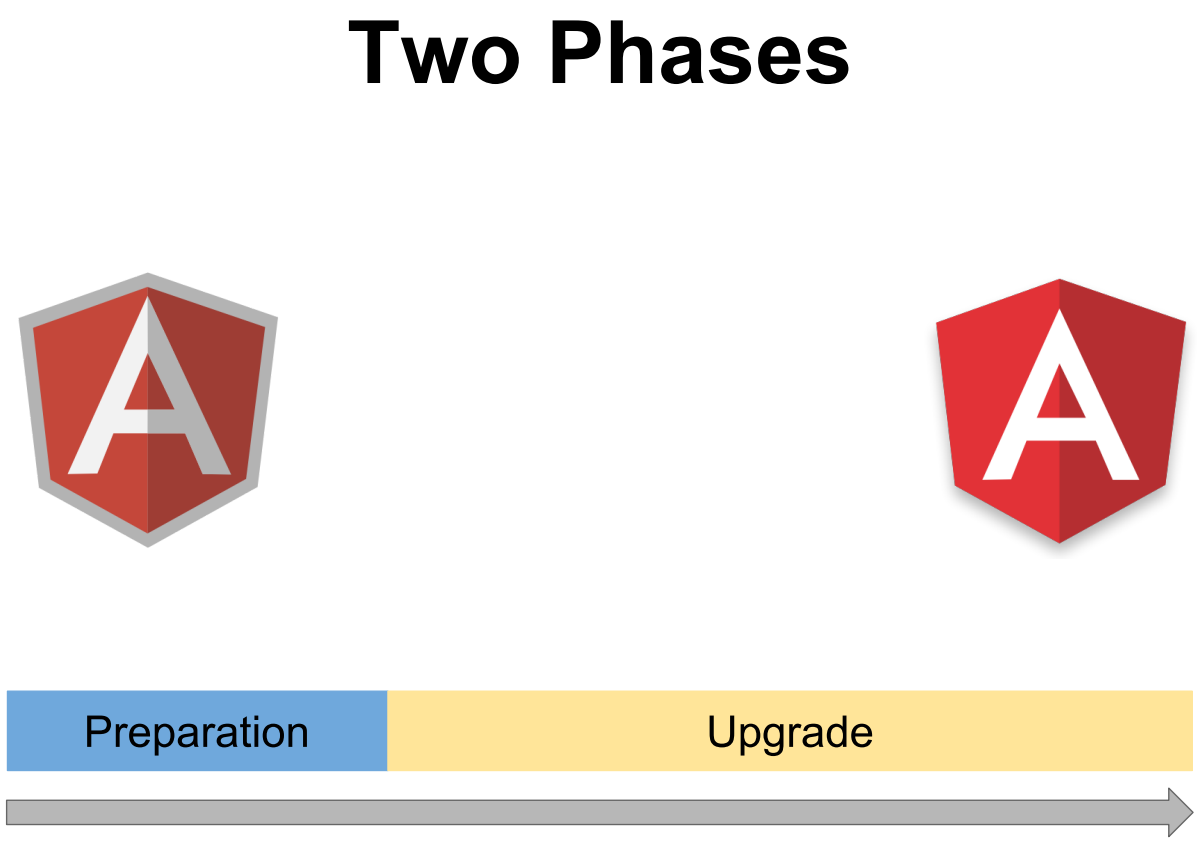
- **Components** - Components are the new building blocks when creating applications with Angular. Almost everything is a component, even our application itself.
- **Inputs/Outputs** - Components communicate via inputs and outputs, if they run in the Browser, these are element properties and events. Our article on demystifying Angular's Template syntax explains how they work.
- **Content Projection** - Basically the new transclusion, but more aligned with the Web Components standard.

- **Dependency Injection** - Instead of having a single injector for our entire application, in Angular each component comes with its own injector. We have a dedicated article on that too.

Of course, there are way more things in Angular that will change or be added to the framework, such as Routing, Forms, the Http layer and more.

How do we get there?

After doing tons of research at thoughtram with Christoph we realised, that the entire upgrade process can basically be categorized in two phases: Preparation and Upgrade.



Preparation

This is the phase that we could start off today. What can we do today to make the upgrade process later on easier? This includes several things like how we structure our application, which tools can we use or maybe even a language upgrade.

Upgrade

The phase of running both frameworks side-by-side. This is where `ngUpgrade` comes into play to make A1 and A2 components interoperable. Keep in mind that the goal of this phase is to stay in it as little as possible, since running both frameworks on the same website is surely not ideal.

Preparing for upgrade

Let's discuss what we can do today to prepare for an actual upgrade.

Layer application by feature or component

Oldie but goldie. We still see applications using a project structure where all directives go into `app/directives`, services go into `app/services`, controllers into `app/controllers` and so on and so forth. You get the idea. While this totally works in smaller applications, it doesn't really take us far when it comes to upgrading. We might want to upgrade component by component. Having an application layered by type rather than by feature/component, makes it harder to extract parts of the code base to upgrade it.

We should rather go with something like:

```
app
|- components
  |- productDetail
  | |- productDetail.js
  | |- productDetail.css
  | |- productDetail.html
  | |- productDetail.spec.js
  |- productList
  |- checkout
  |- wishlist
```

This structure allows us to take e.g. `productDetail` and upgrade it to Angular >= 2.x to integrate it back into the application later on. We don't have to worry if there's anything else related to `productDetail` in the code base that we might forget, because everything is in one place.

Use .service() instead of .factory()

If we plan to not only upgrade the framework, but also the language in which we're writing our application, we should definitely consider to use `.service()` instead of `.factory()` in all the cases where a service can be potentially a class. In Angular, a service is also just a class, so this seems like a logical thing to do. For more information on why `.service()` might be a better fit, read this article.

Write new components in ES2015 or TypeScript

This is an interesting one. When we saw Angular code the very first time, some of us were scared because all of a sudden there were classes and decorators. Despite the fact that we **don't** have to write our Angular apps in TypeScript (as explained in this article), ES2015 is the next standardized version, which means we will write it sooner or later anyways.

We wrote about how to write Angular in ES2015 today and if we do plan to upgrade but can't do it right now, we should definitely write our **new** components in ES2015 or TypeScript.

That being said, **it doesn't really make sense to upgrade existing code to ES2015 or TypeScript**. Even though it seems to be a logical step in preparation for upgrade, it doesn't help us in any way to take the existing and large code base and upgrade it to ES2015 or TypeScript first, before we upgrade the application to Angular >=2.x.

If we have to upgrade to Angular 2.x, it probably makes more sense to just rewrite component by component, without touching the existing code base. But this of course depends on how big our application is.

Again, this is just a language upgrade and it doesn't really help with the upgrade itself, however, it helps us and our team to get used to the new languages features as we're building components with it.

Use decorators in Angular 1?

Of course, we can take our code base closer to what Angular 2.x code would look like, by upgrading our language to TypeScript and use e.g. Decorators that have specifically been created for Angular 1. There are plenty community projects out there, some of them are a1atscript, angular2-now, angular-decorators and ng-classy.

They try solve the same problem, which is adding semantically useful decorators to Angular 1. But do they really help? **I don't think so**. They might improve the developer experience because all of a sudden we can use nice decorators that generate code for us, however,

they don't help when it comes to upgrading an application to Angular >= 2.x. One project, ng-forward, tries to make the exact same Angular syntax available in Angular 1.x.

This *can* be helpful to some extent since you and your team are getting familiar with how to write apps in Angular 2.x while writing Angular 1.x code. On the other hand, it could also be confusing when trying to squeeze Angular 2.x concepts and syntax into the Angular 1.x world. We'll see how practical it is once projects are starting to use it.

Upgrade Strategies

Now that we know what we can do to prepare for an upgrade, let's take a look at the different upgrade strategies available to see which one makes more sense to us.

There are basically two strategies:

- **Big Bang** - Start a spike in Angular >= 2.x and replace entire app once done
- **Incremental** - Upgrade existing app once service or component at a time

Which one should we use?

This really depends! If our application is rather small a big bang rewrite is probably the easiest and fastest way to upgrade. If our application is rather large and it's deployed continuesly, we can't just upgrade the whole thing at once. We need a way to do it step by step, component by component, service by service. This is where the incremental upgrade comes into play.

In the end it's really a matter of how much time we have available to process the upgrade. We will focus on incremental upgrade, since this is what the majority of developers want to understand and see how it works.

Upgrading using ngUpgrade

In order to run both frameworks side-by-side and make components interoperable, the Angular projects comes with a module `ngUpgrade`. The module basically acts as an adapter facade, so we don't really feel that there are two frameworks running side-by-side.

For this to work, four things need to interoperate:

- **Dependency Injection** - Exposing Angular services into Angular 1.x components and vice-versa.
- **Component Nesting** - Angular 1 directives can be used in Angular 2.x components and Angular 2.x components can used Angular 1 directives
- **Content Projection / Transclusion** - Angular 1 components transclude Angular 2.x components and Angular 2.x component project Angular 1 directives
- **Change Detection** - Angular 1 scope digest and change detectors in Angular >= 2.x are interleaved

With these four things being interoperable, we can already start upgrading our applications component by component. Routing is another part that can help but is not necessarily mandatory, since we can totally stick with any Angular 1 routing system while upgrading.

The typical upgrade process

Here's what a typical upgrade process would look like:

- Include Angular and upgrade module
- Pick component to upgrade and change its controller and template Angular 2.x syntax (this is now an Angular 2.x component)
- Downgrade Angular 2.x component to make it run in Angular 1.x app
- Pick a service to upgrade, this usually requires little amount of change (especially if we're on ES2015)
- Repeat step 2 and 3 (and 4)
- Replace Angular 1 bootstrap with Angular 2.x bootstrap

Let's use `ngUpgrade` to upgrade our components to Angular 2.x!

Bootstrapping with ngUpgrade

The first thing we need to do is to upgrade our Angular 1 application with `ngUpgrade`. Whenever we upgrade an Agular app, we **always** have an Angular 1.x module being bootstrap at root level. This means, during the process of upgrade, Angular components are always bootstrap inside Angular 1.x components.

Let's start with an app we want to upgrade;

```
var app = angular.module('myApp', []);
```

Plain old Angular 1 module. Usually, this module is bootstrapped using the `ng-app` attribute, but now we want to bootstrap our module using `ngUpgrade`. We do that by removing the `ng-app` attribute from the HTML, create an `ngUpgrade` adapter from the upgrade module, and call `bootstrap()` on it with `myApp` as module dependency:

```
import { UpgradeAdapter } from '@angular/upgrade';

var adapter = new UpgradeAdapter();
var app = angular.module('myApp', []);

adapter.bootstrap(document.body, ['myApp']);
```

Cool, our app is now bootstrapped using `ngUpgrade` and we can start mixing Angular 1.x components with Angular 2.x components. However, in a real world application, you want to create an instance of `UpgradeAdapter` in a separate module and import it where you need it. This leads to cleaner code when upgrading across your application.

Downgrading Angular 2.x components

Let's upgrade our first component to Angular 2.x and use it in our Angular 1.x application. Here we have a `productDetail` component that needs to be upgraded:

```
app.component('productDetail', () => {
  bindings: {
    product: '='
  },
  controller: 'ProductDetailController',
  template: `
    <h2>{{$ctrl.product.name}}</h2>
    <p>{{$ctrl.product.description}}</p>
  `
});
```

Upgrading this to Angular 2.x looks something like this:

```
@Component({
  selector: 'product-detail',
  template: `
    <h2>{{product.name}}</h2>
    <p>{{product.description}}</p>
  `
})
class ProductDetail {
  @Input() product: Product;
}
```

Note: You might want to define this component in a separate file, for simplicity sake we defined it in place.

Perfect! But how do we get this Angular component into our Angular 1.x application? The `UpgradeAdapter` we've created comes with a method `downgradeNg2Component()`, which takes an Angular component and creates an Angular 1.x directive from it. Let's use our Angular component in Angular 1.x world.

```
app.directive('productDetail',
  adapter.downgradeNg2Component(ProductDetail));
```

Yay! That's it! The adapter will bootstrap this component from within the Angular 1 template where it's used.

But wait, our Angular 2.x component is just an Angular 1.x component eventually? Yes and no. The directive is controlled by Angular 1.x, but the component's view will be controller by Angular >= 2.x. This means the resulting Angular 1.x components takes advantage of Angular 2.x features and performance.

Upgrading Angular 1 components

There might be cases where one component has already been upgraded to Angular >= 2.x, but it still uses Angular 1.x directives in its template. `ngUpgrade` allows us to use Angular 1.x directives in Angular 2.x components by upgrading them using `upgradeNg1Component()`.

Let's say we continued upgrading our application and have a `ProductList` component like this:

```
@Component({
  selector: 'product-list',
  template: `
    <h2>Product List</h2>
    <ol>
      <li *ngFor="let product of products">
        <product-list-item [product]="product">
        </product-list-item>
      </li>
    </ol>
  `
})
class ProductList {
  @Input() products: Product[];
  ...
}
```

`<product-list-item>` is a component that hasn't been ported to Angular 2.x yet and maybe can't even for some reason. It needs to be upgraded but how do we get there? As you can see, there's a `directives` property in the `@Component()` metadata. This property defines which directives are used in the component's template. What we need is a way to add `<product-list-item>` there too.

`upgradeNg1Component()` enables us to do exactly that. It takes the name of a directive that has been registered somewhere on our Angular 1 module and upgrades it to an Angular 2.x component. Here's what it looks like:

```
@NgModule({
  imports: [BrowserModule],
  declarations: [
    AppComponent,
    ProductList,
    adapter.upgradeNg1Component('productListItem')
  ],
  ...
})
export class AppModule {}
```

All we need to do is to downgrade `ProductList` item and we can use it right away!

Adding Angular Providers

Upgrading components is probably the most crucial part in the entire upgrade process. Sometimes however, components have service dependencies which need to work in both worlds too. Luckily, `ngUpgrade` provides APIs for that.

Let's say our `ProductDetail` component, already upgraded to Angular >= 2.x, has a `ProductService` dependency.

```
class ProductService {

}

@Component()
class ProductDetail {
  constructor(productService: ProductService) {

  }
}
```

In Angular >= 2.x, we have to add a provider configuration for the component's injector, but since we don't bootstrap using Angular 2.x, there's no way to do so. `ngUpgrade` allows us to add a provider using the `addProvider()` method to solve this scenario.

```
adapter.addProvider(ProductService);
```

That's all we need to do!

Upgrading Angular 1 Providers

Let's say our `ProductService` depends on another lower level `DataService` to communicate with a remote server. `DataService` is already implemented in our Angular 1 application but not yet upgraded to Angular 2.x.

```
class ProductService {
```

```
  constructor(@Inject('DataService') dataService) {
    ...
  }
}

app.service('DataService', () => {
  ...
});
```

As you can see, we're using `@Inject` to specify the provider token for `DataService`, since we don't have a `DataService` type. If this is unclear, you might want to read our articles on DI in Angular.

However, there's no provider for `'DataService'` in Angular 2.x world yet. Let's make it available using `upgradeNg1Provider()`.

```
adapter.upgradeNg1Provider('DataService');
```

Boom! We can make it even better. Let's assume our Angular 1 service has already been written as class.

```
class DataService {

}

app.service('DataService', DataService);
```

We can use that class as type and token for dependency injection in Angular 2.x. All we have to do, is to upgrade our service with that token.

```
adapter.upgradeNg1Provider('DataService', {asToken: DataService});
```

Now we can inject it using plain old type annotations.

```
@Injectable()
class ProductService {

  constructor(dataService: DataService) {
    ...
```

```
    }
  }
```

Note: We added `@Injectable()` to our service because TypeScript needs at least one decorator to emit metadata for it. Learn more in our article on Injecting Services in Services in Angular.

Downgrading Angular 2.x Providers

Last but not least, we might need to be able to use Angular 2.x services in Angular 1.x components. Guess what, `ngUpgrade` comes with a `downgradeNg2Provider()` method.

```
app.factory('ProductService',
  adapter.downgradeNg2Provider(ProductService));
```

Conclusion

`ngUpgrade` provides many useful APIs and is a big step forward when it comes to truly upgrading an application from Angular 1 to Angular >= 2.x. At AngularConnect we gave a workshop on upgrading and we've open sourced a repository that shows all the steps we've been through, from preparation to upgrade. Make sure to check out the `steps` branch.

Hopefully this article made a bit more clear what this whole upgrade story is all about!

AUTHOR

Pascal Precht

Pascal is a front-end engineer and a Angular Developer Expert nominated by Google. He created the angular-translate module, is an Angular contributor and also part of the Angular Docs Authoring team.

Twitter GitHub

RELATED POSTS

A web animations deep dive with Angular

Angular comes with a built-in animation system that lets us create powerful animations based on the Web Animations API. In...

Custom themes with Angular Material

Angular Material offers great theming capabilities for both, built-in and custom themes. In this article we'll explore how to make...

Angular Master Class - Redux and ngrx

Today we're super excited to announce that we finished working on our new Angular Master Class courseware. Read on for...

Three things you didn't know about the AsyncPipe

This article explains three lesser known features of the AsyncPipe that help us to write better async code.

Using Zones in Angular for better performance

In this article we'll take a look at how to use Zone APIs to improve our app's performance!

Making your Angular apps fast

In this article we discuss tips and tricks to make Angular blazingly fast!

This website was created in collaboration with **Tim Cheung** and **Tim Hartmann**.
Code of Conduct • Legal notice