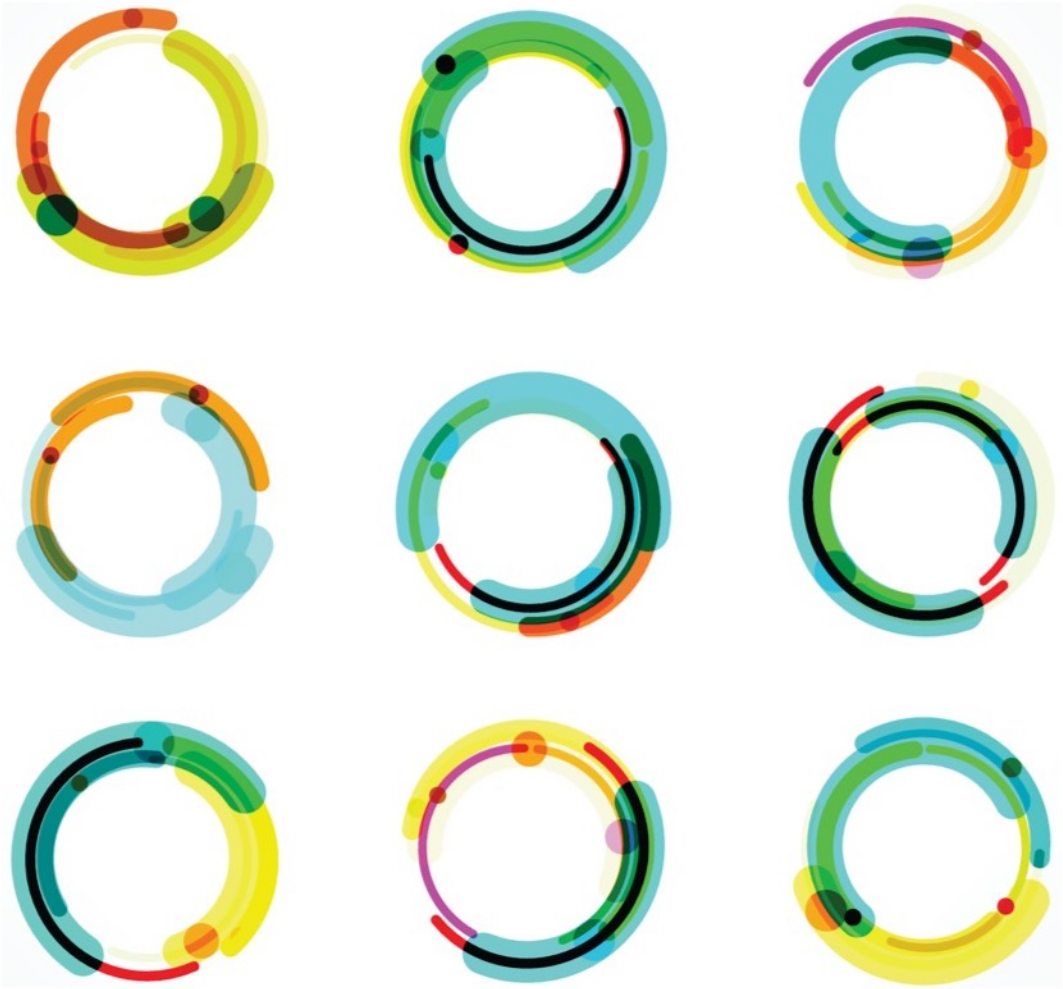
ESSENTIAL ANGULAR

VICTOR SAVKIN & JEFF CROSS



Essential Angular

Victor Savkin, Jeff Cross and Nrwl.io

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

This version was published on 2017-01-27

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

About the Authors

Victor Savkin and Jeff Cross are core contributors to the Angular projects. Victor has been on the Angular team since the inception of Angular 2. Victor developed dependency injection, change detection, forms, and the router. Jeff was one of the earliest core team members on Angular 1. He developed the Angular 2 http and AngularFire2 modules, contributed to RxJS 5, and was most recently the Tech Lead of the Angular Mobile team at Google.

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

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

What is this book about?

This book aims to be a short, but at the same time, fairly complete overview of the key aspects of Angular: it covers the framework's mental model, its API, and the design principles behind it.

To make one thing clear: this book is not a how-to-get-started guide. There is a lot of information about it available online. The goal of this book is different. Read this book after you toyed around

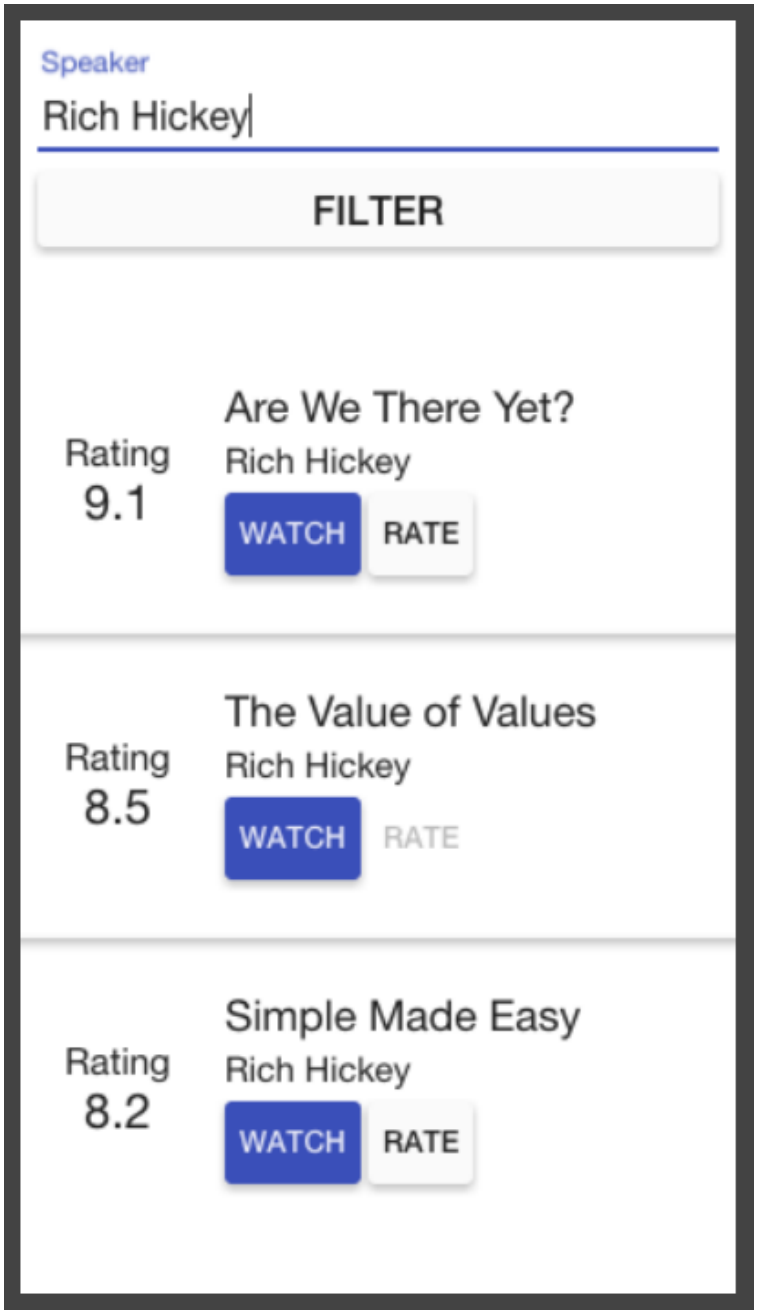
[1] http://nrwl.io

with the framework, but before you embark on writing your fist serious Angular application. The book will give you a strong foundation. It will help you put all the concepts into right places. So you will have a good understanding of why the framework is the way it is.

Let's get started!

Example

For most of the examples in this book we will use the same application. This application is a list of tech talks that you can filter, watch, and rate.

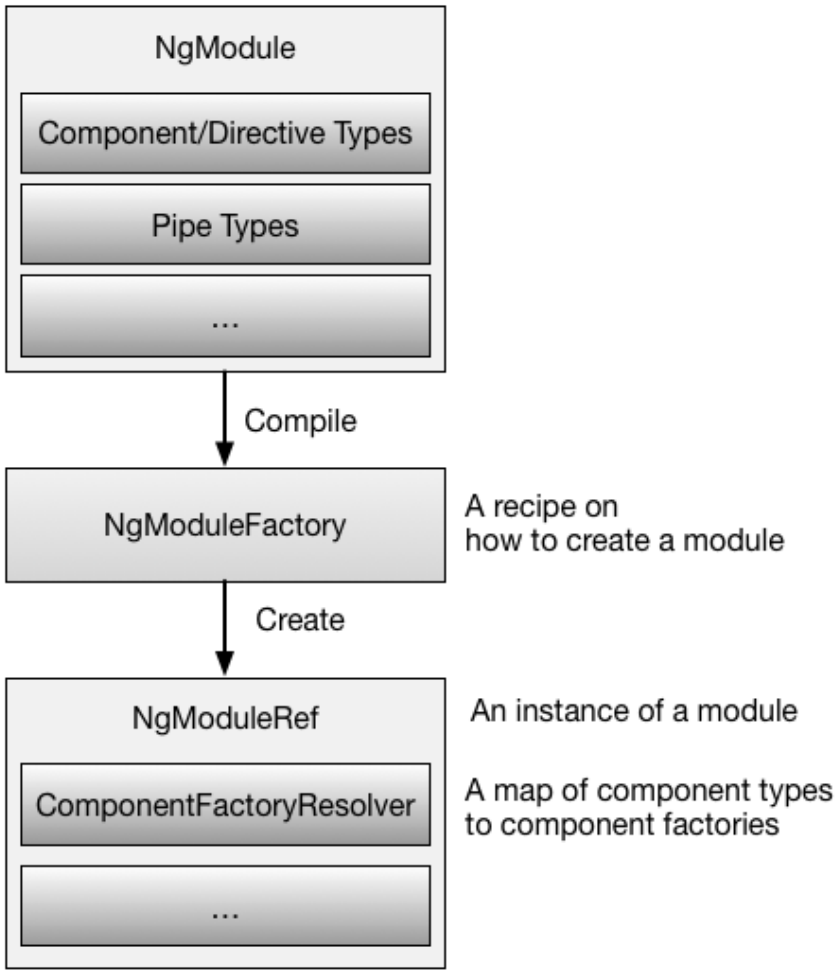


You can find the source code of the application here https://github.com/vsavkin/essential-angular-book-app[2]

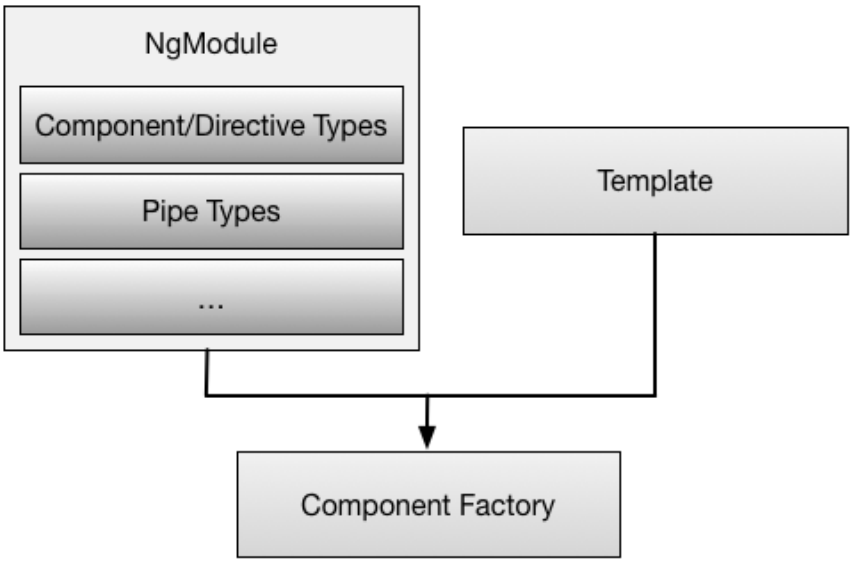
[2] https://github.com/vsavkin/essential-angular-book-app

Chapter 1: Compilation

At the core of Angular is a sophisticated compiler, which takes an `NgModule` type and produces an `NgModuleFactory`.



An `NgModule` has components declared in it. While creating the module factory, the compiler will take the template of every component in the module, and using the information about declared components and pipes, will produces a component factory. The component factory is a JavaScript class the framework can use to stamp out components.



JIT and AOT

Angular 1 is a sophisticated HTML compiler that generates code at runtime. New versions of Angular have this option too: they can generate the code at runtime, or just in time (JIT). In this case the compilation happens while the application is being bootstrapped. But they also has another option: they can run the compiler as part of application's build, or ahead of time (AOT).

Why would I want to do it?

Compiling your application ahead of time is beneficial for the following reasons:

- We no longer have to ship the compiler to the client. And so it happens, the compiler is the largest part of the framework. So it has a positive effect on the download size.
- Since the compiled app does not have any HTML, and instead has the generated TypeScript code, the TypeScript compiler can analyze it to produce type errors. In other words, your templates are type safe.
- Bundlers (e.g., WebPack, Rollup) can tree shake away everything that is not used in the application. This means that you no longer have to create 50-line node modules to reduce the download size of your application. The bundler will figure out which components are used, and the rest will be removed from the bundle.
- Finally, since the most expensive step in the bootstrap of your application is compilation, compiling ahead of time can significantly improve the bootstrap time.

To sum up, using the AOT compilation makes your application bundles smaller, faster, and safer.

How is it possible?

Why did not we do it before, in Angular 1? To make AOT work the application has to have a clear separation of the static and dynamic data in the application. And the compiler has to built in such a way that it only depends on the static data. When designing and building Angular we put a lot of effort to do exactly that. And such primitives as classes and decorators, which the new versions of JavaScript and TypeScript support, made it way easier.

To see how this separation works in practice, let's look at the following example. Here, the information in the decorator is known statically. Angular knows the selector and the template of the talk component. It also knows that the component has an input called `talk` and an output called `rate`. But the framework does not know what the constructor or the `onRate` function do.

```
@Component({
  selector: 'talk-cmp',
  template: `
    {{talk.title}} {{talk.speaker}}
    Rating: {{ talk.rating | formatRating }}
    <watch-button [talk]="talk"></watch-button>
    <rate-button [talk]="talk" (click)="onRate()"></rate-button>
  `
})
class TalkCmp {
  @Input() talk: Talk;
  @Output() rate: EventEmitter;


  constructor() {
    // some initialization logic
  }


  onRate() {
    // reacting to a rate event
  }
}
```

Since Angular knows all the necessary information ahead of time, it can compile this component without actually executing any application code, as a build step.

Trade-offs

Since AOT is so advantageous, we recommend to use it in production. But, as with everything, there are trade-offs. For Angular to be able to compile your application ahead of time, the metadata has to be statically analyzable. For instance, the following code will not work in the AOT mode:

```
@Component({
  selector: 'talk-cmp',
  template: () => window.hide ? 'hidden' : `
    {{talk.title}} {{talk.speaker}}
    Rating: {{ talk.rating | formatRating }}
    <watch-button [talk]="talk"></watch-button>
    <rate-button [talk]="talk" (click)="onRate()"></rate-button>
  `
})
class TalkCmp {
  //...
}
```

The `window.hide` property will not be defined. So the compilation will fail pointing out the error. A lot of work has been done to make the compiler smarter, so it can understand most of the day-to-day patterns you would use when building your application. But certain things will never work, like the example above.

Let's Recap

- The central part of Angular is its compiler.
- The compilation can be done just in time (at runtime) and ahead of time (as a build step).
- The AOT compilation creates smaller bundles, tree shakes dead code, makes your templates type-safe, and improves the bootstrap time of your application.
- The AOT compilation requires certain metadata to be known statically, so the compilation can happen without actually executing the code.

Chapter 2: NgModules

Declarations, Imports, and Exports

`NgModules` are the unit of compilation and distribution of Angular components and pipes. In many ways they are similar to ES6 modules, in that they have declarations, imports, and exports.

Let's look at this example:

```
@NgModule({
  declarations: [FormattedRatingPipe, WatchButtonCmp, RateButtonCmp, TalkCmp, Ta\
lksCmp],
  exports: [TalksCmp]
})
class TalksModule {}

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

Here we define two modules: `TalksModule` and `AppModule`. `TalksModule` has all the components and pipes that do actual work in the application, whereas `AppModule` has only `AppCmp`, which is a thin application shell.

`TalksModule` declares four components and one pipe. The four components can use each other in their templates, similar to how classes defined in an ES module can refer to each other in their methods. Also, all the components can use `FormattedRatingPipe`. So an `NgModule` is the compilation context of its components, i.e., it tells Angular how these components should be compiled. As with ES, a component can only be declared in one module.

In this example `TalksModule` exports only `TalksCmp`–the rest is private. This means that only `TalksCmp` is added to the compilation context of AppModule. Again this is similar to how the export keyword works in JavaScript.

Summary

- NgModules are akin to ES modules: they have declarations, imports, and exports.
- NgModules define the compilation context of their components.

Bootstrap and Entry Components

The bootstrap property defines the components that are instantiated when a module is bootstrapped. First, Angular creates a component factory for each of the bootstrap components. And then, at runtime, it'll use the factories to instantiate the components.

To generate less code, and, as a result, to produce smaller bundles, Angular won't generate component factories for any `TalksModule`'s components. The framework can see their usage statically, it can inline their instantiation, so no factories are required. This is true for any component used statically (or declaratively) in the template.

For instance, let's look at `TalkCmp`:

```
@Component({
  selector: 'talk-cmp',
  template: `
    {{talk.title}} {{talk.speaker}}
    {{talk.rating | formatRating}}
    <watch-button [talk]="talk"></watch-button>
    <rate-button [talk]="talk"></rate-button>
  `
})
class TalkCmp {
  @Input() talk: Talk;
  @Output() rate: EventEmitter;
  //...
}
```

Angular knows, at compile time, that `TalkCmp` uses `WatchButtonCmp` and `RateButtonCmp`, so it can instantiate them directly, without any indirection or extra abstractions.

Now let's look at a different component that uses the router.

```
@Component({
  selector: 'router-cmp',
  template: `
    <router-outlet></router-outlet>
  `
})
class RouterCmp {}

@NgModule({
  declarations: [RouterCmp],
  imports: [BrowserModule, RouterModule, TalksModule],
  bootstrap: [RouterCmp],
  providers: [
    {provide: ROUTES, useValue: [
      { path: 'talks', component: TalksCmp },
      { path: 'settings', component: SettingsCmp }
    ]}
  ]
})
class RouterModule {}
```

Angular cannot statically figure out what components can be loaded into the outlet, and, as a result, cannot instantiate them directly. Here we need the extra abstraction, we need the component factories for both `TalksCmp` and `SettingsCmp`. We can tell Angular to generate those by listing them as entry components.

```
@NgModule({
  declarations: [RouterCmp],
  imports: [BrowserModule, RouterModule, TalksModule],
  bootstrap: [RouterCmp],
  entryComponents: [TalksCmp, SettingsCmp],
  providers: [
    {provide: ROUTES, useValue: [
      { path: 'talks', component: TalksCmp },
      { path: 'settings', component: SettingsCmp }
    ]}
  ]
})
class RouterModule {}
```

Even though we do not use `TalksCmp` or `SettingsCmp` in any template, the router configuration is still static. And it is cumbersome to declare every component used by the router in the entry

components. Because this is so common, Angular supports a special provider token to automatically pre-populate entryComponents.

```
@NgModule({
  declarations: [RouterCmp],
  imports: [BrowserModule, RouterModule, TalksModule],
  bootstrap: [RouterCmp],
  providers: [
    {provide: ROUTES, useValue: [
        { path: 'talks', component: TalksCmp },
        { path: 'settings', component: SettingsCmp }
    ]},
    {provide: ANALYZE_FOR_ENTRY_COMPONENTS, multi: true, useExisting: ROUTES}
  ]
})
class RouterModule {}
```

And when using `RouterModule.forRoot` or `RouterModule.forChild`, the router module takes care of it.

```
@NgModule({
  declarations: [RouterCmp],
  imports: [BrowserModule, TalksModule, RouterModule.forRoot([
    { path: 'talks', components: TalksCmp },
    { path: 'settings', components: SettingsCmp }
  ])],
  bootstrap: [RouterCmp]
})
class RouterModule {}
```

Summary

- To be more efficient, Angular separates components used statically (declaratively) from the components used dynamically (imperatively).
- Angular directly instantiates components used statically; no extra abstraction is required.
- Angular generates a component factory for every component listed in entryComponents, so that they can be instantiated imperatively.

Providers

I'll cover providers and dependency injection in Chapter 4. Here I'd like to just note that NgModules can contain providers. And the providers of the imported modules are merged with the target

module's providers, left to right, i.e., if multiple imported modules define the same provider, the last module wins.

```
@NgModule({
  providers: [
    Repository
  ]
})
class TalksModule {}

@NgModule({
  imports: [TalksModule]
})
class AppModule {}
```

Injecting NgModules and Module Initialization

Angular instantiates NgModules and registers them with dependency injection. This means that you can inject modules into other modules or components, like this:

```
@NgModule({
  imports: [TalksModule]
})
class AppModule {
  constructor(t: TalksModule) {}
}
```

This can be useful for coordinating the initialization of multiple modules, as shown below:

```
@NgModule({
  imports: [ModuleA, ModuleB]
})
class AppModule {
  constructor(a: ModuleA, b: ModuleB) {
    a.initialize().then(() => b.initialize());
  }
}
```

Bootstrap

To bootstrap an Angular application in the JIT mode, you pass a module to bootstrapModule.

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {AppModule} from './app';

platformBrowserDynamic().bootstrapModule(AppModule);
```

This will compile `AppModule` into a module factory and then use the factory to instantiate the module. If you use AOT, you may need to provide the factory yourself.

```
import {platformBrowser} from '@angular/platform-browser';
import {AppModuleNgFactory} from './app.ngfactory';

platformBrowser().bootstrapModuleFactory(AppModuleNgFactory);
```

I said "may need to" because the CLI and the WebPack plugin take care of it for you. They will replace the `bootstrapModule` call with `bootstrapModuleFactory` when needed.

Lazy Loading

As I mentioned above NgModules are not just the units of compilation, they are also the units of distribution. That is why we bootstrap an `NgModule`, and not a component. We don't distribute components, we distribute modules. And that's why we lazy load NgModules as well.

```
import {NgModuleFactoryLoader, Injector} from '@angular/core';

class MyService {
  constructor(loader: NgModuleFactoryLoader, injector: Injector) {
    loader.load("mymodule").then((f: NgModuleFactory) => {
      const moduleRef = f.create(injector);
      moduleRef.injector; // module injector
      moduleRef.componentFactoryResolver; // all the components factories of the\
 lazy-loaded module
    });
  }
}
```

The loader compiles the modules if the application is running in the JIT mode, and does not in the AOT mode. The default loader `@angular/core` ships with uses SystemJS, but, as most things in Angular, you can provide your own.

Let’s Recap

- NgModules are the units of compilation. They tell Angular how components should be compiled.
- Similar to ES module they have declarations, imports, and exports.
- Every component belongs to a NgModule.
- Bootstrap and entry components are configured in NgModules.
- NgModules configure dependency injection.
- NgModules are used to bootstrap applications.
- NgModules are used to lazy load code.