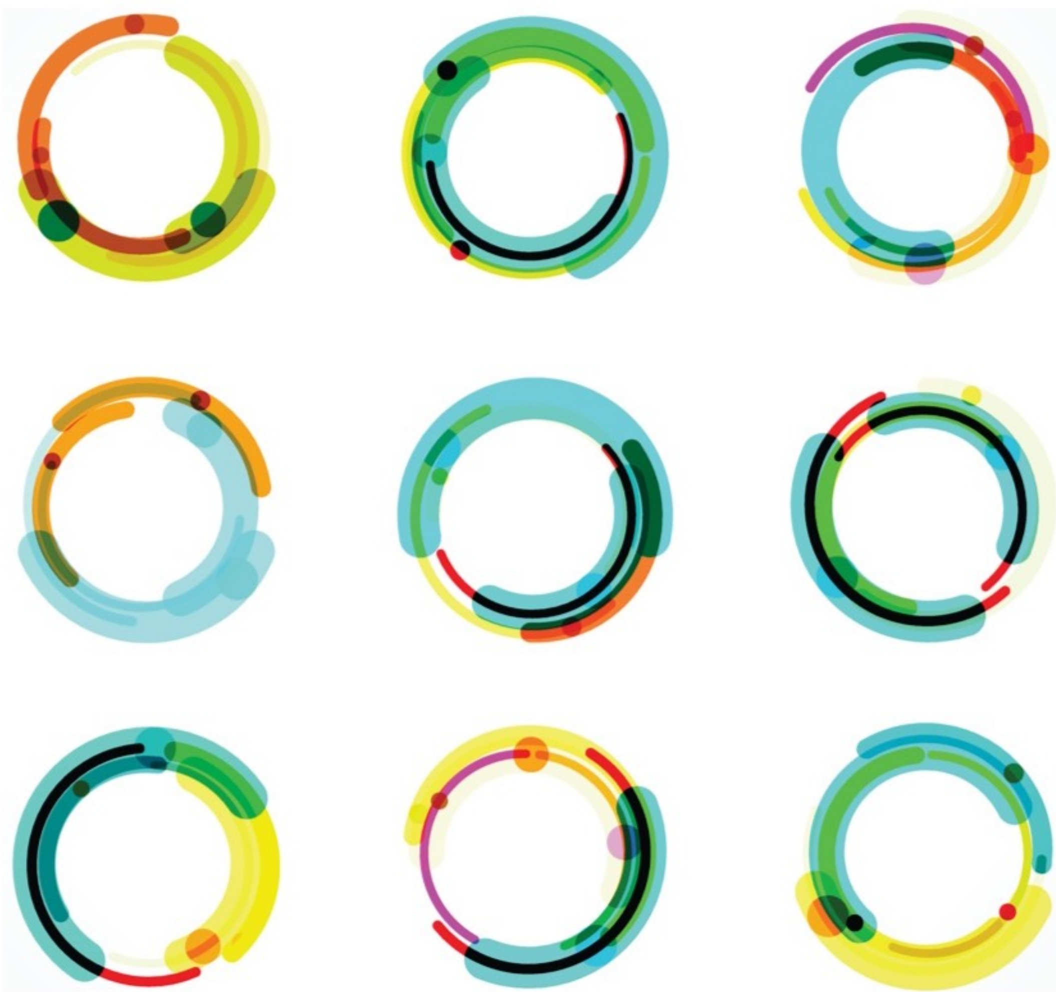
# ESSENTIAL

VICTOR SAVKIN & JEFF CROSS

# ANGULAR



Angular consulting for enterprise customers,
from core contributors

# 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 2017-01-24

Leanpub

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

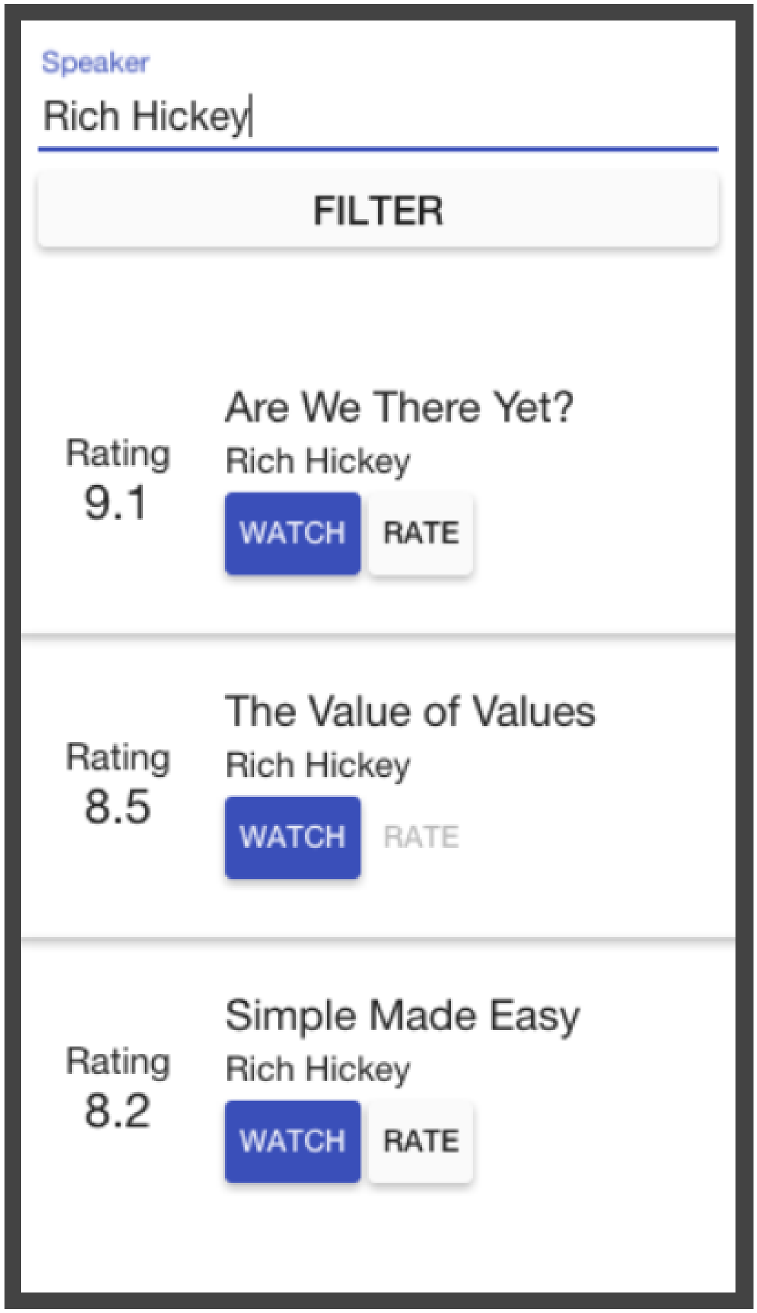
[1]http://nrwl.io

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.

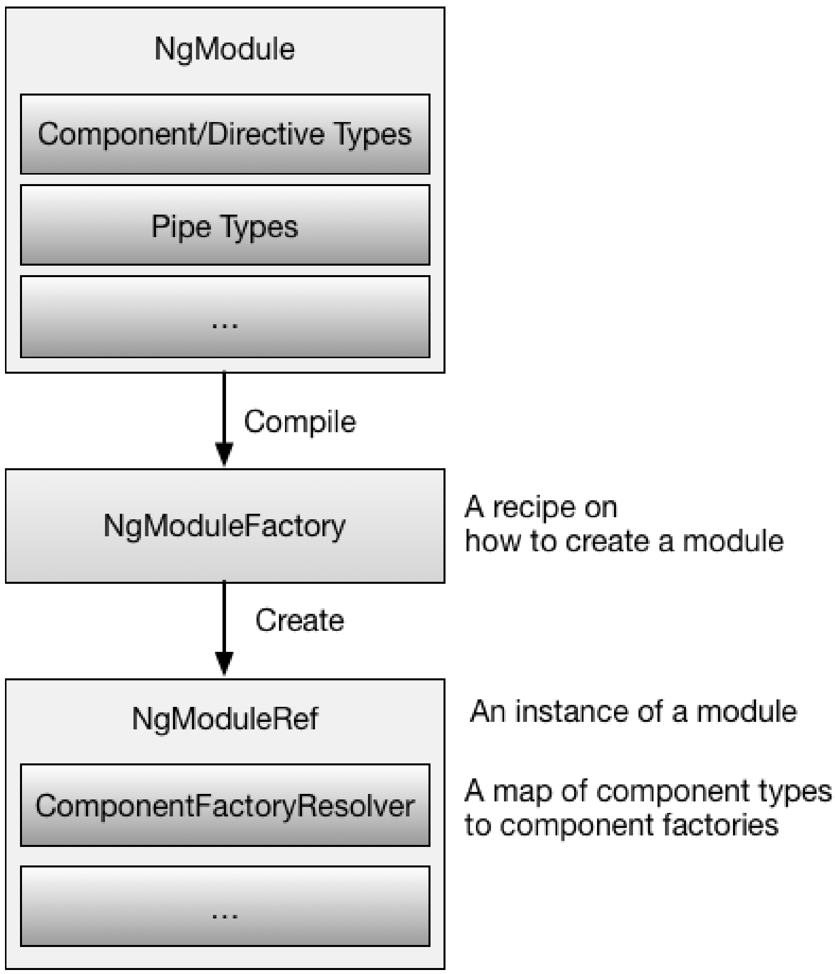


You can find the source code of the application here https://github.com/vsavkin/essential-angular-book-app[2]

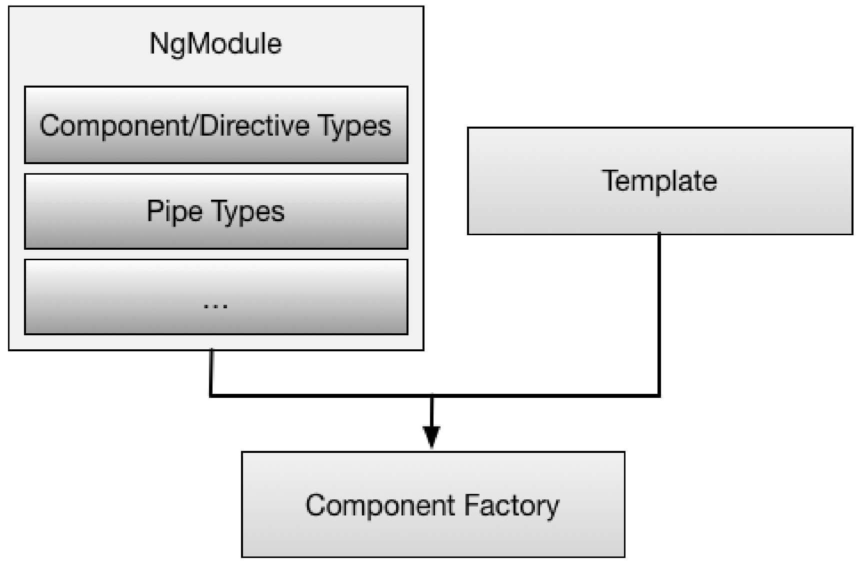
[2] https://github.com/vsavkin/essential-angular-book-app

# 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
    });
  }
}
```

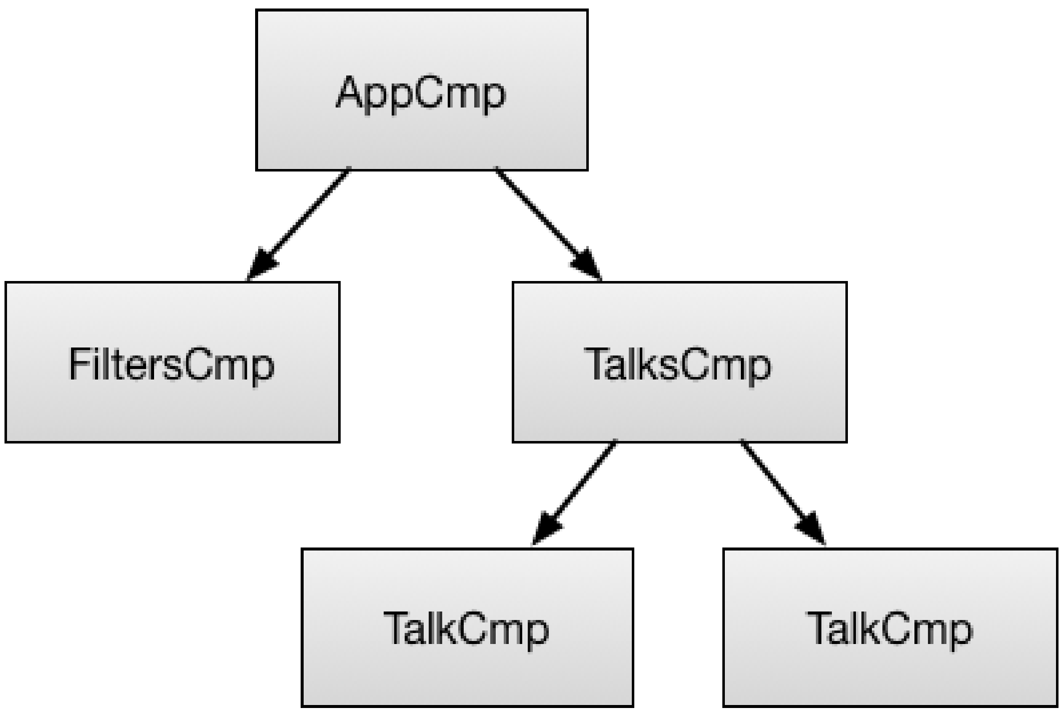
The loader compiles the modules if the application is running in the JIT mode, and does not in the AOT mode. The default loader `@angular/core` ships with uses SystemJS, but, as most things in Angular, you can provide your own.

## 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.

# Chapter 3: Components and Directives

**To build an Angular application you define a set of components, for every UI element, screen, and route. An application will always have root components (usually just one) that contain all other components**. To make things simpler, in this book let's assume the application has a single root component, and thus our Angular application will have a component tree, that may look like this:




`AppCmp` is the root component. The `FiltersCmp` component has the speaker input and the filter button. `TalksCmp` is the list you see at the bottom. And TalkCmp is an item in that list. To understand what constitutes a component in Angular, let's look closer at TalkCmp.

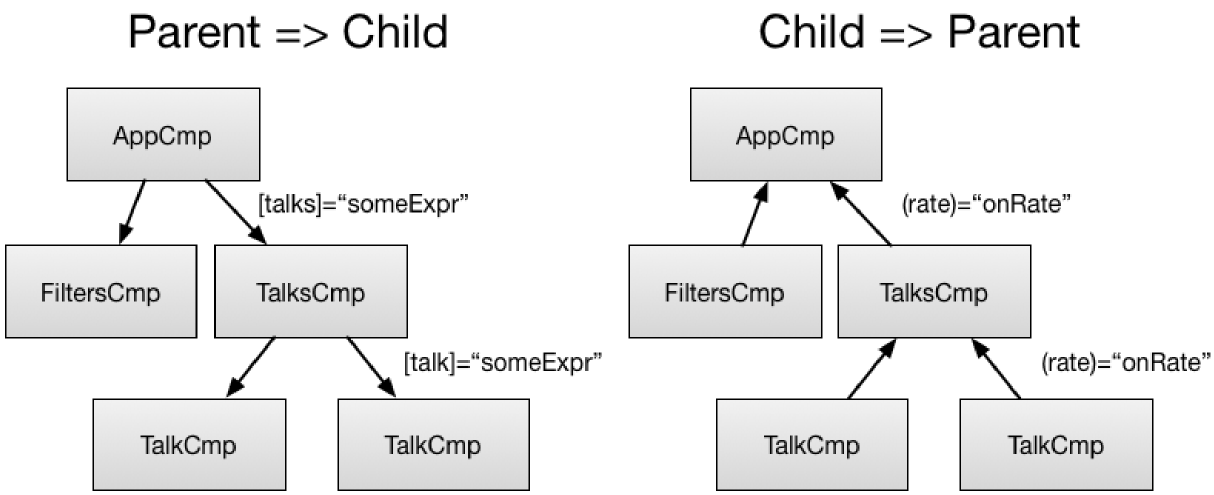
```
@Component({
  selector: 'talk-cmp',
  template: `
    {{talk.title}} {{talk.speaker}}
    {{talk.rating | formatRating }}
    <watch-button [talk]="talk"></watch-button>
    <rate-button [talk]="talk"></rate-button>
  `
})
class TalkCmp {
  @Input() talk: Talk;
  @Output() rate: EventEmitter;
 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

## Input and Output Properties

**A component has input and output properties, which can be defined in the component decorator or using property decorators.**

```
class TalkCmp {
  @Input() talk: Talk;
  @Output() rate: EventEmitter;
  //...
}
```

**Data flows into a component via input properties. Data flows out of a component via output properties**, hence the names: 'input' and 'output'.




**Input and output properties are the public API of a component**. You use them when you instantiate a component in your application.

```
<talk-cmp [talk]="someExp" (rate)="onRate($event.rating)"></talk-cmp>;
```

You can set input properties using property bindings, through square brackets. You can subscribe to output properties using event bindings, through parenthesis.

# Template

A component has a template, which describes how the component is rendered on the page.

```
@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 {}
```

You can define the template inline, as shown above, or externally using templateUrl. In addition to the template, a component can define styles using the style and styleUrls properties.

```
@Component({
  selector: 'talk-cmp',
  template: `
    {{talk.title}} {{talk.speaker}}
    {{talk.rating | formatRating}}
    <watch-button [talk]="talk"></watch-button>
    <rate-button [talk]="talk"></rate-button>
  `,
  styles: [`
    watch-button {
      margin: 10px;
    }
  `]
})
class TalkCmp {}
```

By default the styles are encapsulated, so the margin defined above won't affect any other component using `watch-button`.

# Lifecycle

**Components have a well-defined lifecycle, which you can tap into.** `TalkCmp` does not subscribe to any lifecycle events, but some other components can. For instance, this component will be notified when its input properties change.

```
@Component({
  selector: 'cares-about-changes'
})
class CaresAboutChanges implements OnChanges {
  @Input() field1;
  @Input() field2;

  ngOnChanges(changes) { //.. }
}
```

# Providers

**A component can configure dependency injection by defining the list of providers the component and its children may inject.**

```
@Component({
  selector: 'conf-app',
  providers: [Logger]
})
class AppCmp { //... }

@Component({
  ...
})
class TalksCmp {
  constructor(logger:Logger) { //... }
}
```

In this example, we have the logger service declared in the app component, which makes them available in the whole application. The talks component injects the logger service. I will cover dependency injection in detail in Chapter 4. For now, just remember that components can configure dependency injection.

# Host Element

To turn an Angular component into something rendered in the DOM you have to associate an Angular component with a DOM element. We call such elements host elements.

A component can interact with its host DOM element in the following ways:

- It can listen to its events.
- It can update its properties.
- It can invoke methods on it.

The component, for instance, listens to the input event using `hostListeners`, trims the value, and then stores it in a field. Angular will sync up the stored value with the DOM.

```
@Directive({
  selector: '[trimmed-input]'
})
class TrimmedInput {
  @HostBinding() value: string;

  @HostListener("input", "$event.target.value")
  onChange(updatedValue: string) {
    this.value = updatedValue.trim();
  }
}
```

Note, I don't actually interact with the DOM directly. Angular aims to provide a higher-level API, so the native platform, the DOM, will just reflect the state of the Angular application. This is useful for a couple of reasons:

- It makes components easier to refactor.
- It allows unit testing most of the behavior of an application without touching the DOM. Such tests are easier to write and understand. In addition, they are significantly faster.
- It allows running Angular applications in a web worker, server, or other platforms where a native DOM isn't present.
- Sometimes you just need to interact with the DOM directly. Angular provides such APIs, but our hope is that you will rarely need to use them.

# Queries

In addition to access its host element, a component can interact with its children. There are two types of children a component can have: **content children** and **view children**. To understand the difference between them, let's look at the following example:

```
@Component({
  selector: 'tab',
  template: `...`
})
class TabCmp {}

@Component({
  selector: 'tabs',
  template: `
    Tabs:
    <div>
      <ng-content></ng-content>
    </div>
    <div>
      <button (click)="selectPrev()">Prev</button>
      <button (click)="selectNext()">Next</button>
    </div>
  `
})
class TabsCmp {}

@Component({
  template: `
```

```
    <tabs>
      <tab title="One"></tab>
      <tab title="Two"></tab>
      <tab title="Three"></tab>
    </tabs>
  `
})
class CmpUsingTabs {
}
```

The content children of the tabs component are the three tab components. The user of the tabs component provided those. The previous and next buttons are the view children of the tabs component. The author of the tabs component provided those. Components can query their children using the `ContentChild`, `ContentChildren`, `ViewChild`, and `ViewChildren` decorators.

Angular will set this list during the construction of the tabs component, and will keep it updated when content children get created, removed, or reordered. I will talk more about this in Chapter 5.

# Let's Recap

What I have listed constitutes a component.

- A component knows how to interact with its host element.
- A component knows how to interact with its content and view children.
- A component knows how to render itself.
- A component configures dependency injection.
- A component has a well-defined public API of input and output properties.

**All of these make components in Angular self-describing, so they contain all the information needed to instantiate them. And this is extremely important.**

This means that any component can be bootstrapped. It does not have to be special in any way. Moreover, any component can be loaded into a router outlet. As a result, you can write a component that can be bootstrapped as an application, loaded as a route, or used in some other component directly. This results in less API to learn. And it also makes components more reusable.

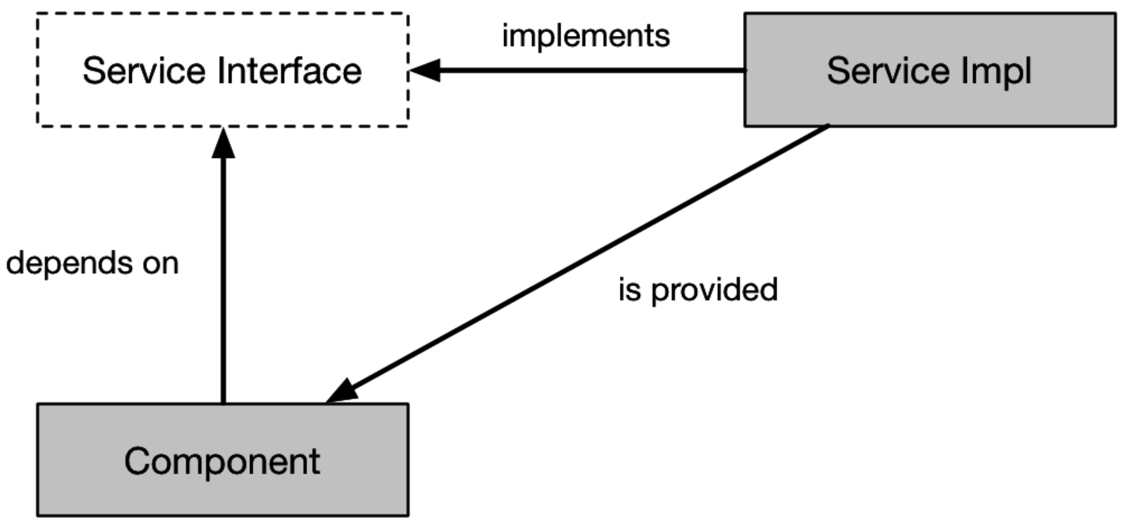
# What About Directives?

If you are familiar with Angular 1, you must be wondering "What happened to directives?".

Actually directives are still here in Angular. The component is just the most important type of a directive, but not the only one. A component is a directive with a template. But you can still write decorator-style directives, which do not have templates.

# Chapter 4: Dependency Injection

The idea behind dependency injection is very simple. If you have a component that depends on a service. You do not create that service yourself. Instead, you request one in the constructor, and the framework will provide you one. By doing so you can depend on interfaces rather than concrete types. This leads to more decoupled code, which enables testability, and other great things.




Angular comes with a dependency injection system. To see how it can be used, let's look at the following component, which renders a list of talks using the `for` directive:

```
@Component({
  selector: 'talks-cmp',
  template: `
    <h2>Talks:</h2>
    <talk *ngFor="let t of talks" [talk]="t"></talk>
  `
})
class TalksCmp {
  constructor() { //..get the data }
}
```

Let's mock up a simple service that will give us the data.

```
class TalksAppBackend {
  fetchTalks() {
    return [
      { name: 'Are we there yet?' },
      { name: 'The value of values' }
    ];
  }
}
```

How can you use this service? One approach is to create an instance of this service in our component.

```
class TalksCmp {
  constructor() {
    const backend = new TalksAppBackend();
    this.talks = backend.fetchTalks();
  }
}
```

This is fine for a demo app, but not good for real applications. In a real application `TalksAppBackend` won't just return an array of objects, it will make http requests to get the data. This means that the unit tests for this component will make real http requests—not a great idea. This problem is caused by the fact that you have coupled `TalksCmp` to `TalksAppBackend` and its new operator.

You can solve this problem by injecting an instance of `TalksAppBackend` into the constructor, so you can easily replace it in tests, like this:

```
class TalksCmp {
  constructor(backend:TalksAppBackend) {
    this.talks = backend.fetchTalks();
  }
}
```

This tells Angular that `TalksCmp` depend on `TalksAppBackend`. Now, you need to tell Angular how to create an instance of `TalksAppBackend`.

## Registering Providers

To do that you need to register a provider, and there are two places where you can do it. One is in the component decorator.

```
@Component({
  selector: 'talks-cmp',
  template: `
    <h2>Talks:</h2>
    <talk *ngFor="let t of talks" [talk]="t"></talk>
  `,
  providers: [TalksAppBackend]
})
class TalksCmp {
  constructor(backend:TalksAppBackend) {
    this.talks = backend.fetchTalks();
  }
}
```

And the other one is in the module decorator.

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

What is the difference and which one should you prefer?

Generally, I recommend to register providers at the module level when they do not depend on the DOM, components, or directives. And only UI-related providers that have to be scoped to a particular component should be registered at the component level. Since `TalksAppBackend` has nothing to do with the UI, register it at the module level.

# Injector Tree

Now you know that the dependency injection configuration has two parts:

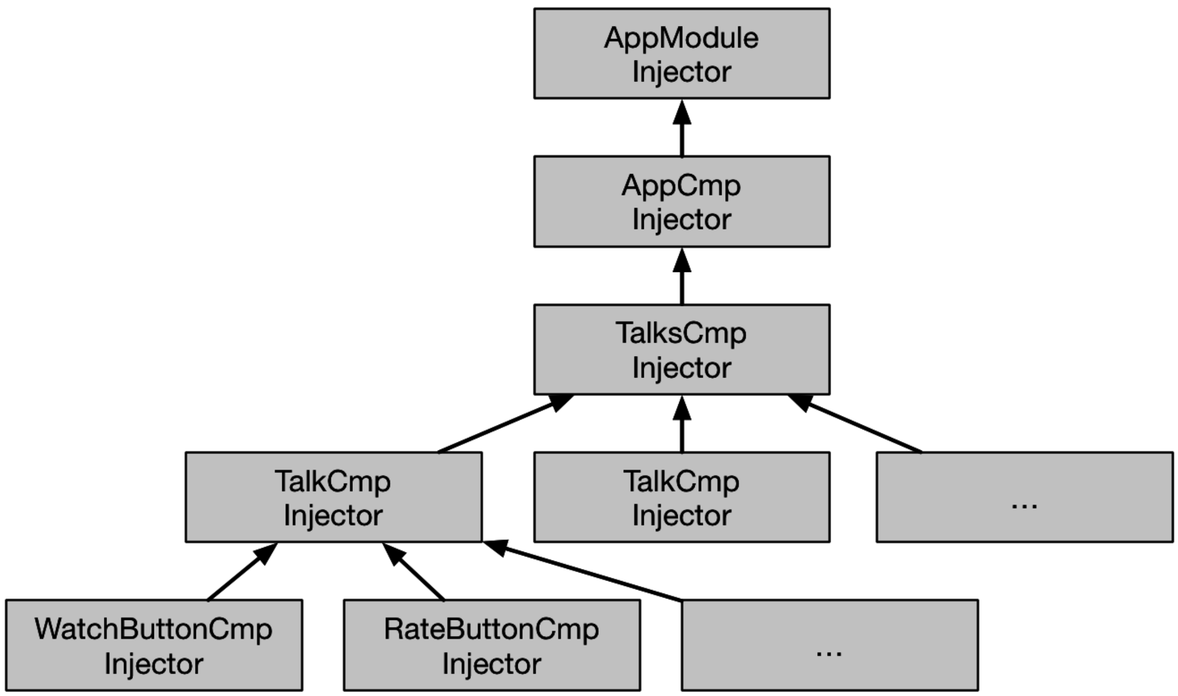
- **Registering providers**: How and where an object should be created.
- **Injecting dependencies**: What an object depends on.

And everything an object depends on (services, directives, and elements) is injected into its constructor. To make this work the framework builds a tree of injectors.

First, every DOM element with a component or a directive on it gets an injector. This injector contains the component instance, all the providers registered by the component, and a few “local” objects (e.g., the element).

Second, when bootstrapping an `NgModule`, Angular creates an injector using the module and the providers defined there.

So the injector tree of the application will look like this:




## Resolution

And this is how the dependency resolution algorithm works.

```
// this is pseudocode.
let inj = this;
while (inj) {
  if (inj.has(requestedDependency)) {
    return inj.get(requestedDependency);
  } else {
    inj = inj.parent;
  }
}
throw new NoProviderError(requestedDependency);
```

When resolving the backend dependency of `TalksCmp`, Angular will start with the injector of the talks component itself. Then, if it is unsuccessful, it will climb up to the injector of the app component, and, finally, will move up to the injector created from `AppModule`. That is why, for `TalksAppBackend` to be resolved, you need to register it at `TalkCmp`, `AppCmp`, or `AppModule`.

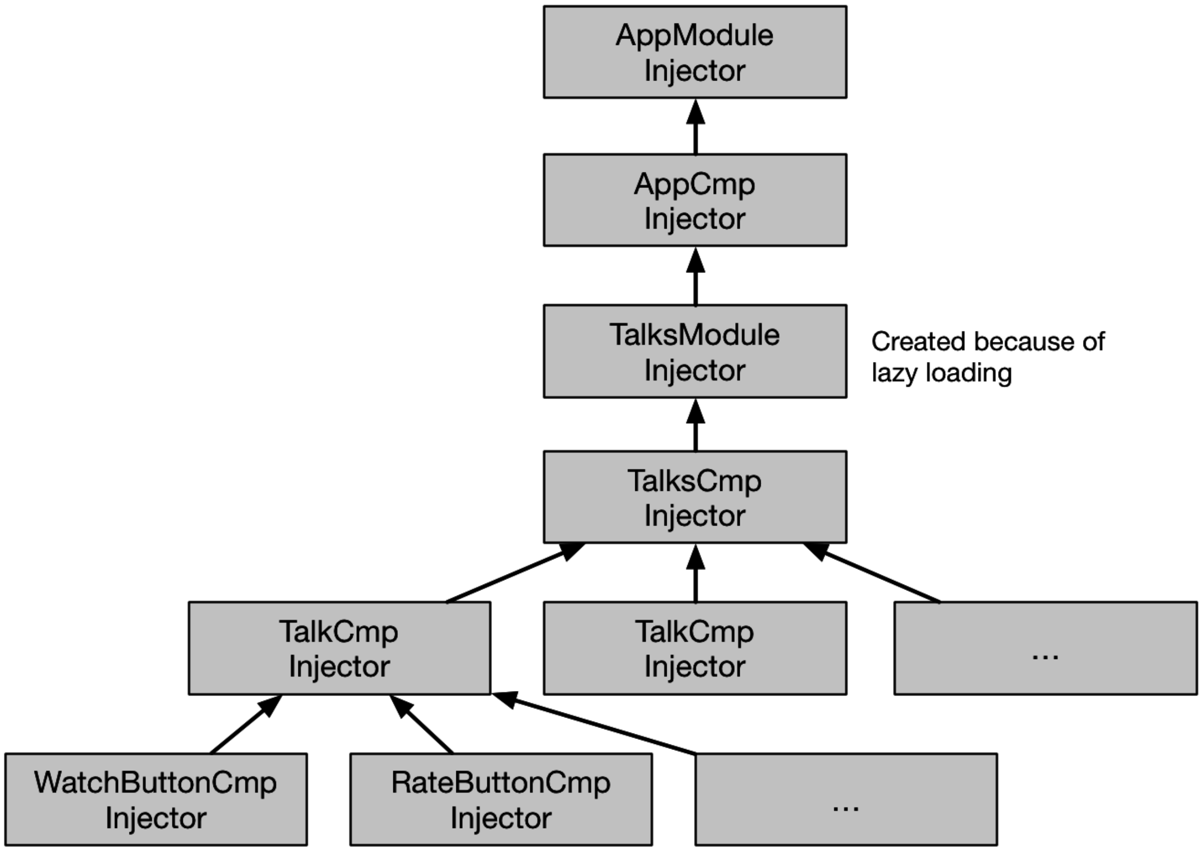
# Lazy Loading

The setup gets more complex once you start using lazy-loading.

Lazy-loading a module is akin to bootstrapping a module in that it creates a new injector out of the module and plugs it into the injector tree. To see it in action, let's update our application to load the talks module lazily.

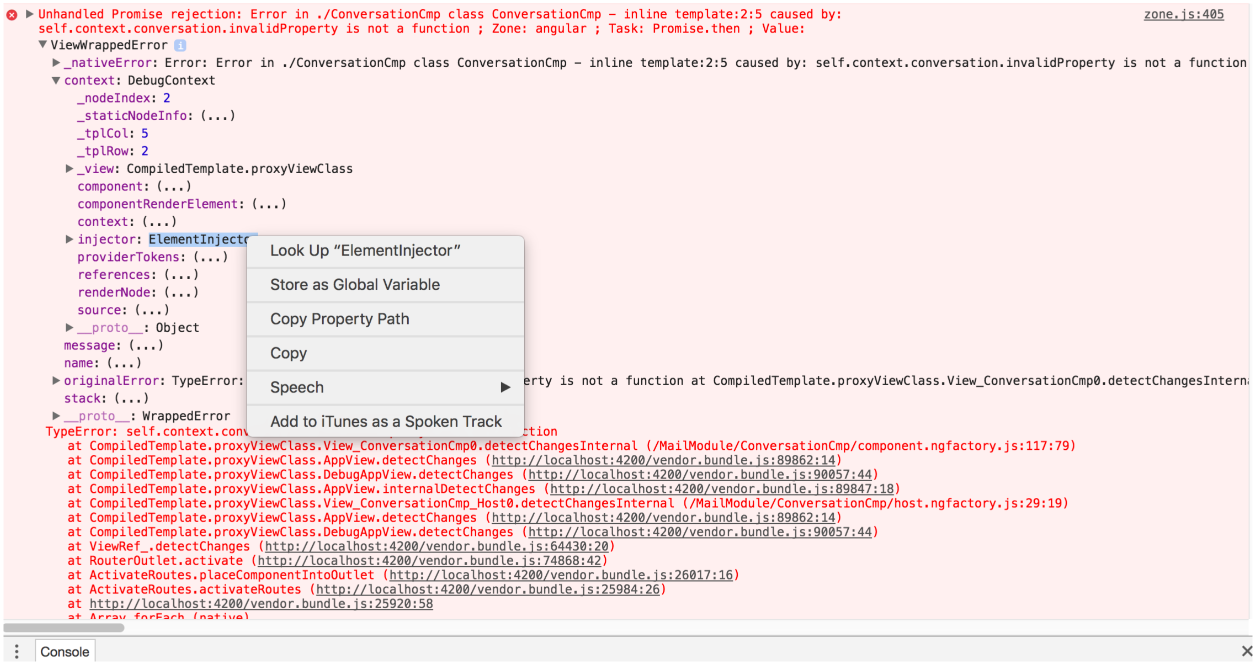
```
@NgModule({
  declarations: [AppCmp],
  providers: [RouterModule.forRoot([
    {path: 'talks', loadChildren: 'talks'}
  ])]
})
class AppModule {}

@NgModule({
  declarations: [FormattedRatingPipe, WatchButtonCmp, RateButtonCmp, TalkCmp, Ta\
lksCmp],
  entryComponents: [TalksCmp],
  providers: [TalksAppBackend, RouteModule.forChild([
    {path: '', component: TalksCmp}
  ])]
})
class TalksModule {}
```

With this change, the injector tree will look as follows:




## Getting Injector

You can use ngProbe to poke at an injector associated with an element on the page. You can also see an element's injector when an exception is thrown.

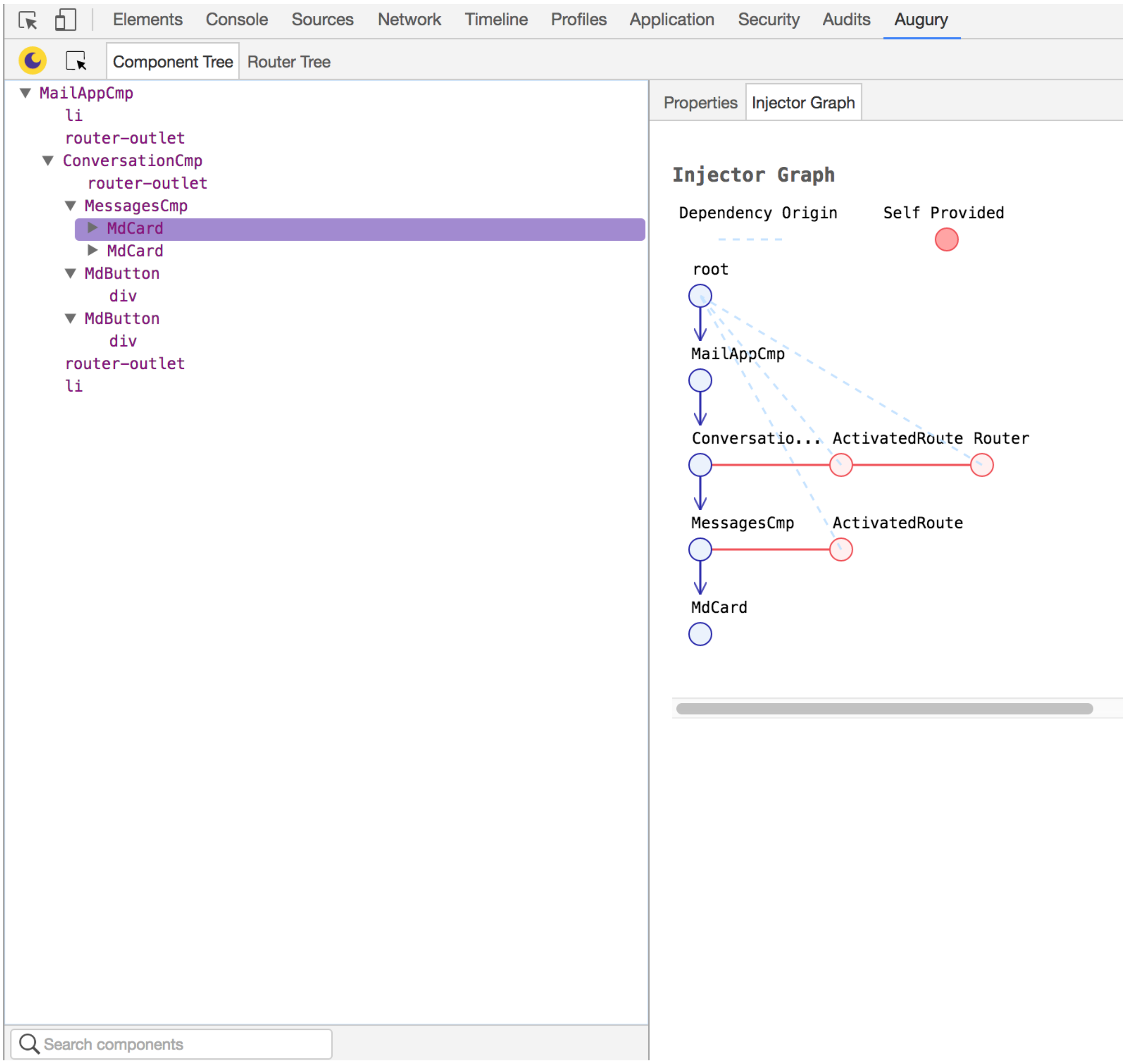



Right click on any of these objects to store them as a global variable, so you can interact with them in the console.

# Visualizing Injector Tree

If you more of a visual person, use the #Angular Augury[3] chrome extension to inspect the component and injector trees.

[3] https://augury.angular.io/




# Advanced Topics

## Controlling Visibility

You can be more specific where you want to get dependencies from. For instance, you can ask for another directive on the same element.

```
class CustomInputComponent {
  constructor(@Self() f: FormatterDirective) {}
}
```

Or you can ask for a directive in the same template, i.e., you can only inject an ancestor directive from the same HTML file.

```
class CustomInputComponent {
  constructor(@Host() f: CustomForm) {}
}
```

Finally, you can ask to skip the current element, which can be handy for decorating existing providers or building up tree-like structures.

```
class SomeComponent {
  constructor(@SKipSelf() ancestor: SomeComponent) {}
}
```

## Optional Dependencies

To mark a dependency as optional, use the `Optional` decorator.

```
class Login {
  constructor(@Optional() service: LoginService) {}
}
```

## More on Registering Providers

Passing a class into an array of providers is the same as using a provider with `useClass`, i.e., the two examples below are identical:

```
@NgModule({
  providers: [
    SomeClass
  ]
})
class MyModule {}
```

```
@NgModule({
  providers: [
    { provide: SomeClass, useClass: SomeClass}
  ]
})
class MyModule {}
```

When `useClass` does not suffice, you can configure providers with `useValue`, `useFactory`, and `useExisting`.

```
@NgModule({
  providers: [
    { provide: 'one', useValue: 1},
    { provide: 'sameOne', useExisting: 'one'},
    { provide: 'sum', useFactory: (one, sameOne) => one + sameOne, deps: ['one',\
 'sameOne']}
  ]
})
class MyModule {
  constructor(@Inject('sum') sum: number) {
    console.log("sum", sum);
  }
}
```

As you can see above, we can use the `@Inject` decorator to configure dependencies when the type parameter does not match the provided token.

## Aliasing

It's common for components and services to alias themselves.

```
@Component({
  selector: 'component-reexporting-itself',
  providers: [
    {provide: 'alias', useExisting: forwardRef(() => ComponentReexportingItself)}
  ]
})
class ComponentReexportingItself {}
```

Now we can use both `@Inject(ComponentReexportingItself)` and `@Inject('alias')` to inject this component.

## Overrides

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 one wins.

```
@NgModule({
  providers: [
    {provide: 'token', useValue: 'A'}
  ]
})
class ModuleA {}

@NgModule({
  providers: [
    {provide: 'token', useValue: 'B'}
  ]
})
class ModuleB {}

@NgModule({
  imports: [ModuleA, ModuleB]
})
class ModuleC {
  constructor(@Inject('token') t: string) {
    console.log(t); // will print 'B'
  }
}
```

The example above will print 'B'. If we change ModuleC to have its own 'token' provider, that one will be used, and the example will print 'C'.

```
@NgModule({
  imports: [ModuleA, ModuleB],
  providers: [
    {provide: 'token', useValue: 'C'}
  ]
})
class ModuleC {
  constructor(@Inject('token') t: string) {
    console.log(t); // will print 'C'
  }
}
```

## Let's Recap

- Dependency injection is a key component of Angular.
- You can configure dependency injection at the component or module level.
- Dependency injection allows us to depend on interfaces rather than concrete types.
- This results in more decoupled code.
- This improves testability.

# Chapter 5: Change Detection

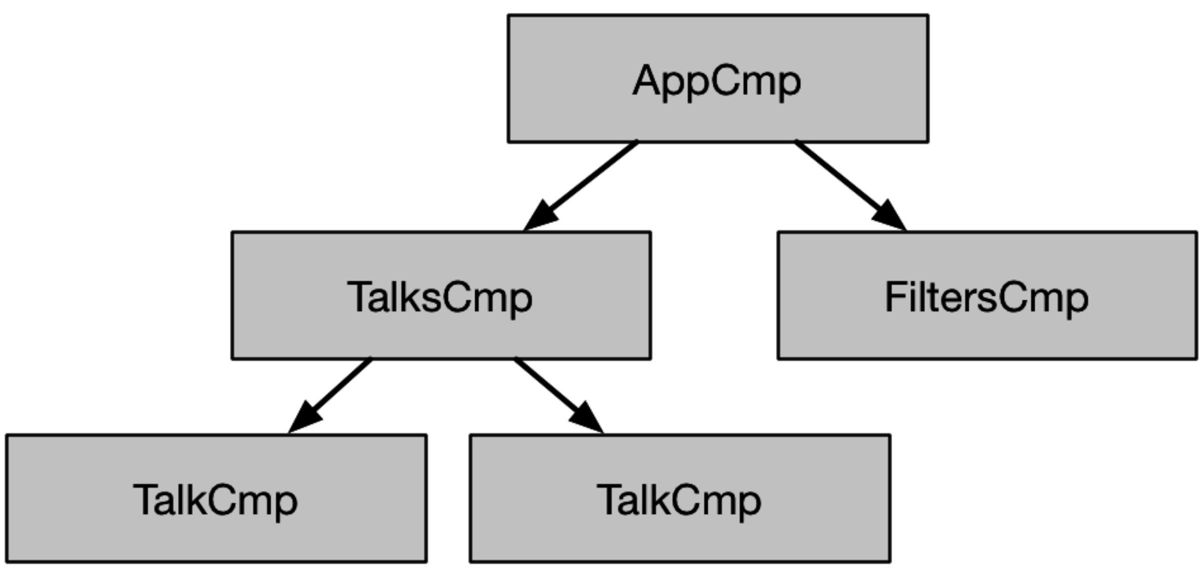
## Two Phases

**Angular separates updating the application model and reflecting the state of the model in the view into two distinct phases**. The developer is responsible for updating the application model. Angular via bindings, by means of change detection, is responsible for reflecting the state of the model in the view. The framework does it automatically on every VM turn.

**Event bindings**, which can be added using the () syntax, can be used to capture a browser event or component output to execute some function on a component or a directive. So they often trigger the first phase.

**Property bindings**, which can be added using the [] syntax, should be used only for reflecting the state of the model in the view.

As we have learned, an Angular application consists of nested components, so it will always have a component tree. Let's say for this app it looks as follows:




Next, define the application model that will store the state of our application.

```
{
  "filters": {"speakers": "Rich Hickey"},
  "talks": [
    {
      "id":898,
      "title": "Are we there yet",
      "speaker": "Rich Hickey",
      "yourRating":null,
      "rating": 9.1
    },
  ]
}
```

Now, imagine an event changing the model. Let's say I watched the talk "Are we there yet", I really liked it, and I decided to give it 9.9.

The code snippet below shows one way to do it. The `handleRate` function is called, via an event binding, when the user rates a talk.

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

  constructor(private app: App){}

  handleRate(newRating: number) {
    this.app.rateTalk(this.talk, newRating);
  }
}

@Component({
   selector: 'talks',
   template: `
     <h2>Talks:</h2>
     <talk *ngFor="let t of app.talks" [talk]="t"></talk>
```

```
  `
})
class TalksCmp {
  constructor(public app: App) {}
}

// contains the business logic
class App {
  talks: Talk[] = [];

  rateTalk(talk: Talk, rating: number){
    // model is immutable, so we have to build a new model object
    // constructing a new talk instance with an updated rating
    const updatedTalk = updateRecord(talk, {rating});

    // constructing a new collection instance
    this.talks = updateArray(this.talks, updatedTalk);
  }
}
```

In this example, we do not mutate the talk, and instead create a new array of new talks every time a change happens, which results in a few good properties. But it is worth noting that Angular doesn't require us to use immutable objects, and we could just as easily write something like `talk.rating = newRating`.

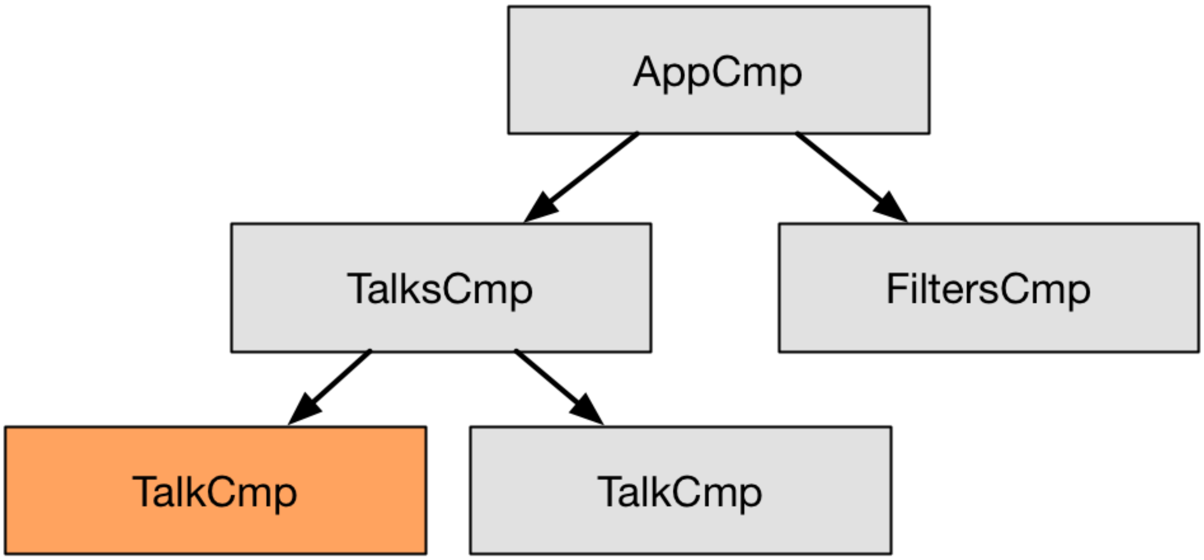
All right, after `rateTalk` executes, the updated model will look like this:

```
{
  "filters": {"speakers": "Rich Hickey"},
  "talks": [
    {
      "id":898,
      "title": "Are we there yet",
      "speaker": "Rich Hickey",
      "yourRating": 9.9,
      "rating": 9.1
    },
  ]
}
```

At this point nothing has changed in the view. Only the model has been updated.

Next, at the end of the VM turn, change detection kicks in to propagate changes in the view.

**First, change detection goes through every component in the component tree to check if the model it depends on changed. And if it did, it will update the component**. In this example, the first talk component gets updated:




Then, the framework updates the DOM. In our example, the rate button gets disabled because we cannot rate the same talk twice.

Note, the framework has used **change detection** and **property bindings** to execute this phase.

In our example we are using shared state and immutable data. But even if we used local state and mutable data, it would not change the property that the application model update and the view state propagation are separated.

## Why?

Now, when we have understood how we had separated the two phases, let's talk about why we did it.

### Predictability

First, using change detection only for updating the view state limits the number of places where the application model can be changed. In this example it can happen only in the `rateTalk` function. A watcher cannot "automagically" update it. This makes ensuring invariants easier, which makes code easier to troubleshoot and refactor.

Second, it helps us understand the view state propagation. Consider what we can say about the talk component just by looking at it in isolation. Since we use immutable data, we know that as long as we do not do talk= in the Talk component, the only way to change what the component displays is by updating the input. These are strong guarantees that allow us to think about this component in isolation.

Finally, by explicitly stating what the application and the framework are responsible for, we can set different constraints on each part. For instance, it is natural to have cycles in the application model. So the framework should support it. On the other hand, html forces components to form a tree structure. We can take advantage of this and make the system more predictable.

**Starting with Angular 2.x it gets easier to think about components because the framework limits the number of ways it can modify the components, and those modifications are predictable**.

### Performance

The major benefit of the separation is that it allows us to constrain the view state propagation. This makes the system more predictable, but it also makes it a lot more performant. For example, the fact that the change detection graph in Angular can be modeled as a tree allowed us to get rid of digest TTL (multiple digest runs until no changes occur). Now the system gets stable after a single pass.

## How Does Angular Enforce It?

What happens if I try to break the separation? What if I try to change the application model inside a setter that is invoked by the change detection system?

Angular tries to make sure that the setter we define for our component only updates the view state of this component or its children, and not the application model. To do that Angular will check all bindings twice in the developer mode. First time to propagate changes, and second time to make sure there are no changes. If it finds a change during the second pass, it means that one of our setters updated the application model, the framework will throw an exception, pointing at the place where the violation happened.

## Content and View Children

Earlier I said “change detection goes through every component in the component tree to check if the model it depends on changed” without saying much about how the framework does it. In what order does it do it? Understanding this is crucial, and that’s what I’m going to cover in this section.

There are two types of children a component can have: content children and view children. To understand the difference between them, let’s look at the following example:

```
@Component({
  selector: 'tab',
  template: `...`
})
class TabCmp {
}

@Component({
  selector: 'tabs',
  template: `
    Number of tabs: {{tabs.length}}
    <div>
      <ng-content></ng-content>
    </div>
    <div>
      <button (click)="selectPrev()">Prev</button>
      <button (click)="selectNext()">Next</button>
    </div>
  `
})
class TabsCmp {
  @ContentChildren(TabCmp) tabs: QueryList<Tab>;
}

@Component({
  template: `
    <tabs>
      <tab title="One"></tab>
      <tab title="Two"></tab>
      <tab title="Three"></tab>
    </tabs>
  `
})
class CmpUsingTabs {
}
```

The content children of the tabs component are the three tab components. The user of the tabs component provided those. The previous and next buttons are the view children of the tabs component. The author of the tabs component provided those.

This is the order in which Angular will check the bindings:

- It will check the bindings of the tabs component first, of which there are none.

- It will check the three tab component, the content children of the tabs component.
- It will check the template of the tabs component.

## ChangeDetectionStrategy.OnPush

If we use mutable objects that are shared among multiple components, Angular cannot know about when those components can be affected. A component can affect any other components in the system. That is why, by default, Angular does not make any assumptions about what a component depends upon. So it has be conservative and check every template of every component on every browser event. Since the framework has to do it for every component, it might become a performance problem even though the change detection in the new versions of Angular got way faster.

If our model application state uses immutable objects, like in the example above, we can tell a lot more about when the talk component can change. The component can change if and only if any of its inputs changes. And we can communicate it to Angular by setting the change detection strategy to OnPush.

```
Component({
  selector: 'talk',
  template: `
    {{talk.title}} {{talk.speaker}}
    {{talk.rating | formatRating}}
    <watch-button [talk]="talk"></watch-button>
    <rate-button [talk]="talk" (rate)="handleRate($event)"></rate-button>
  `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
class TalkCmp {
  //...
}
```

Using this change-detection strategy restricts when Angular has to check for updates from "any time something might change" to "only when this component's inputs have changed". As a result, the framework can be a lot more efficient about detecting changes in TalkCmp. If no inputs change, no need to check the component's template. In addition to depending on immutable inputs OnPush components can also have local mutable state.

## Let's Recap

- Angular separates updating the application model and updating the view.
- Event bindings are used to update the application model.

- Change detection uses property bindings to update the view. Updating the view is unidirectional and top-down. This makes the system more predictable and performant.
- We make the system more efficient by using the OnPush change detection strategy for the components that depend on immutable input and only have local mutable state.

# Chapter 6: Testing

One of the design goals of Angular is to make testing easy. That's why the framework relies on dependency injection, separates the user code from the framework code, and comes with a set of tools for writing and running tests. In this chapter I will look at four ways to test Angular components: isolated tests, shallow tests, integration tests, and protractor tests.

## Isolated Tests

It is often useful to test complex components without rendering them. To see how it can be done, let's write a test for the following component:

```
@Component({
  selector: 'filters-cmp',
  templateUrl: './filters.component.html',
  styleUrls: ['./filters.component.css']
})
export class FiltersCmp {
  @Output() change = new EventEmitter();

  filters = new FormGroup({
    speaker: new FormControl(),
    title: new FormControl(),
    highRating: new FormControl(false),
  });

  constructor(@Inject('createFiltersObject') createFilters: Function) {
    this.filters.valueChanges.debounceTime(200).subscribe((value) => {
      this.change.next(createFilters(value));
    });
  }
}
```

filters.component.html:

```
<div [formGroup]="filters">
  <md-input-container>
    <input md-input formControlName="title" placeholder="Title">
  </md-input-container>

  <md-input-container>
    <input md-input formControlName="speaker" placeholder="Speaker">
  </md-input-container>

  <md-checkbox formControlName="highRating">
    High Rating
  </md-checkbox>
</div>
```

There a few things in this example worth noting:

- We are using reactive forms in the template of this component. This require us to manually create a form object in the component class, which has a nice consequence: we can test input handling without rendering the template.
- We listen to all the form changes, debounce them using the RxJS debounceTime operator, and then emit a change event.
- Finally, we inject a function to create a filters object out of the form.

Now, let's look at the test.

```
describe('FiltersCmp', () => {
  describe("when filters change", () => {
    it('should fire a change event after 200 millis', fakeAsync(() => {
      const component = new FiltersCmp((v) => v);

      const events = [];
      component.change.subscribe(v => events.push(v));

      component.filters.controls['title'].setValue('N');
      setTimeout(() => { component.filters.controls['title'].setValue('Ne'); }, \
150);
      setTimeout(() => { component.filters.controls['title'].setValue('New'); },\
 200);

      expect(events).toEqual([]);
```

```
      tick(1000);

      // only one item because of debouncing
      expect(events).toEqual([
        {title: 'New', speaker: null, highRating: false}
      ]);

      component.filters.controls['title'].setValue('New!');

      tick(1000);

      expect(events).toEqual([
        {title: 'New', speaker: null, highRating: false},
        {title: 'New!', speaker: null, highRating: false}
      ]);
    }));
  });
});
```

As you can see, testing Angular components in isolation is no different from testing any other JavaScript object. We do not use any Angular UI-specific utilities. We, however, use fakeAsync. This is a utility provided by `zone.js` and using it we can control time, which is handy for testing the debouncing. Also, this test does not exercise the template of this component. The template might as well be empty–the test will still pass.

## Shallow Testing

Testing component classes without rendering their templates works in certain scenarios, but not in all of them. Sometimes we can write a meaningful test only if we render a component's template. We can do that and still keep the test isolated. We just need to render the template without rendering the component's children. This is what is colloquially known as shallow testing.

Let's see this approach in action.

```
@Component({
  selector: 'talks-cmp',
  template: '<talk-cmp *ngFor="let t of talks" [talk]="t"></talk-cmp>'
})
export class TalksCmp {
  @Input() talks: Talk[];
}
```

This component simply renders a collection of `TalkCmp`

Now let's look at its test.

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import { DebugElement, NO_ERRORS_SCHEMA } from '@angular/core';

import { TalksCmp } from './talks.component';

describe('TalksCmp', () => {
  let component: TalksCmp;
  let fixture: ComponentFixture<TalksCmp>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [TalksCmp],
      schemas: [NO_ERRORS_SCHEMA]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(TalksCmp);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should render a list of talks', () => {
    component.talks = <any>[
      { title: 'Are we there yet?' },
      { title: 'The Value of Values' }
    ];
    fixture.detectChanges();

    const s = fixture.debugElement.nativeElement;
    const ts = s.querySelectorAll("talk-cmp");

    expect(ts.length).toEqual(2);
    expect(ts[0].talk.title).toEqual('Are we there yet?');
    expect(ts[1].talk.title).toEqual('The Value of Values');
  });
});
```

First, look at how we configured our testing module. We only declared TalksCmp, nothing else. This means that all the elements in the template will be treated as simple DOM nodes, and only common directives (e.g., ngIf and ngFor) will be applied. This is exactly what we want. Second, passing NO_ERRORS_SCHEMA tells the compiler not to error on unknown elements and attributes, which is what we need for shallow tests. The result will create a list of `talk-cmp` DOM elements, which we inspected in the test. No instances of `TalkCmp` were created.

# Integration Testing

We can also write an integration test that will exercise the whole application.

```
import {TestBed, async, ComponentFixture, inject} from '@angular/core/testing';
import {AppCmp} from './app.component';
import {AppModule} from './app.module';
import {App} from "./app";

describe('AppCmp', () => {
  let component: AppCmp;
  let fixture: ComponentFixture<AppCmp>;
  let el: Element;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      imports: [AppModule]
    });
    TestBed.compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(AppCmp);
    component = fixture.componentInstance;
    fixture.detectChanges();
    el = fixture.debugElement.nativeElement;
  });

  it('should filter talks by title', async(inject([App], (app: App) => {
    app.model.talks = [
      {
        "id": 1,
        "title": "Are we there yet?",
        "speaker": "Rich Hickey",
```

```
        "yourRating": null,
        "rating": 9.0
      },
      {
        "id": 2,
        "title": "The Value of Values",
        "speaker": "Rich Hickey",
        "yourRating": null,
        "rating": 8.0
      },
    ];
    fixture.detectChanges();

    expect(el.innerHTML).toContain("Are we there yet?");
    expect(el.innerHTML).toContain("The Value of Values");

    component.handleFiltersChange({
      title: 'we',
      speaker: null,
      minRating: 0
    });
    fixture.detectChanges();

    expect(el.innerHTML).toContain("Are we there yet?");
    expect(el.innerHTML).not.toContain("The Value of Values");
  })));
});
```

Note here we are importing `AppModule`, which means that Angular will create all the registered provides and will compile all the registered components. The test itself is self explanatory.

Even though both the shallow and integration tests render components, these tests are very different in nature. In the shallow test we mock up every single dependency of a component, and we do not render any of the component's children. The goal is to exercise one slice of the component tree in isolation. In the integration tests we mock up only platform dependencies (e.g, location), and we use production code for the rest. Shallow tests are isolated, and, as a result, can be used to drive the design of our components. Integration tests are only used to check the correctness.

## Protractor Tests

Finally, we can always write a protractor test exercising the whole application.

```
import {browser, element, by} from 'protractor';

export class TalksAppPage {
  navigateTo() {
    return browser.get('/');
  }

  getTitleInput() {
    return element(by.css('input[formcontrolname=title]'));
  }

  getTalks() {
    return element.all(by.css('talk-cmp'));
  }

  getTalkText(index: number) {
    return this.getTalks().get(index).getText();
  }
}

describe('e2e tests', function() {
  let page: TalksAppPage;

  beforeEach(() => {
    page = new TalksAppPage();
  });

  it('should filter talks by title', () => {
    page.navigateTo();

    const title = page.getTitleInput();
    title.sendKeys("Are we there");

    expect(page.getTalks().count()).toEqual(1);
    expect(page.getTalkText(0)).toContain("Are we there yet?");
  });
});
```

First, we created a page object, which is a good practice for making tests more domain-centric, so they talk more about user stories and not the DOM. Second, we wrote a protractor test verifying that filtering by title works.

Both protractor tests and integration tests (as defined above), solve the same problem–they verify correctness, i.e., they verify that a particular use case have been implemented. Which ones should we use? I tend to test most of the behavior using integration tests, and use protractor only for a few smoke tests, but it is highly dependent on the culture of the team.

## Let's Recap

In this chapter we looked at four ways to test Angular components: isolated tests, shallow tests, integration tests, and protractor tests. Each of them have their time and place: isolated tests are a great way to test drive your components and test complex logic. Shallow tests are isolated tests on steroids, and they should be used when writing a meaningful test requires to render a component's template. Finally, integration and protractor tests verify that a group of components and services (i.e., the application) work together.