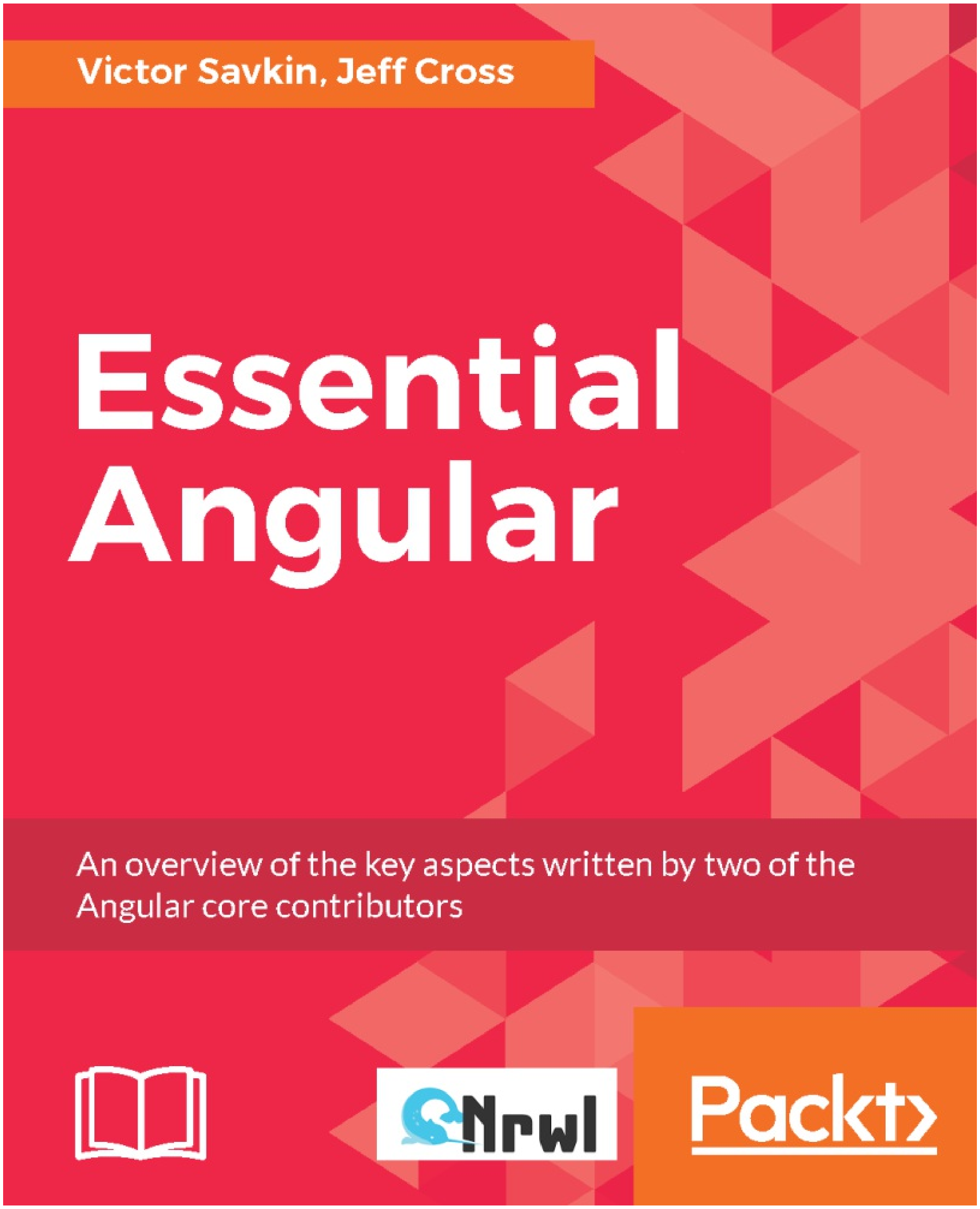
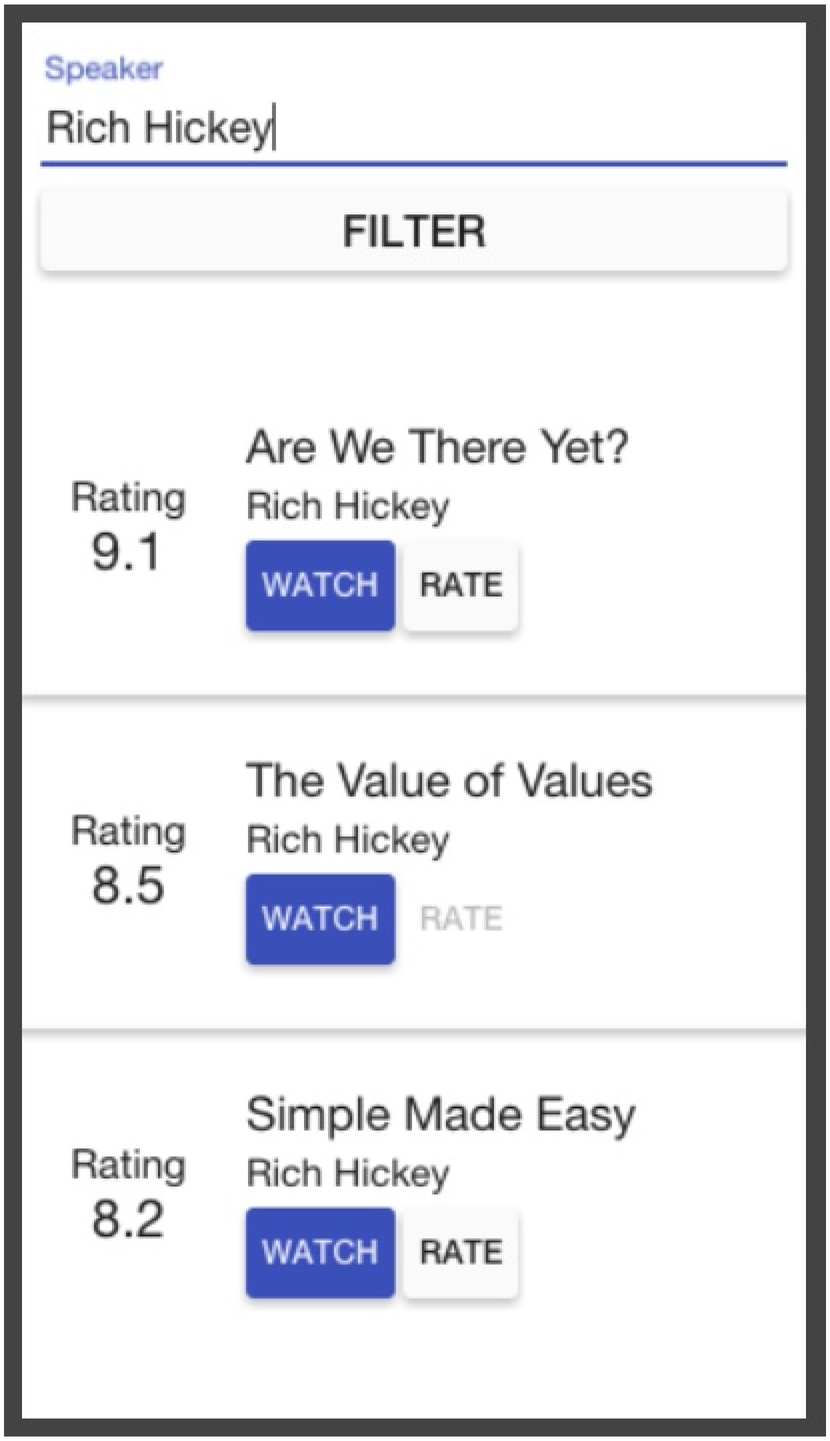

Victor Savkin, Jeff Cross
Essential Angular
An overview of the key aspects written by two of the Angular core contributors
Nrwl
Packt

Contents

Chapter 1. Example

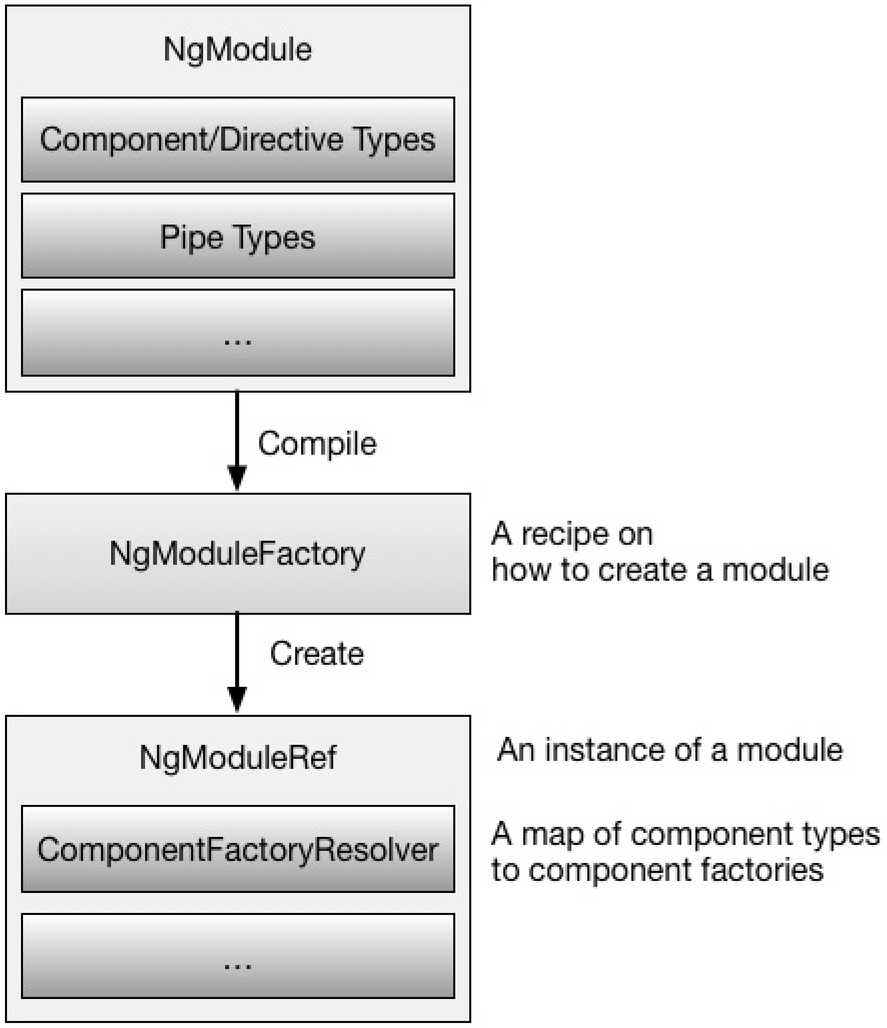
For most of the examples in this book we will use the same application. This application is a list of tech talks that you can filter, watch, and rate.



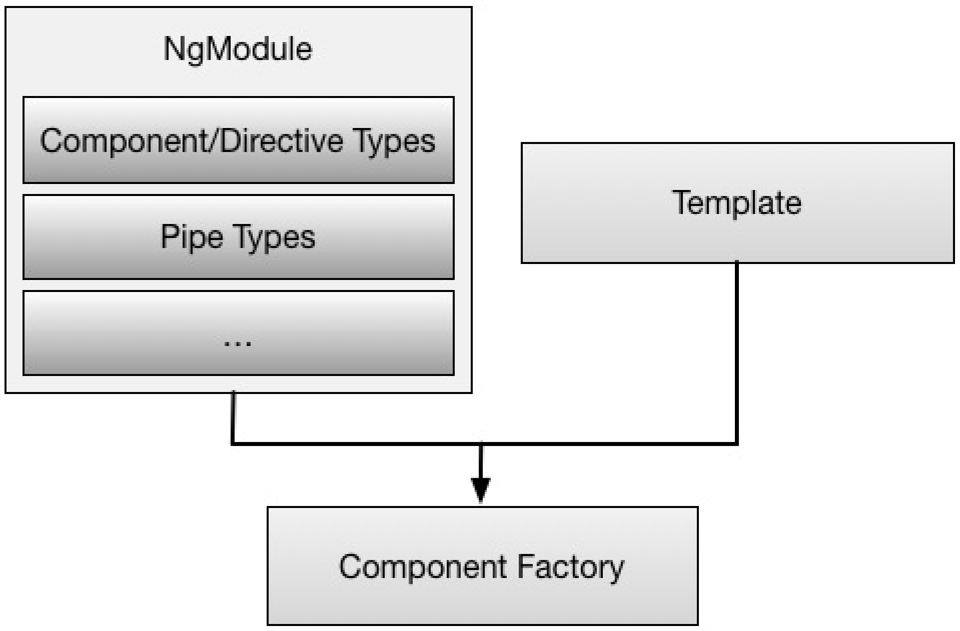
You can find the source code of the application here https://github.com/vsavkin/essential-angular-book-app.

Chapter 2. Compilation

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



A `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 produce a component factory. The component factory is a JavaScript class the framework can use to stamp out components.


NgModule
Component/Directive Types
Pipe Types
...
Template
Component Factory

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 have 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 (for example, WebPack and 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 the 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()...
```

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-butto
  `
})
class TalkCmp {
  //...
}
```

The `window.hide` property will not be defined. So the compilation will fail to point 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 preceding example.

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 3. 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, TalksCmp],
  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, that is, it tells Angular how these components should be compiled. As with ES, a component can...

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 components of `TalksModule`. 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...

Providers

I'll cover providers and dependency injection in Chapter 5, *Dependency Injection*. 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, that is, 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 here:

```
@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-browse
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 earlier NgModules are not just the units of compilation, they are also the units of distribution. That is why we bootstrap a 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: Injecto
    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
    });
  }
}
```

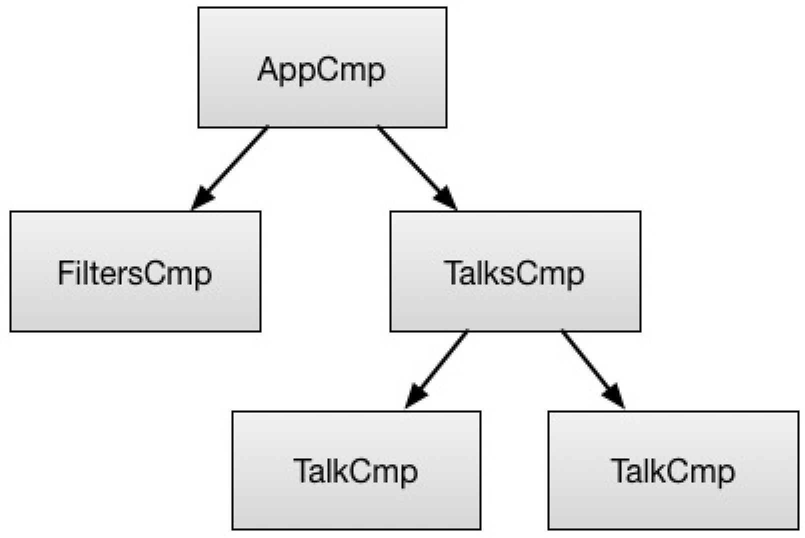
The loader compiles the modules if the application is running in the JIT mode, and does not in the AOT mode. The default loader `@angular/core` ships with uses SystemJS, but, as most things in Angular, you can provide your own.

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 4. 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>
  `
})
```

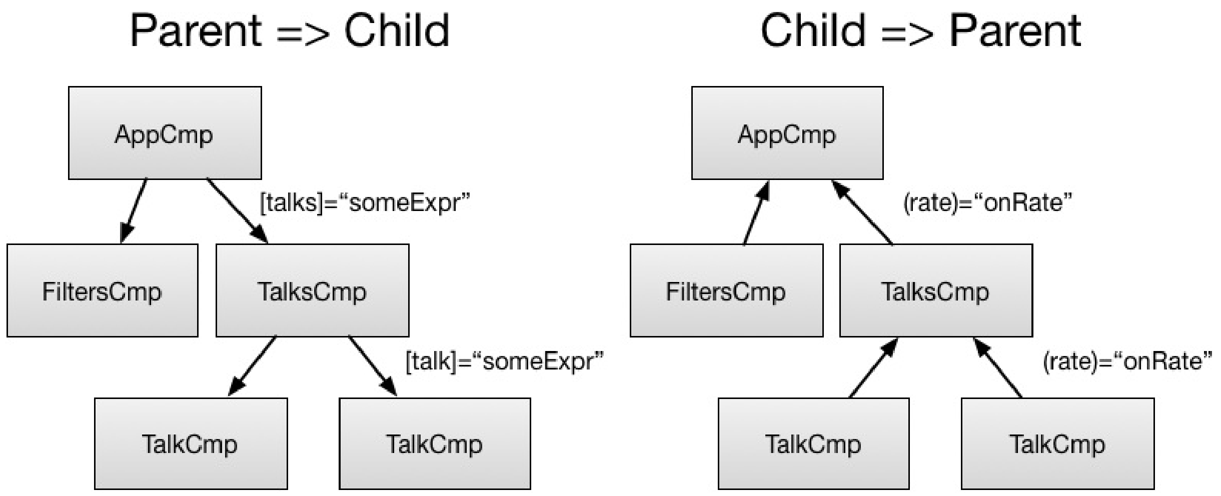
```
class TalkCmp {
  @Input() talk: Talk;
  @Output() rate: EventEmitter;
  //...
}
```

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-
```

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 in the preceding code, or externally using `templateUrl`. In addition to the template, a component can define styles using the `styles` 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 in the preceding code won't affect any other component using `watch-button`.

Life cycle

Components have a well-defined life cycle, which you can tap into. `TalkCmp` does not subscribe to any life cycle 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 5, *Dependency Injection*. 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,...

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 ></tab>
      <tab title="Two"></tab>
      <tab ></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...

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 5. Templates

In modern web development, there are two main techniques for describing what components render: using JavaScript and using templates. In this chapter I will talk about why Angular uses templates, and how its templates work.

Why templates?

Before we jump into how, let's talk about why.

Using templates is aligned with the rule of least power, which is a useful design principle.

> *the less powerful the language, the more you can do with the data stored in that language. [...] I chose HTML not to be a programming language because I wanted different programs to do different things with it: present it differently, extract tables of contents, index it, and so on.*

–Tim Berners-Lee, on the Principle of Least Power

Note

For the full article refer https://www.w3.org/DesignIssues/Principles.html#PLP

Because templates are pure HTML, and hence are constrained, tools and developers can make smart assumptions about what components are and how they behave, and Angular can do a lot of interesting things that would have not been possible if components used JavaScript instead. Let's look at some of those.

Swapping implementations

Performance is tricky: often it is hard to know ahead of time how well a particular technique will work. In addition, what is performant right now may not be...

Angular templates

Templates *can* be analyzable, transformable, and declarative in a way that JavaScript *fundamentally* cannot be. When designing Angular we put a lot of effort to make sure the template language has these properties. Let's look at what we ended up with.

Property and event bindings

Input and output properties are the public API of a directive. Data flows into a directive via its inputs and flows out via its outputs. We can update input properties using property bindings. And we can subscribe to output properties using event bindings.

Say we have a component that renders a button that allows rating a conference talk. We could use this component in our template as follows:

```
<rate-button[talk]="myTalk"(rate)="handleRate($event)"></rate-
```

This tells Angular that whenever `myTalk` changes, Angular needs to automatically update the `rate-button` component by calling the setter. This also tells Angular that if an event called `rate` is fired, it should invoke `handleRate`.

Now, let's look at the `RateButtonCmp` class:

```
@Component({
  selector: 'rate-button',
  templateUrl: './rate-button.component.html',
  styleUrls:...
```

Let's recap

That's how Angular templates work. Let's see if they are analyzable and transformable, and if they provide the benefits I talked about at the beginning of this chapter.

Contrasting with Angular 1, there is a lot more the framework can tell about the template statically.

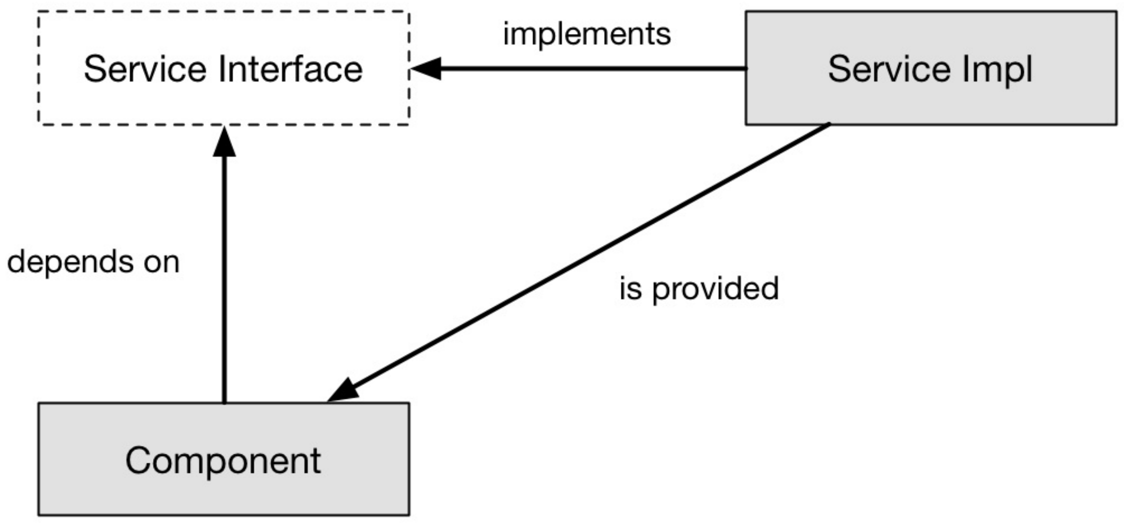
For example, regardless of what the component element is, Angular knows that `name` in `[property1]="name"` is a field read and `name` in `property2="name"` is a string literal. It also knows that the `name` property cannot be updated by the component: property bindings update from the parent to the child. Similarly, Angular can tell what variables are defined in the template by just looking at `*ngFor`. Finally, it can tell apart the static and dynamic parts in the template.

All of these allow us to reliably introspect and transform templates at compile time. That's a powerful feature, that, for instance, the Angular i18n support is built upon.

Having designed templates in this way, we also solved one of the biggest Angular 1 problems: IDE support. When using Angular 1, editors and IDEs had to do a lot of guessing to provide completion, refactoring, and navigation....

Chapter 6. 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()...
```

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, TalksCmp],
  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:

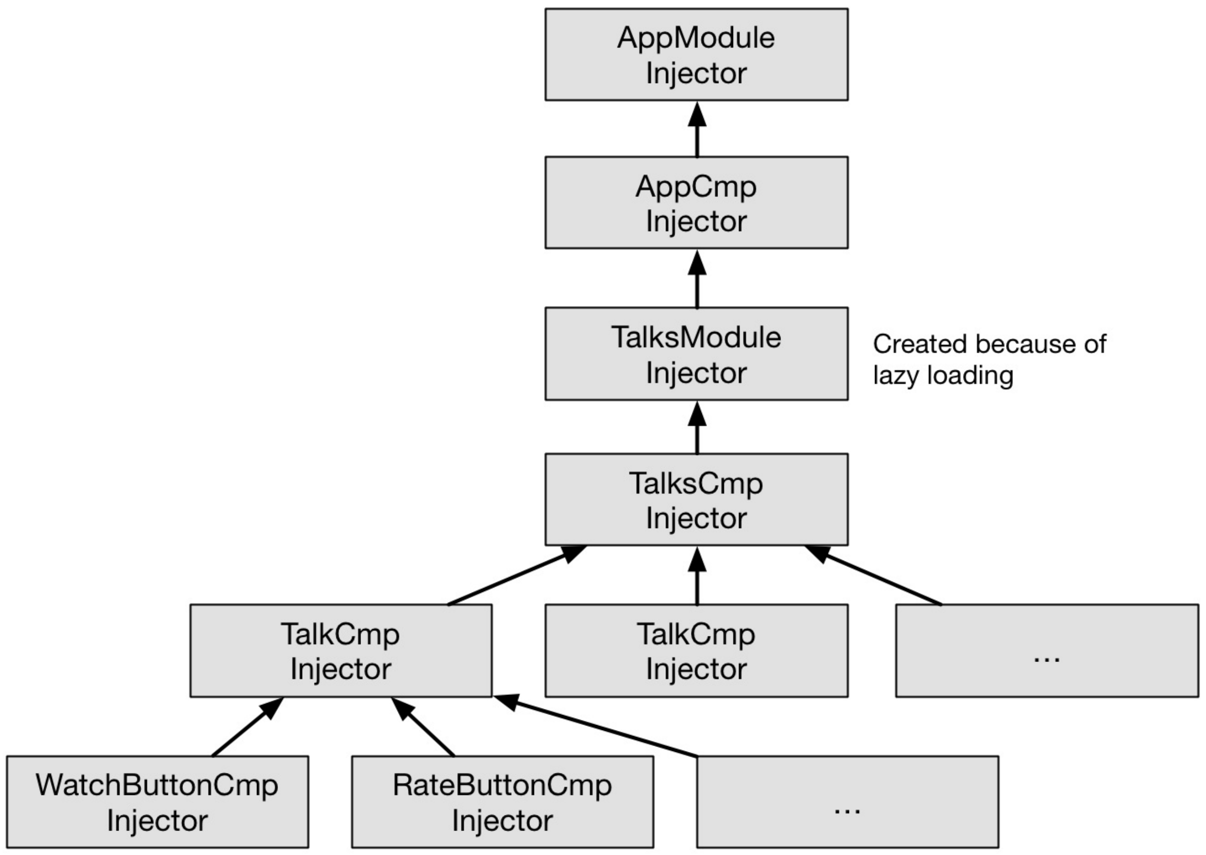
- **Registering providers**: How and where an object should be created.
- **Injecting dependencies**: What an object depends on.

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 (for example, 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:


AppModule Injector
AppCmp Injector
TalksModule Injector
Created because of lazy loading
TalksCmp Injector
TalkCmp Injector
TalkCmp Injector
...
WatchButtonCmp Injector
RateButtonCmp Injector
...

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`.

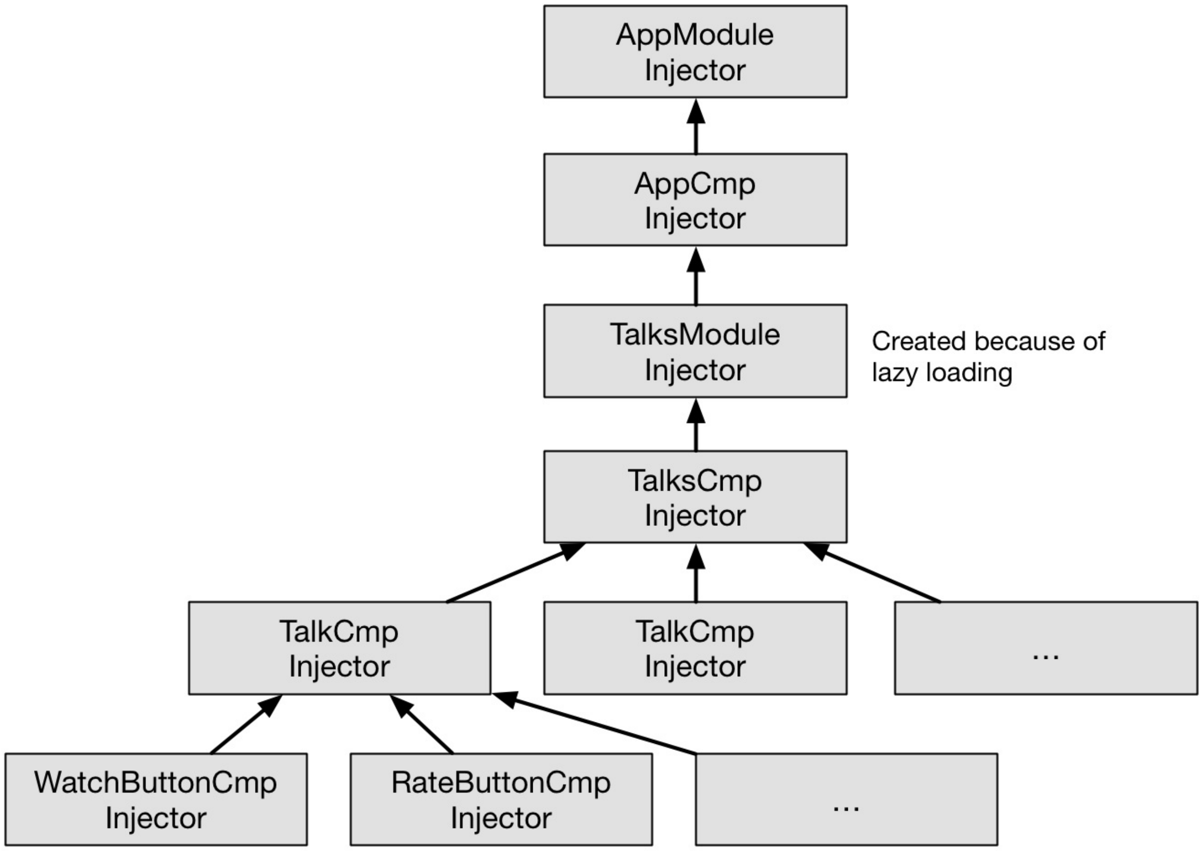
Lazy loading

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.

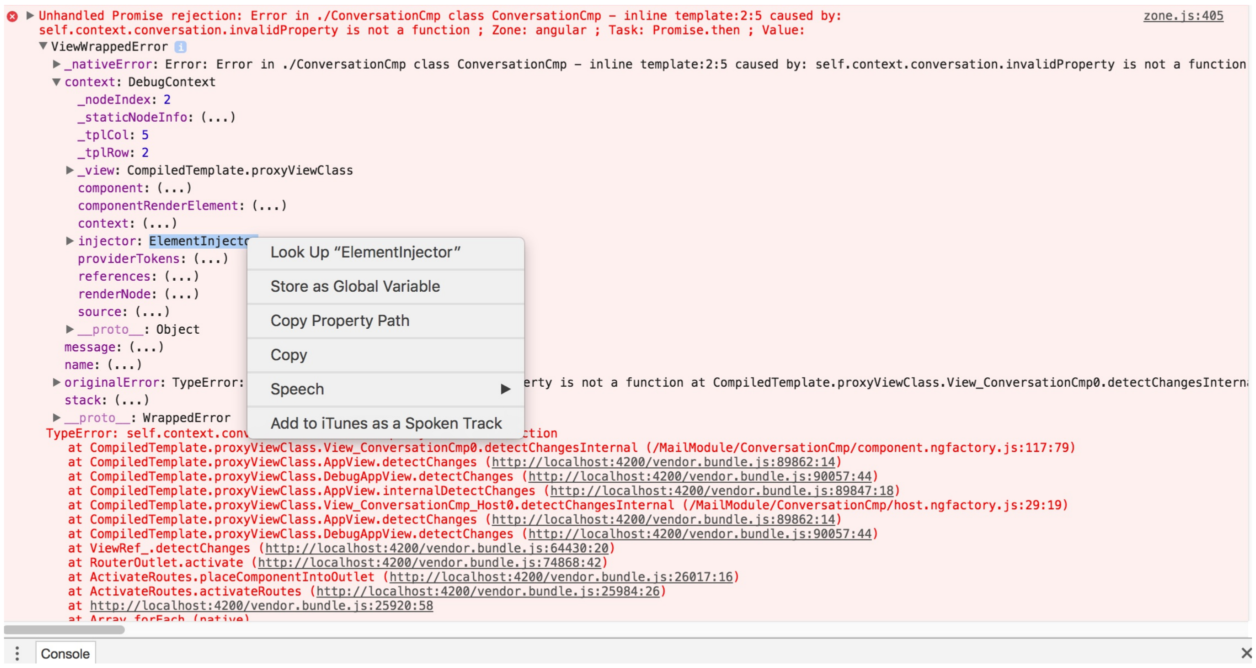
```
@NgModule({
  declarations: [AppCmp],
  providers: [RouterModule.forRoot([
    {path: 'talks', loadChildren: 'talks'}
  ])]
})
class AppModule {}

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

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


AppModule Injector
AppCmp Injector
TalksModule Injector
Created because of lazy loading
TalksCmp Injector
TalkCmp Injector
TalkCmp Injector
...
WatchButtonCmp Injector
RateButtonCmp Injector
...

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.

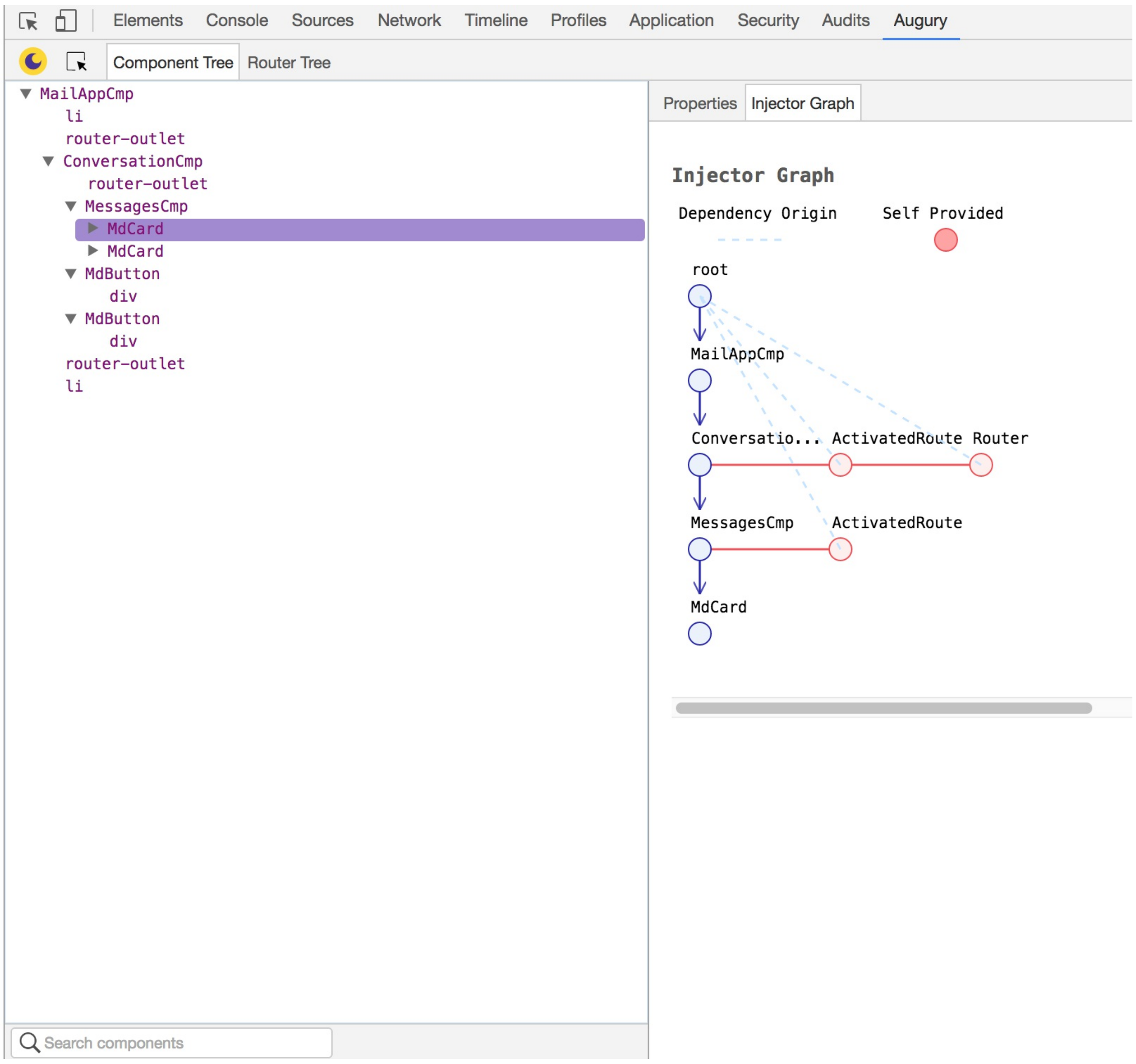


```
Unhandled Promise rejection: Error in ./ConversationCmp class ConversationCmp - inline template:2:5 caused by:        zone.js:405
self.context.conversation.invalidProperty is not a function ; Zone: angular ; Task: Promise.then ; Value:
▼ViewWrappedError
  ▶_nativeError: Error: Error in ./ConversationCmp class ConversationCmp - inline template:2:5 caused by: self.context.conversation.invalidProperty is not a function
  ▼context: DebugContext
     _nodeIndex: 2
     _staticNodeInfo: (...)
     _tplCol: 5
     _tplRow: 2
    ▶_view: CompiledTemplate.proxyViewClass
     component: (...)
     componentRenderElement: (...)
     context: (...)
    ▶injector: ElementInjecto
     providerTokens: (...)
     references: (...)
     renderNode: (...)
     source: (...)
    ▶__proto__: Object
   message: (...)
   name: (...)
  ▶originalError: TypeError:                                erty is not a function at CompiledTemplate.proxyViewClass.View_ConversationCmp0.detectChangesIntern
   stack: (...)
  ▶__proto__: WrappedError
 TypeError: self.context.conv                               ction
    at CompiledTemplate.proxyViewClass.View_ConversationCmp0.detectChangesInternal (/MailModule/ConversationCmp/component.ngfactory.js:117:79)
    at CompiledTemplate.proxyViewClass.AppView.detectChanges (http://localhost:4200/vendor.bundle.js:89862:14)
    at CompiledTemplate.proxyViewClass.DebugAppView.detectChanges (http://localhost:4200/vendor.bundle.js:90057:44)
    at CompiledTemplate.proxyViewClass.AppView.internalDetectChanges (http://localhost:4200/vendor.bundle.js:89847:18)
    at CompiledTemplate.proxyViewClass.View_ConversationCmp_Host0.detectChangesInternal (/MailModule/ConversationCmp/host.ngfactory.js:29:19)
    at CompiledTemplate.proxyViewClass.AppView.detectChanges (http://localhost:4200/vendor.bundle.js:89862:14)
    at CompiledTemplate.proxyViewClass.DebugAppView.detectChanges (http://localhost:4200/vendor.bundle.js:90057:44)
    at ViewRef_.detectChanges (http://localhost:4200/vendor.bundle.js:64430:20)
    at RouterOutlet.activate (http://localhost:4200/vendor.bundle.js:74868:42)
    at ActivateRoutes.placeComponentIntoOutlet (http://localhost:4200/vendor.bundle.js:26017:16)
    at ActivateRoutes.activateRoutes (http://localhost:4200/vendor.bundle.js:25984:26)
    at http://localhost:4200/vendor.bundle.js:25920:58
    at Array.forEach (native)
```

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 (https://augury.angular.io/) chrome extension to inspect the component and injector trees.



```
▼ MailAppCmp
    li
    router-outlet
  ▼ ConversationCmp
      router-outlet
    ▼ MessagesCmp
      ▶ MdCard
      ▶ MdCard
    ▼ MdButton
        div
    ▼ MdButton
        div
    router-outlet
    li
```

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, that is, 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`, that is, the following two examples are identical:

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

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

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 7. Change Detection

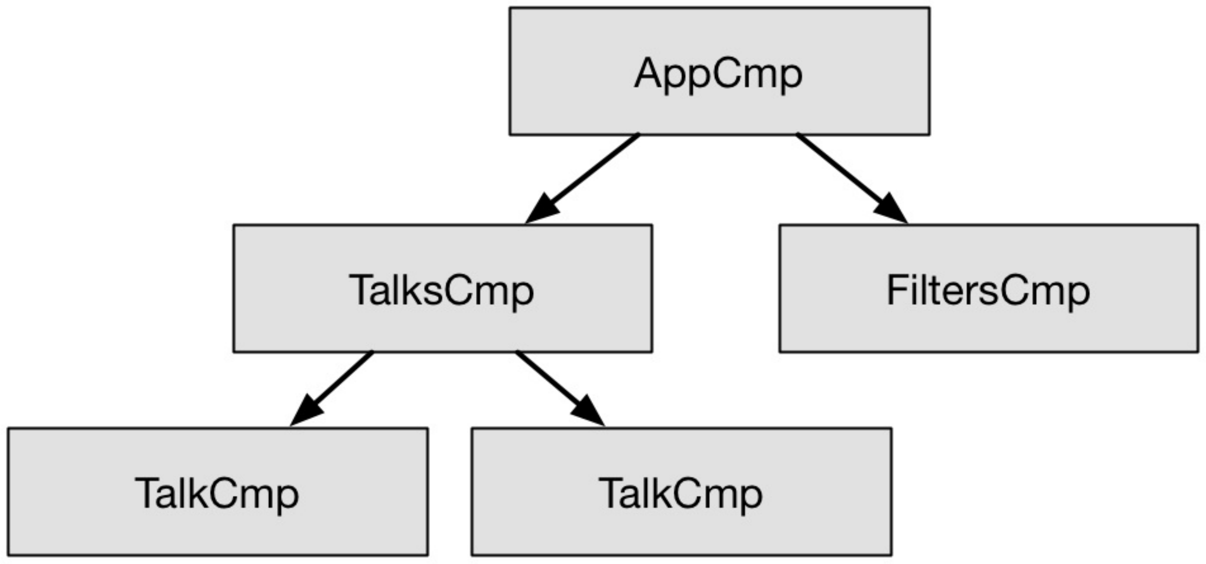
Two phases

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",
     ...
```

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...

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 ></tab>
      <tab ></tab>
      <tab ></tab>
    </tabs>
  `
})
class CmpUsingTabs {
}
```

The content children of the `tabs` component are the three ...

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 preceding example, 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)">\
   ...
```

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 8. Forms

Web applications heavily rely on forms. In many ways Angular is so successful because two-way bindings and `ng-model` made creating dynamic forms easy.

Although very flexible, the AngularJS 1.x approach has some issues: the data flow in complex user interactions is hard to understand and debug.

Angular 2+ builds up on the ideas from AngularJS 1.x: it preserves the ease of creating dynamic forms, but avoids the issues making data flow hard to understand.

In this chapter we will look at how form handling (or input handling) works in Angular.

Two modules

In AngularJS 1.x, the `ng-model` directive was baked into the core framework. This is no longer the case. The `@angular/core` package doesn't contain a form-handling library. It only provides the key primitives we can use to build one.

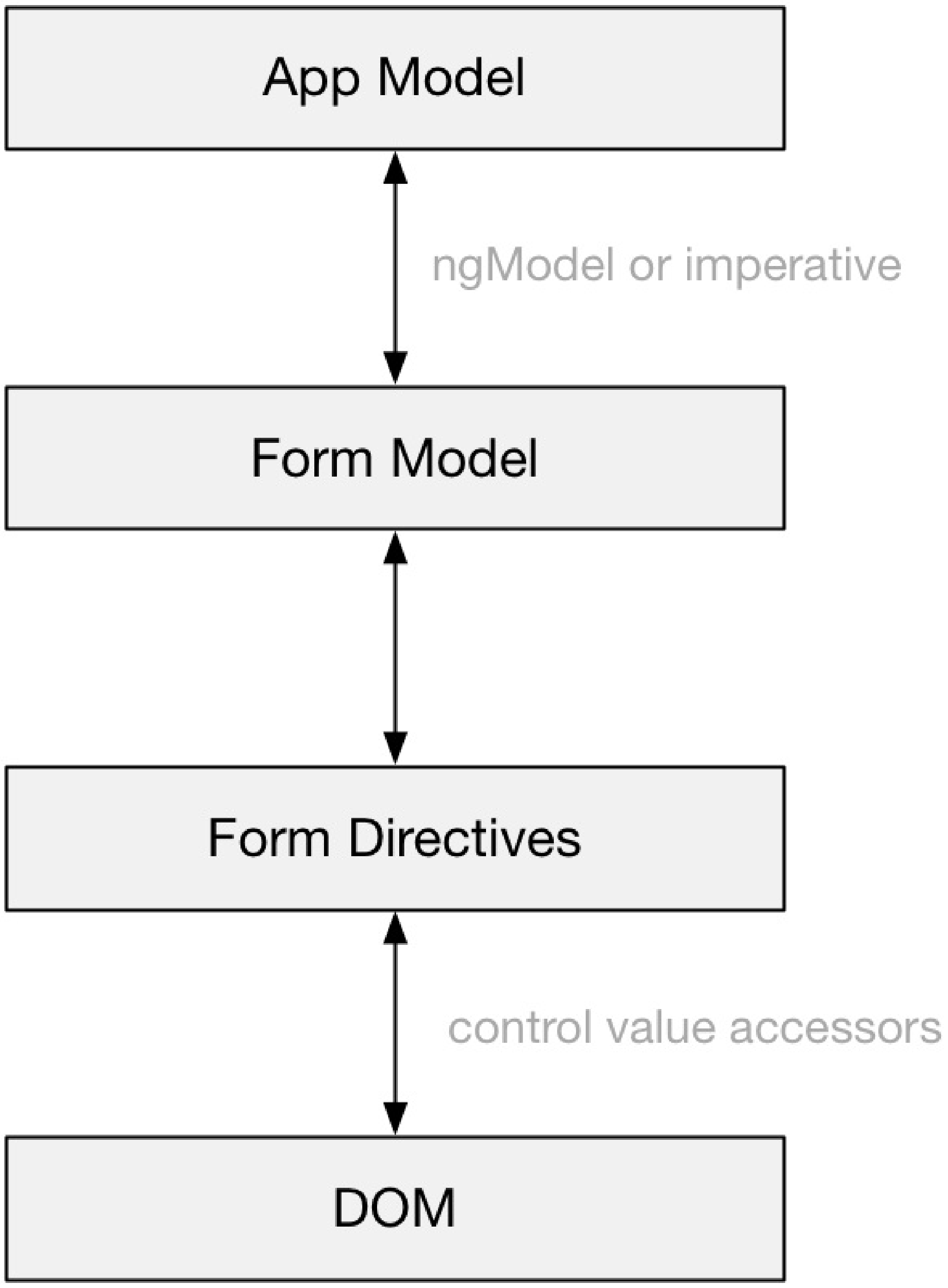
Of course, making everyone to build their own would not be practical. And that's why the Angular team built the `@angular/forms` package with two modules: `FormsModule` and `ReactiveFormsModule`.

`FormsModule` implements AngularJS-style form handling. We create a form by placing directives in the template. We then use data bindings to get data in and out of that form.

`ReactiveFormsModule` is another take on handling input, where we define a form in the component class and just bind it to elements in the template. We tend to use reactive programming to get data in and out of the form, hence the name "reactive".

At first glance, these two modules seem very different. But once we understand the underlying mechanisms, we will see how much they have in common. In addition, it will give us an idea of how to build our own form-handling module if needed.

High-level overview



App model

The app model is an object provided by the application developer. It can be a JSON object fetched from the server, or some object constructed on the client side. Angular doesn't make any assumptions about it.

Form model

The form model is a UI-independent representation of a form. It consists of three building blocks: `FormControl`, `FormGroup`, and `FormArray`. We will look at the form model in detail later in this chapter. Both `FormsModule` and `ReactiveFormsModule` use this model.

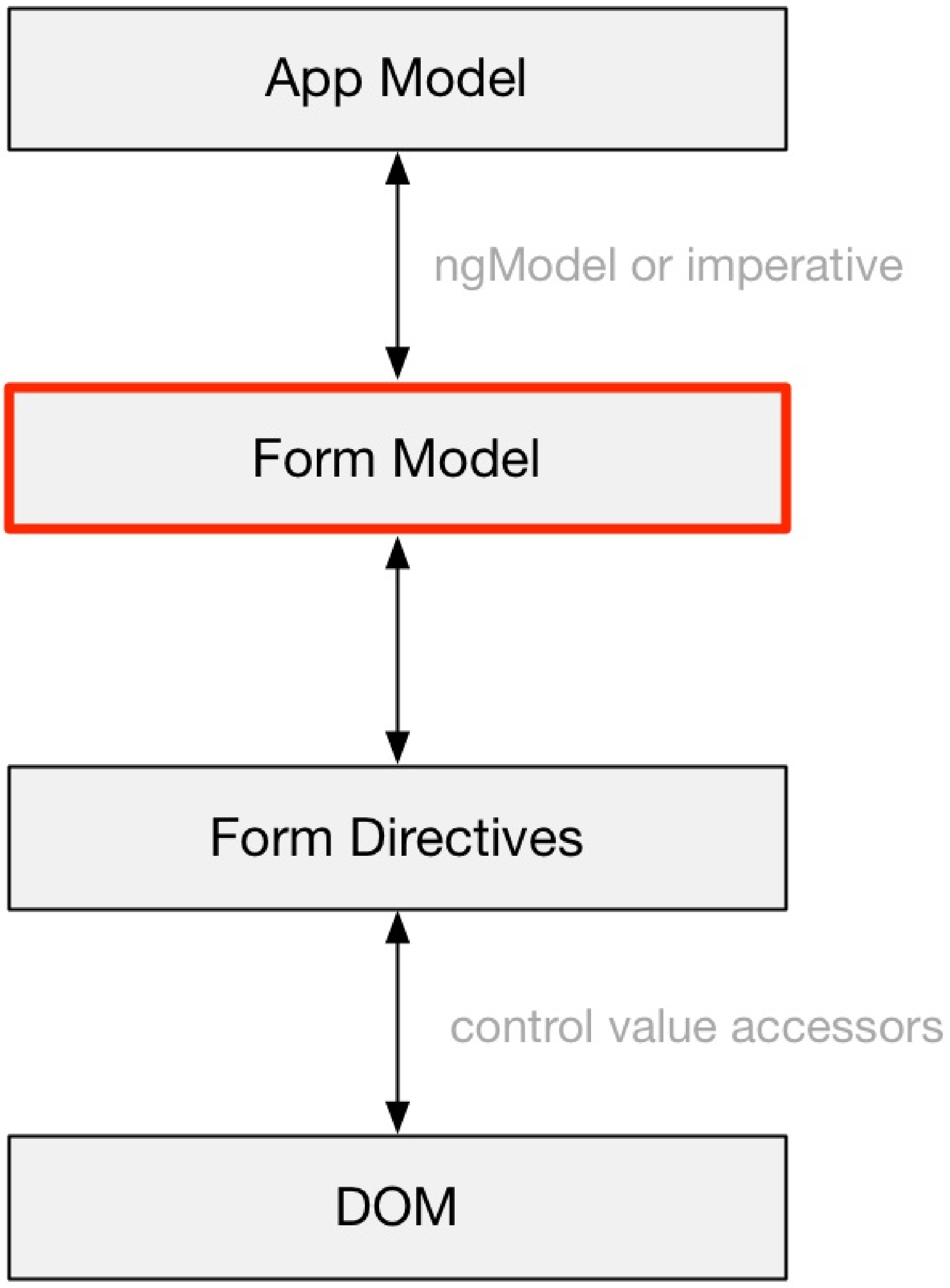
Form directives

These are the directives connecting the form model to the DOM (for example, NgModel). `FormsModule` and `ReactiveFormsModule` provide different sets of these directives.

DOM

These are ours inputs, checkboxes, and radio buttons.

Form model


App Model
ngModel or imperative
Form Model
Form Directives
control value accessors
DOM

To make form handling less UI-dependent, `@angular/forms` provides a set of primitives for modelling forms: `FormControl`, `FormGroup`, and `FormArray`.

FormControl

`FormControl` is an indivisible part of the form, an atom. It usually corresponds to a simple UI element, such as an input.

```
const c = newFormControl('Init Value', Validators.required);
```

A `FormControl` has a value, status, and a map of errors:

```
expect(c.value).toEqual('Init Value');
expect(c.errors).toEqual(null); //null means 'no errors'
expect(c.status).toEqual('VALID');
```

FormGroup

`FormGroup` is a fixed-size collection of controls, a record.

```
const c = new FormGroup({
  login: new FormControl(''),
  password: new FormControl('', Validators.required)
});
```

A `FormGroup` is itself a control, and, as such, has the same methods as `FormControl`.

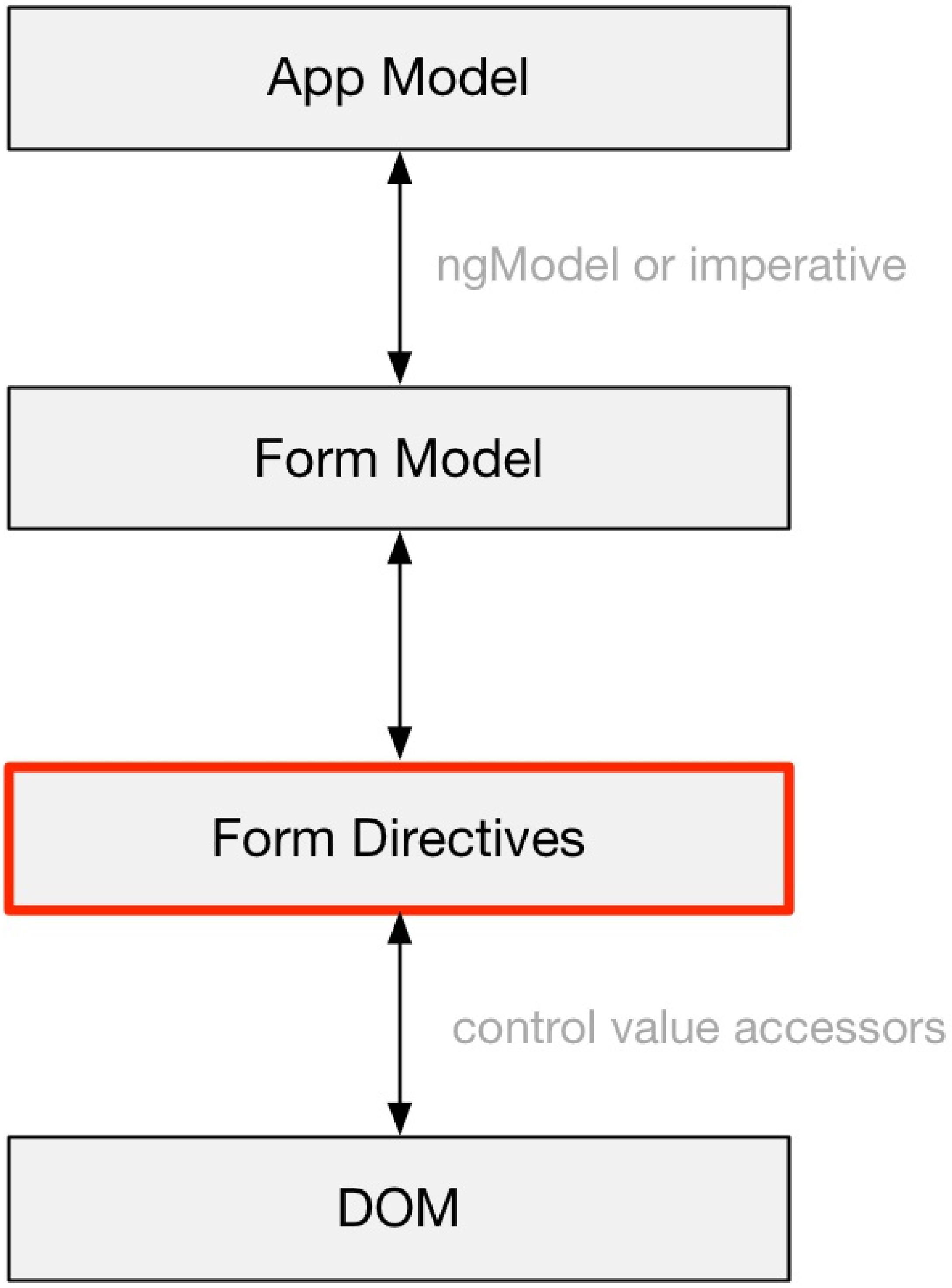
```
expect(c.value).toEqual({login: '', password: ''});
expect(c.errors).toEqual(null);
expect(c.status).toEqual('INVALID');
```

The value of a group is just an aggregation of the values of its children. Any time the value of a child control changes, the value of the group will change as well.

In opposite to the value, the form group doesn't aggregate the...

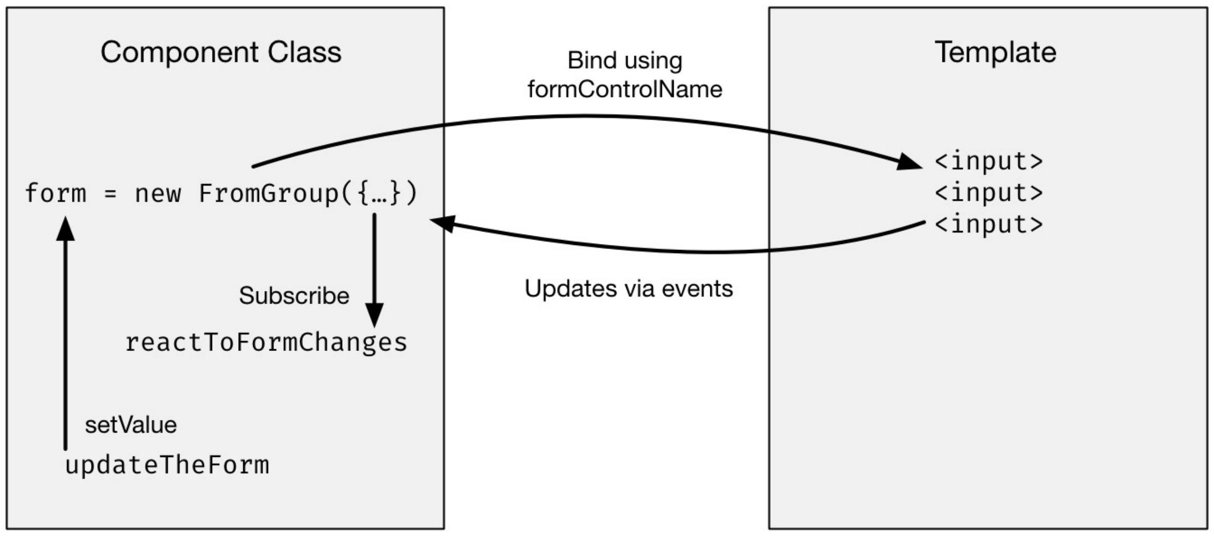
Form directives

Abstractly describing input is all well and good, but at some point we will need to connect it to the UI.



`@angular/forms` provides two modules that do that: `FormsModule` and `ReactiveFormsModule`.

ReactiveFormsModule



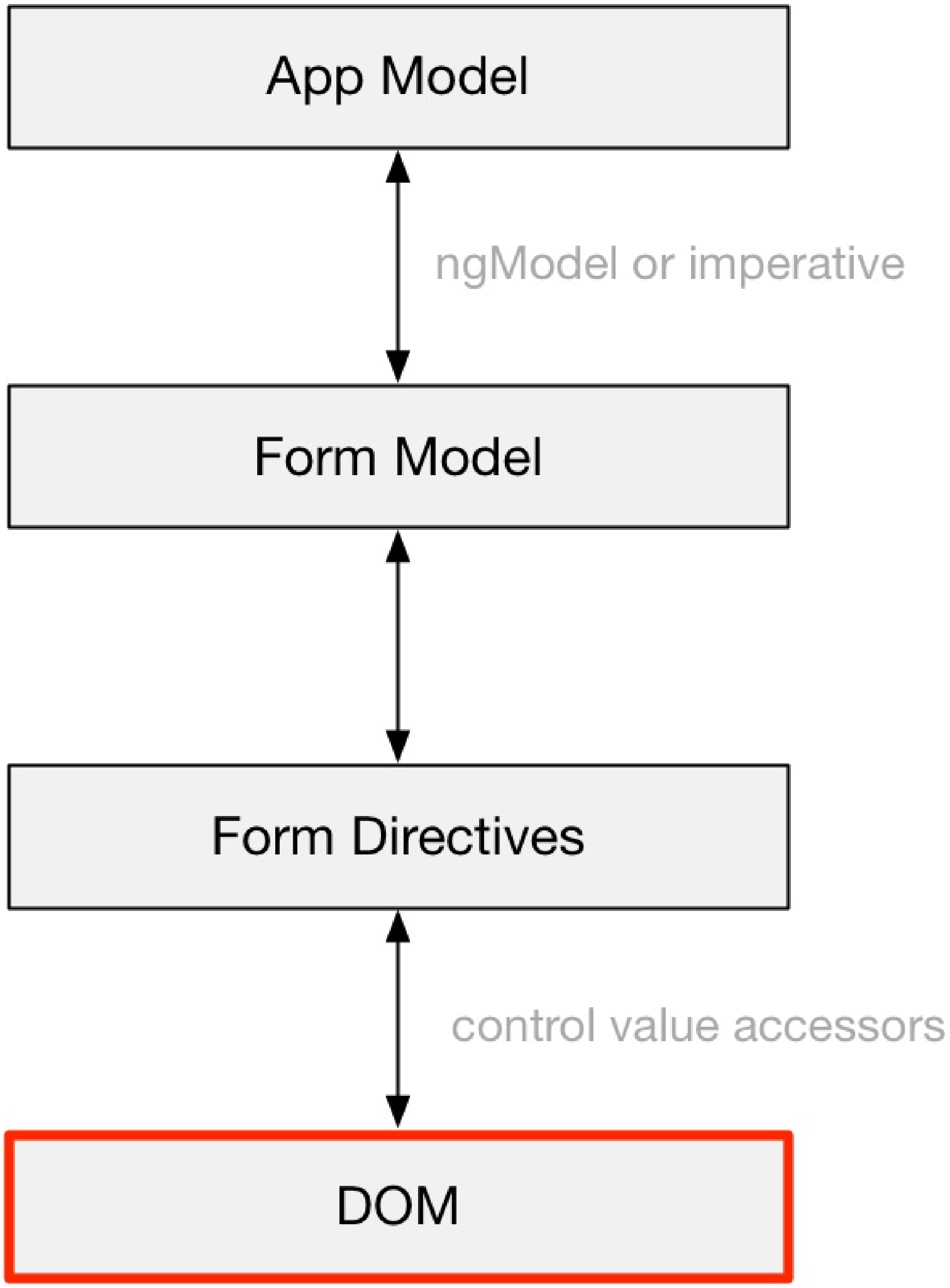
`ReactiveFormsModule` is simpler to understand and explain than `FormsModule`. That's why I will cover it first.

```
@Component({
selector: 'filters-cmp',
template: `
    <div [formGroup]="filters">
      <input formControlName="title" placeholder="Title">
      <input formControlName="speaker" placeholder="Speaker">
      <input type="checkbox" formControlName="highRating">
      High Rating
    </div>
  `
})
export class FiltersCmp {
  filters = new FormGroup({
    speaker: new FormControl(),
    title: new FormControl(),
    highRating: new FormControl(false),
  });

  constructor(app: App) {
    this.filters.valueChanges.debounceTime(200).\
    subscribe((value) => {
      app.applyFilters(value);
    });
  }
}

@NgModule({
  imports: [
    ReactiveFormsModule
  ],
  declarations: [
    FiltersCmp
  ]
})
class AppModule {}
```

There are a few things here to note.

First, we import `ReactiveFormsModule`, which...

The DOM


App Model
ngModel or imperative
Form Model
Form Directives
control value accessors
DOM

The `ngModel`, `ngControlName`, and other form directives bind the form model to UI elements, which are often native to the platform (for example, `<input>`), but they do not have to be. For instance, `NgModel` can be applied to an Angular component.

```
<md-input[(ngModel)]="speaker" name="speaker" placeholder="Spea
```

A `ControlValueAccessor` is a directive that acts like a adapter connecting a UI element to `NgModel`. It knows how to read and write to the native UI-element.

The `@angular/forms` package comes with value accessors for all built-in UI elements (`input`, `textarea`, so on). But if we want to apply an `NgModel` to a custom element or an Angular component, we will have to provide a value accessor for it ourselves.

```
@Component({
  selector: 'custom-input',
  providers: [
    {
      provide: NG_VALUE_ACCESSOR,
      useExisting: forwardRef(() => CustomInputCmp),
      multi: true
    }
  ]
})
class CustomInputCmp implements ControlValueAccessor {
  //...
}
```

Wrapping up

Form handling is a complex problem. One of the main reasons AngularJS got so successful is that two-way bindings and `ng-model` provided a good solution for it. But there were some downsides, mainly complex forms built with `ng-model` made the data flow of the application hard to follow and debug. Angular 2+ builds up on the ideas from Angular 1, but avoids its problems.

`NgModel` and friends are no longer part of the core framework. The `@angular/core` package only contains the primitives we can use to build a form-handling module. Instead, Angular has a separate package —`@angular/forms`—that comes with `FormsModule` and `ReactiveFormsModule` that provide two different styles of handling user input.

Both the modules depend on the form model consisting of `FormControl`, `FormGroup`, and `FormArray`. Having this UI-independent model, we can model and test input handling without rendering any components.

Finally, `@angular/forms` comes with a set of directives to handle build-in UI elements (such as `<input>`), but we can provide our own.

Chapter 9. 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: Fu
    this.filters.valueChanges.debounceTime(200).\
    subscribe((value) => {
      this.change.next(createFilters(value));
    });
  }
}
```

Following is the code for `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="Spea
  </md-input-container>

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

There a few things in this example worth noting:

- We are...

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"></ta
})
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
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]
    })
   ...
```

Integration testing

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

```
import {TestBed, async, ComponentFixture, inject} from '@angula
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
        },
...
```

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).geText();
  }
}

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 in the preceding...

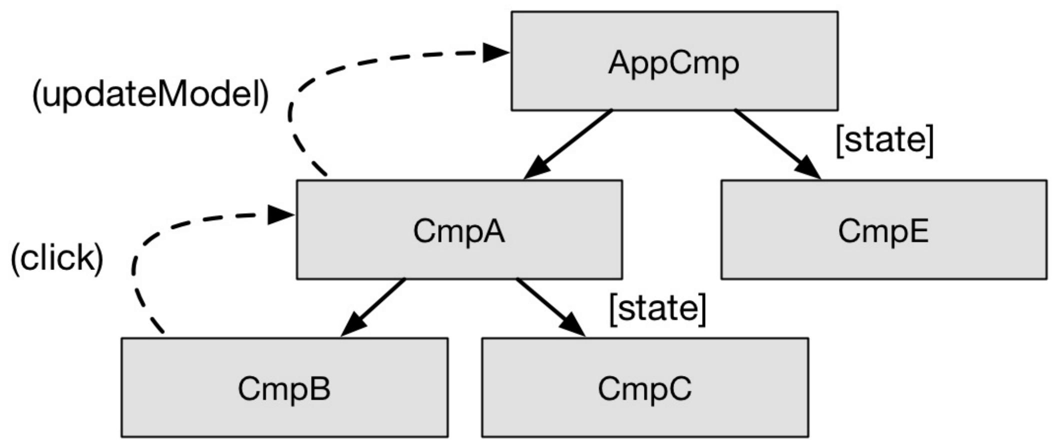
Let's recap

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.

Chapter 10. Reactive Programming in Angular

Reactive programming in the core framework

An Angular application is a reactive system. The user clicks on a button, the application reacts to this event and updates the model. The model gets updated, the application propagates the changes through the component tree.



Angular implements these two arrows very differently. Let's explore why.

Events and state

To understand why Angular uses two very different ways of reactive programming, we need to look at the differences between events and the state.

We often talk about events or event streams when discussing reactivity. Event streams are an important category of reactive objects, but so is state. So let's compare their properties.

Events are discrete and cannot be skipped. Every single event matters, including the order in which the events are emitted. The "most recent event" is not a special thing we care about. Finally, very rarely are events directly displayed to the user.

State, on the other hand, is continuous, that is, it is defined at any point in time. We usually do not care about how many times it gets updated—only the most recent value matters. The state is often displayed or...

Reactive programming in the Angular ecosystem

We have looked at how the Angular core framework itself supports reactive programming. Now let's look at the Angular ecosystem.

@angular/forms

Angular has always had strong support for building dynamic forms. It's one of the main reasons the framework got so successful.

Now the framework comes with a module that adds support for handling input using reified reactive programming.

```
import {ReactiveFormsModule, FormGroup, FormControl} from '@ang

@Component({
  selector: 'filters-and-talks',
  template: `
    <form [formGroup]="filtersForm">
      Title <input formControlName="title">
      Speaker <input formControlName="speaker">
    </form>

    <talk *ngFor="let t of talks|async" [talk]="t"></talk>
  `
})
class TalksAndFiltersCmp {
  filtersForm = new FormGroup({
    title: new FormControl(''),
    speaker: new FormControl('')
  });

  talks: Observable<Talk[]>;

  constructor(backend: Backend) {
    this.talks = this.filtersForm.valueChanges.
      debounceTime(100).
      switchMap(filters => backend.fetch(filters));
  }
}

@NgModule({
  declarations: [TalksAndFiltersCmp,...
```

Summary

An Angular application is a reactive system. And that's why we need to understand reactive programming to be productive with Angular.

Reactive programming works with event streams and the state. And it can be divided into transparent and reified.

Since the very beginning the framework has had excellent support for transparent reactive programming. It was used both to propagate the state and to handle events. It is simple and fast. And the new versions of the framework still support it.

But it can also be limiting at times and make solving certain problems difficult. That's why Angular now comes with support for reified reactive programming, using observables.

The Angular ecosystem embraced these ideas as well. The reactive forms module, the router, and other libraries like NgRx, all provide observable-based APIs.