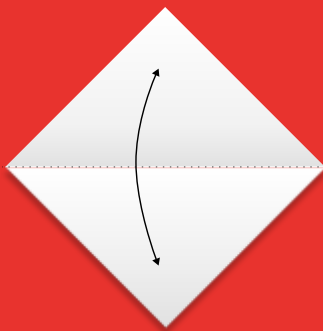


LEARN ANGULAR: YOUR FIRST WEEK



1.



2.



3.

GET STARTED WITH ANGULAR

Your First Week With Angular

Copyright © 2018 SitePoint Pty. Ltd.

Ebook ISBN: 978-0-6483315-6-8

- **Product Manager:** Simon Mackie
- **English Editor:** Ralph Mason
- **Project Editor:** Nilson Jacques
- **Cover Designer:** Alex Walker

Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.



Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood

VIC Australia 3066

Web: www.sitepoint.com

Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit <http://www.sitepoint.com/> to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

Table of Contents

Preface x

Who Should Read This Book?xi

Conventions Usedxi

Chapter 1: Angular Introduction: What It Is, and

Why You Should Use It 14

Why Do I Need a Framework? 15

Angular Introduction: What Angular IS 16

Angular Introduction: the Advantages of Angular 17

Angular Introduction: Angular's Complexities 18

Conclusion20

Chapter 2: AngularJS and Angular: a Detailed

Comparison..... 21

Frameworks in AngularJS and Angular 22

Templates in AngularJS and Angular 24

Dependency Injection in AngularJS and Angular..... 25

JavaScript vs TypeScript..... 27

Tooling Support for AngularJS and Angular 28

| | |
|---------------|----|
| Summary | 29 |
|---------------|----|

Chapter 3: An Introduction to TypeScript: Static

Typing for the Web.....30

| | |
|---------------------------------------|----|
| What Exactly is TypeScript? | 31 |
| How Does it Work? | 31 |
| How Do I Set it Up?..... | 34 |
| Okay, What About the Community? | 37 |
| Further Reading | 37 |
| Conclusion | 37 |

Chapter 4: Getting Past Hello World in Angular38

| | |
|-------------------------------------|----|
| Environment Setup | 39 |
| Components..... | 41 |
| Decorators..... | 42 |
| Examining a Component..... | 43 |
| Services | 50 |
| Tree of Components..... | 52 |
| Bootstrapping our Application | 53 |
| Summary..... | 54 |

Chapter 5: Angular Components: Inputs and

| | |
|---|-----------|
| Outputs..... | 56 |
| Component Hierarchy | 58 |
| Angular Components: Inputs and Outputs..... | 61 |
| @Input() and @Output() | 64 |
| Wrapup | 65 |

Chapter 6: A Practical Guide to Angular Directives67

| | |
|--|----|
| Basic overview | 68 |
| Using the Existing Angular Directives..... | 69 |

Chapter 7: Angular Components and Providers:

| | |
|--|-----------|
| Classes, Factories & Values | 77 |
| What are Providers? | 79 |
| Injecting Non-Class Providers | 84 |
| Providers and Singletons | 86 |
| Wrapup | 90 |

Chapter 8: Quickly Create Simple Yet Powerful

| | |
|-----------------------------|-----------|
| Angular Forms..... | 91 |
| Prerequisites | 92 |
| Requirements | 92 |
| Template-Driven Forms | 93 |

| | |
|----------------------|-----|
| Reactive Forms | 99 |
| Wrapping Up | 106 |

Chapter 9: Using Angular NgModules for Reusable

| | |
|---|------------|
| Code and More | 107 |
| JavaScript Modules Aren't NgModules | 108 |
| The Basic NgModule, the AppModule | 108 |
| The Properties of NgModule | 109 |
| NgModule Examples | 111 |
| Summary | 120 |

Chapter 10: Angular Testing: A Developer's

| | |
|------------------------------------|------------|
| Introduction | 121 |
| Prerequisites | 122 |
| Angular Testing Technologies | 124 |
| Writing Unit Tests | 128 |
| End-to-end Angular Testing | 142 |
| Code Coverage | 145 |
| Additional Utilities | 146 |
| Wrapping Up | 149 |

Chapter 11: Creating UIs with Angular Material

Design Components151

 What's Material Design?152

 Introduction to Angular Material152

 Requirements153

 Create the Project with the Angular CLI153

 Getting Started with Angular Material154

 Importing Angular Material Components 156

 Create the UI for the Application..... 158

 Conclusion 176

Chapter 12: Developing Angular Apps without a

Back End Using MockBackend178

 A Ticketing System without a Real Back End..... 180

 The Angular Project Setup 182

 Working with Components 186

 The Ticket Service 192

 Conclusion 193

Chapter 13: React vs Angular: An In-depth

Comparison.....195

 Where to Start? 196

 Maturity..... 197

| | |
|--|-----|
| Features..... | 198 |
| Languages, Paradigms, and Patterns | 200 |
| Ecosystem | 203 |
| Adoption, Learning Curve and Development Experience..... | 206 |
| Putting it Into Context..... | 208 |
| One Framework to Rule Them All? | 209 |

Preface

So, why Angular? Well, because it's supported on various platforms (web, mobile, desktop native), it's powerful, modern, has a nice ecosystem, and it's just cool. Not convinced? Let's be a bit more eloquent, then:

- **Angular presents you not only the tools but also design patterns to build your project in a maintainable way.** When an Angular application is crafted properly, you don't end up with a tangle of classes and methods that are hard to modify and even harder to test. The code is structured conveniently and you won't need to spend much time in order to understand what is going on.
- **It's JavaScript, but better.** Angular is built with TypeScript, which in turn relies on JS ES6. You don't need to learn a totally new language, but you still receive features like static typing, interfaces, classes, namespaces, decorators etc.
- **No need to reinvent the bicycle.** With Angular, you already have lots of tools to start crafting the application right away. You have directives to give HTML elements dynamic behavior. You can power up the forms using `FormControl` and introduce various validation rules. You may easily send asynchronous HTTP requests of various types. You can set up routing with little hassle. And there are many more goodies that Angular can offer us!
- **Components are decoupled.** Angular strived to remove tight coupling between various components of the application. Injection happens in NodeJS-style and you may replace various components with ease.
- **All DOM manipulation happens where it should happen.** With Angular, you don't tightly couple presentation and the application's logic making your markup much cleaner and simpler.
- **Testing is at the heart.** Angular is meant to be thoroughly tested and it supports both unit and end-to-end testing with tools like Jasmine and Protractor.
- **Angular is mobile and desktop-ready**, meaning you have one framework for multiple platforms.
- **Angular is actively maintained** and has a large community and ecosystem.

You can find lots of materials on this framework as well as many useful third-party tools.

So, we can say that Angular is not just a framework, but rather a *platform* that empowers developers to build applications for the web, mobile, and the desktop.

This book provides a rapid introduction to Angular, getting you up and running with no fuss.

Who Should Read This Book?

This book is for all front-end developers who want to get proficient with Angular and its related tools. You'll need to be familiar with HTML and CSS and have a reasonable level of understanding of JavaScript in order to follow the discussion.

Conventions Used

You'll notice that we've used certain typographic and layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>  
<p>It was a lovely day for a walk in the park.  
The birds were singing and the kids were all back at school.</p>
```

Where existing code is required for context, rather than repeat all of it, `:` will be displayed:

```
function animate() {
```

```
    :  
    new_variable = "Hello";  
  }
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An ↪ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-  
↪design-real-user-testing/?responsive1");
```

Tips, Notes, and Warnings



Hey, You!

Tips provide helpful little pointers.



Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.



Make Sure You Always ...

... pay attention to these important points.



Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.



Live Code

This example has a Live Codepen.io Demo you can play with.



Github

This example has a code repository available at Github.com.

Angular Introduction: What It Is, and Why You Should Use It

Ilya Bodrov-Krukowski

Chapter

1

In this article, I'm going to give you a general overview of a very popular and widely used client-side framework called Angular. This Angular introduction is mostly aimed at newcomer developers who have little experience with JS frameworks and wish to learn the basic idea behind Angular as well as understand its differences from AngularJS.

A *JavaScript framework* is a kind of buzzword these days: everyone keeps discussing these frameworks, and many developers are arguing about the best solution.

So, let's get this Angular introduction started, shall we?

Why Do I Need a Framework?

If you're not sure what a JavaScript (or client-side) framework is, that's a technology providing us the right tools to build a web application while also defining how it should be designed and how the code should be organized.

Most JS frameworks these days are *opinionated*, meaning they have their own philosophy of how the web app should be built and you may need to spend some time to learn the core concepts. Other solutions, like Backbone, do not instruct developers on how they should craft the project, thus some people even call such technologies simply libraries, rather than frameworks.

Actually, JavaScript frameworks emerged not that long ago. I remember times where websites were built with poorly structured JS code (in many cases, powered by jQuery). However, client-side UIs have become more and more complex, and JavaScript lost its reputation as a "toy" language. Modern websites rely heavily on JS and the need to properly organize (and test!) the code has arisen. Therefore, client-side frameworks have become popular and nowadays there are at least dozen of them.

Angular Introduction: What Angular IS

AngularJS used to be the “golden child” among JavaScript frameworks, as it was initially introduced by Google corporation in 2012. It was built with the Model-View-Controller concept in mind, though authors of the framework often called it “Model-View-***” or even “Model-View-Whatever”.

The framework, written in pure JavaScript, was intended to decouple an application’s logic from DOM manipulation, and aimed at dynamic page updates. Still, it wasn’t very intrusive: you could have only a part of the page controlled by AngularJS. This framework introduced many powerful features allowing the developer to create rich, single-page applications quite easily.

Specifically, an interesting concept of data binding was introduced that meant automatic updates of the view whenever the model (data) changed, and vice versa. On top of that, the idea of directives was presented, which allowed inventing your own HTML tags, brought to life by JavaScript. For example, you may write:

```
<calendar></calendar>
```

This is a custom tag that will be processed by AngularJS and turned to a full-fledged calendar as instructed by the underlying code. (Of course, your job would be to code the appropriate directive.)

Another quite important thing was Dependency Injection, which allowed application components to be wired together in a way that facilitated reusable and testable code. Of course, there’s much more to AngularJS, but we’re not going to discuss it thoroughly in this article.

AngularJS became popular very quickly and received a lot of traction. Still, its maintainers decided to take another step further and proceeded to develop a new version which was initially named *Angular* (later, simply *Angular* without the “JS” part). It’s no coincidence the framework received a new name: actually, it

was fully re-written and redesigned, while many concepts were reconsidered.

The first stable release of Angular was published in 2016, and since then AngularJS started to lose its popularity in favor of a new version. One of the main features of Angular was the ability to develop for multiple platforms: web, mobile, and native desktop (whereas AngularJS has no mobile support out of the box).

Then, to make things even more complex, by the end of 2016, *Angular 4* was released. “So, where is version 3?”, you might wonder. I was asking the same question, as it appears that version 3 was never published at all! How could this happen? As explained in the official blog post, maintainers decided to stick with the semantic versioning since Angular.

Following this principle, changing the *major* version (for example, “2.x.x” becomes “3.x.x”) means that some breaking changes were introduced. The problem is that the Angular Router component was already on version 3. Therefore, to fix this misalignment it was decided to skip Angular 3 altogether. Luckily, the transition from Angular to 4 was less painful than from AngularJS to Angular, though many developers were still quite confused about all this mess.

Angular 5 was released in November 2017. It is also backwards compatible with prior Angular versions. Angular 6 should be released quite soon, hopefully giving us even more cool features and enhancements.

Angular Introduction: the Advantages of Angular

So, why Angular? Well, because it’s supported on various platforms (web, mobile, desktop native), it’s powerful, modern, has a nice ecosystem, and it’s just cool. Not convinced? Let me be a bit more eloquent then:

- **Angular presents you not only the tools but also design patterns to build your project in a maintainable way.** When an Angular application is crafted properly, you don’t end up with a tangle of classes and methods that are hard

to modify and even harder to test. The code is structured conveniently and you won't need to spend much time in order to understand what is going on.

- **It's JavaScript, but better.** Angular is built with TypeScript, which in turn relies on JS ES6. You don't need to learn a totally new language, but you still receive features like static typing, interfaces, classes, namespaces, decorators etc.
- **No need to reinvent the bicycle.** With Angular, you already have lots of tools to start crafting the application right away. You have directives to give HTML elements dynamic behavior. You can power up the forms using `FormControl` and introduce various validation rules. You may easily send asynchronous HTTP requests of various types. You can set up routing with little hassle. And there are many more goodies that Angular can offer us!
- **Components are decoupled.** Angular strived to remove tight coupling between various components of the application. Injection happens in NodeJS-style and you may replace various components with ease.
- **All DOM manipulation happens where it should happen.** With Angular, you don't tightly couple presentation and the application's logic making your markup much cleaner and simpler.
- **Testing is at the heart.** Angular is meant to be thoroughly tested and it supports both unit and end-to-end testing with tools like Jasmine and Protractor.
- **Angular is mobile and desktop-ready**, meaning you have one framework for multiple platforms.
- **Angular is actively maintained** and has a large community and ecosystem. You can find lots of materials on this framework as well as many useful third-party tools.

So, we can say that Angular is not just a framework, but rather a *platform* that empowers developers to build applications for the web, mobile, and the desktop. You may learn more about its architecture [in this guide](#).

Angular Introduction: Angular's Complexities

I have to say that, unfortunately, Angular is quite a big and complex framework with its own philosophy, which can be challenging for newcomers to understand

and get used to. Learning the framework's concepts is not the only task, however; on top of this, you also have to be comfortable with a handful of additional technologies:

- It's recommended to code Angular apps in TypeScript, so you must understand it. It is possible to write the code with modern JavaScript (ES6), though I rarely see people doing this.
- TypeScript is a superset of JavaScript, so you'll need to be comfortable with it as well.
- It's a good idea to get the grasp of the Angular CLI to speed up the development process even further.
- Node's package manager npm is used extensively to install Angular itself and other components, so you'll need to be comfortable with that as well.
- Learning how to set up a task runner like Gulp or Grunt can come in really handy, as there can be lots of things to be done before the application is actually deployed to production.
- Using minifiers (like UglifyJS) and bundlers (like Webpack) is also very common these days.
- While developing the app, it's vital to be able to debug the code, so you should know how to work with debugging tools like Augury.
- Of course, it's very important to test Angular applications, which can become very complex. One of the most popular testing tools out there are called Jasmine (which is a framework for testing) and Protractor (which is used for end-to-end testing).

So, as you see, there are quite a lot of things to learn in order to start creating client-side web applications. But don't be put off: there are numerous resources on the net that may help you learn all these tools and technologies. Of course, you'll need some time to get the grasp of them, but as a result, you'll get valuable experience and will be able to create complex apps with confidence.

One last thing worth mentioning is that sometimes using Angular for an app may be overkill. If you have a small or medium-sized project without any complex user interfaces and interactions, it may be a much better idea to stick with plain old

JavaScript. Therefore, it's very important to assess all the requirements, features of the new application, as well as take deadlines into consideration before making a decision on whether to use a JavaScript framework or not.

Conclusion

In this Angular introduction, we've discussed Angular, a client-side framework supporting multiple platforms. We've covered some of its features and concepts, and also have seen how it differs from AngularJS, the previous version of the framework.

Hopefully you've now got a basic idea of what Angular is and in what cases it may come in handy!

AngularJS and Angular: a Detailed Comparison

Manjunath M

Chapter

2

This article compares the major differences between the the original AngularJS and Angular. If you're currently stuck with an AngularJS project and not sure whether you should make the jump, this article should help you get started.

In recent years, we've seen Angular grow tremendously as a framework and as a platform for developing single page applications (SPAs) and progressive web apps (PWAs). AngularJS was built on top of the idea that declarative programming should be used for building the views. This required decoupling the DOM manipulation from the business logic of the application and the approach had many benefits on its own.

However, AngularJS had many shortcomings in terms of performance and how things worked under the hood. Hence, the development team spent a year rewriting the code from scratch and finally released Angular 2 in late 2016. Most developers felt that Angular was a different platform that had very little resemblance to the original AngularJS.

So let's compare and contrast AngularJS and Angular.

Frameworks in AngularJS and Angular

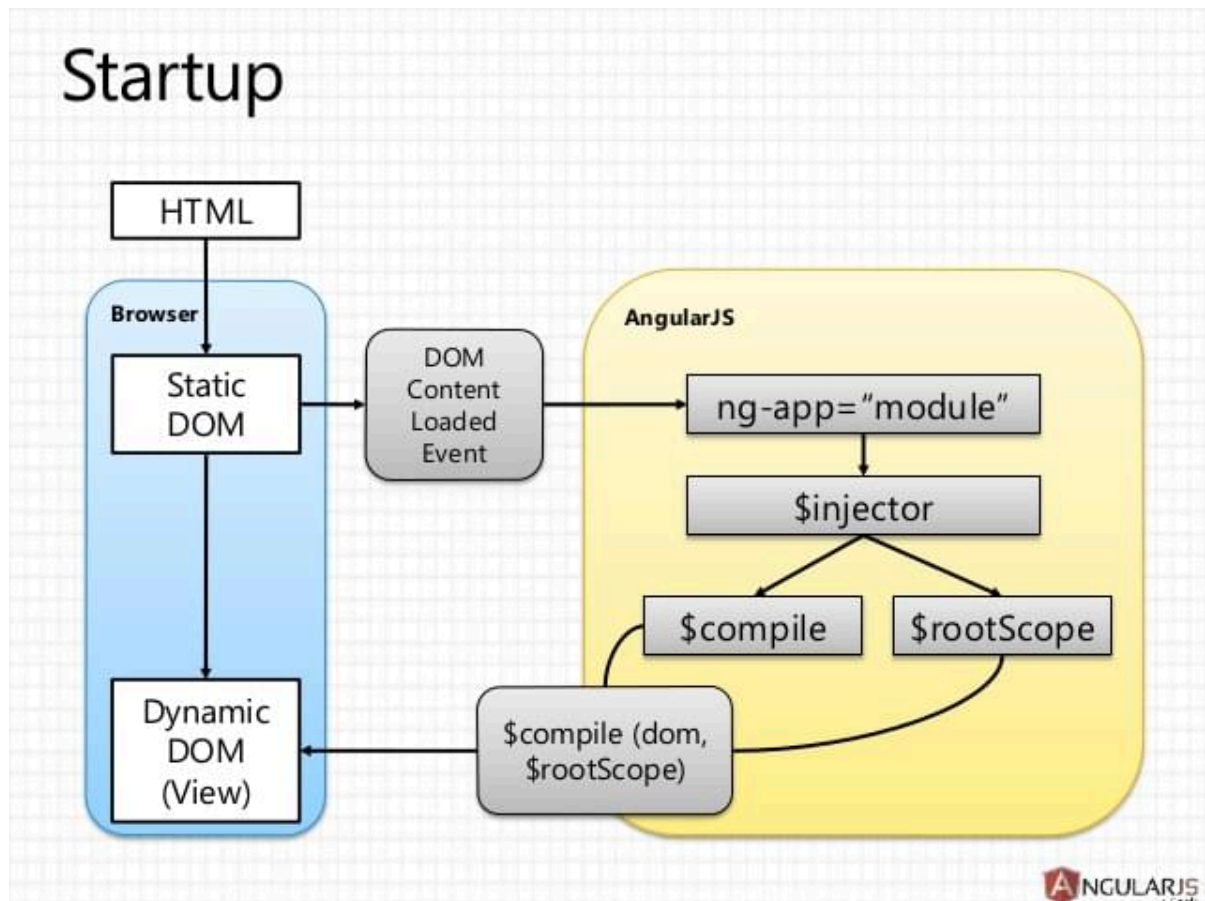
AngularJS follows the traditional MVC architecture that comprises a model, a view and a controller.

- Controller: the controller represents how user interactions are handled and binds both the model and the view.
- Views: the view represents the presentation layer and the actual UI.
- Model: the model is an abstract representation of your data.

Some developers are of the opinion that AngularJS follows MVVM pattern that replaces the Controller with a View-Model. A View-Model is a JavaScript function that's similar to that of the controller. What makes it special is that it synchronizes the data between a view and a model. The changes made to a UI

element automatically propagate to the model and vice versa.

The following diagram shows how various AngularJS pieces are connected together.

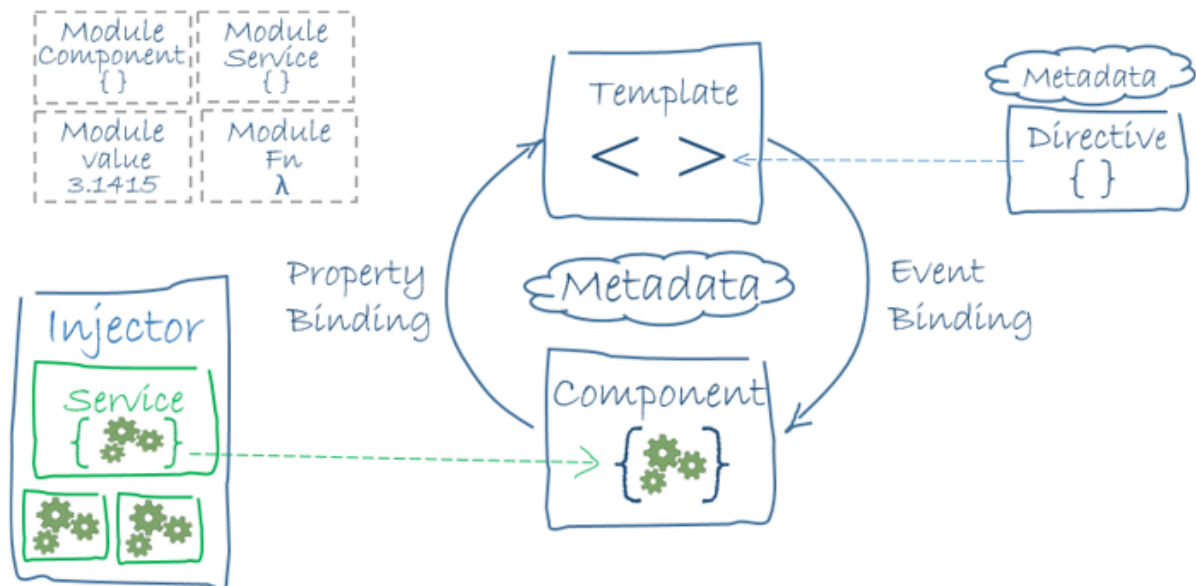


2-1. AngularJS and Angular: AngularJS architecture

You can read more about AngularJS's architecture on the [official documentation page](#).

Angular, on the other, hand has a component-based architecture. Every Angular application has at least one component known as the root component. Each component has an associated class that's responsible for handling the business logic and a template that represents the view layer. Multiple, closely related components can be stacked together to create a module and each module

forms a functional unit on its own.



2-2. AngularJS and Angular: High-level overview of the Angular architecture

As you can see in the figure, the component is bound to the template. Components are composed using TypeScript classes and templates are attached to them using `@Component` annotations. Services can be injected into a component using Angular's dependency injection subsystem. The concept of modules in Angular is drastically different from that of the AngularJS modules. An `NgModule` is a container for defining a functional unit. An `NgModule` can comprise components, services and other functions. The modular unit can then be imported and used with other modules.

All the Angular concepts are better explained at [Angular.io](https://angular.io).

Templates in AngularJS and Angular

In AngularJS the template is written using HTML. To make it dynamic, you can add AngularJS-specific code such as attributes, markups, filters and form controls. In addition, it supports the two-way data binding technique mentioned earlier. The following code snippet demonstrates the use of directives and

double curly brackets within the template:

```
<html ng-app>
  <!-- Body tag augmented with ngController directive -->
  <body ng-controller="MyController">
    <input#t ng-model="foo" value="bar">
    <!-- Button tag with ngClick directive -->
    <!-- Curly bracket is a template binding syntax -->
    button ng-click="changeFoo()">{{buttonText}}</button>
    <script src="angular.js"></script>
  </body>
</html>
```

In Angular, AngularJS's template structure was reworked and lots of new features were added to the templates. The primary difference was that each component had a template attached to it. All the HTML elements except `<html>`, `<body>`, `<base>`, and `<script>` work within the template. Apart from that, there are features such as template binding, template interpolation, template statements, property binding, event binding and two-way binding. Built-in attribute directives like `NgClass`, `NgStyle` and `NgModel` and built-in structural directives such as `NgIf`, `NgForOf`, `NgSwitch` are also part of the template.

Dependency Injection in AngularJS and Angular

Dependency Injection is a design pattern that takes care of satisfying dependencies and injecting them into the components when they're required. This avoids the need for hardcoding the dependencies into a component. AngularJS has an injector subsystem that's responsible for creating components, injecting dependencies and resolving the list all dependencies. The following components can be injected on an on-demand basis:

- value
- factory
- service

- provider
- constant

Services, directives and filters can be injected by using a factory method. Here's an example of a factory method in action. The factory method is registered with a module named `myModule`:

```
angular.module('myModule', [])
.factory('serviceId', ['depService', function(depService) {
  // ...
}])
.directive('directiveName', ['depService', function(depService) {
  'directiveName' // ...
}])
.filter('filterName', ['depService', function(depService) {
  // ...
}]);
```

Although the approach has stayed the same, Angular has a newer dependency injection system that's different from that of the older DI pattern. Angular's dependency injection is managed through the `@NgModule` array that comprises `providers` and `declarations`. The `declarations` array is the space where components and directives are declared. Dependencies and services are registered through the `providers` array.

Imagine you have a service that retrieves a list of contacts called `ContactListService` and provides it to a `ContactList` component. You'd first need to register the `ContactListService` in `app.module.ts` inside the `providers` array. Next, you'd need to inject the service into the component as follows:

```
import { Component } from '@angular/core';
import { Contact } from './contact';
import { ContactListService } from './contactlist.service';

@Component({
```

```

    selector: 'app-contacts-list',
    template: `
      <div *ngFor="let contact of contacts">
        {{contact.id}} - {{contact.name}} - {{contact.number}}
      </div>
    `
  })
  export class ContactListComponent {
    contacts: Contact[];

    constructor(contactlistService: ContactlistService) {
      this.contacts = contactlistService.getcontacts();
    }
  }
}

```

Here, we're telling Angular to inject the service into the component's constructor.

JavaScript vs TypeScript

AngularJS is a pure JavaScript framework, and models in AngularJS are plain old JavaScript objects. This makes the whole process of setting up the project a lot easier. Any developer with some basic JavaScript experience can get started with the framework. Because of this, AngularJS.0 has a very gentle learning curve compared to that of other front-end frameworks.

Angular introduced TypeScript as the default language for building applications. TypeScript is a syntactic superset of JavaScript that compiles down to plain JavaScript. The Angular team chose TypeScript over JavaScript because of the type annotation feature that lets you do optional static type checking. Type checking can prevent compile-time errors crawling into your code that would otherwise go unnoticed. This makes your JavaScript code more predictable.

Apart from that, TypeScript is also popular for its classes, interfaces and decorators (class decorators, property decorators and parameter decorators). TypeScript classes are used by Angular for defining components. `@Component` is

a popular example of how class decorators are used to attach metadata to a component. Usually, this includes component configuration details such as the template selector tag, the `templateUrl` and a `providers` array so that you can inject any related dependency into that component:

```
@Component({
  selector:    'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers:   [ HeroService ]
})
export class HeroListComponent implements OnInit {
  /* . . . */
}
```

Tooling Support for AngularJS and Angular

Better tooling support helps developers build things faster and it adds to the overall development workflow. A command-line interface (CLI) for instance can considerably reduce the amount of time spent creating an application from scratch. Similarly, there are other tools such as IDEs, text editors, test toolkits etc. that help you make development a whole lot easier.

AngularJS didn't have an official CLI, but there were many third-party generators and tools available. For the IDE, WebStorm and Aptana were the popular choices among the developers. If you're like me, you could customize a normal text editor like the Sublime Text editor and add the right plugins into it. AngularJS has a browser extension for debugging and testing called ng-inspector. The structure of AngularJS allowed third-party modules to be imported without any hassle. You can find all the popular ng modules at ngmodules.org, which is an open-source project for hosting AngularJS modules.

Angular has more tooling support compared to AngularJS. There's an official CLI that lets you initialize new projects, serve them and also build optimized bundles for production. You can read more about Angular CLI at GitHub. Because Angular

uses TypeScript instead of JavaScript, Visual Studio is supported as an IDE. That's not all. There are many IDE plugins and independent tools that help you automate and speed up certain aspects of your development cycle. Augury for debugging, NgRev for code analysis, Codelyzer for code validation etc. are pretty useful tools.

Summary

AngularJS had many flaws — most of them related to performance — but it used to be the first go-to choice for rapid prototyping. However, it doesn't make any sense to go back to AngularJS or maintain an AngularJS project anymore. If you haven't already made the shift, you should consider doing so.

In this article, we've covered the top five differences between AngularJS and Angular. Except for the template structure and the approach for dependency injection, almost all other features were revamped. Many of the popular AngularJS.0 features such as controllers, scope, directives, module definition etc. have been replaced with other alternatives. Furthermore, the underlying language has been changed and the structure modified.

An Introduction to TypeScript: Static Typing for the Web

Byron Houwens

Chapter

3

TypeScript is one of many attempts at creating a better experience with JavaScript.

“Oh, I’m using Gulp because of reason A” or “Oh, I’m using Redux because of reason B”. You hear these sorts of things from front-end developers all the time. It’s become fashionable to use new ways of improving on JavaScript’s old faults, and that’s not a bad thing. Even ES2015 and the updates that have followed have been pretty determined attempts at righting those wrongs.

TypeScript is a promising change to our favorite language that may be having a significant impact on JavaScript’s future.

What Exactly is TypeScript?

TypeScript is a strongly-typed superset of JavaScript, which means it adds some syntactical benefits to the language while still letting you write normal JavaScript if you want to. It encourages a more declarative style of programming through things like interfaces and static typing (more on these later), offers modules and classes, and most importantly, integrates relatively well with popular JavaScript libraries and code. You could think of it as a strongly static layer over current JavaScript that has a few features to make life (and debugging especially) a bit more bearable.

TypeScript gained particular attention a few years ago because it was selected for full support by Angular and following (which is also written in TypeScript itself). It’s also developed by Microsoft, which means it has the backing of two major tech companies (not a bad place for any language). Since this time, it’s gained more of a following and mainstream status.

Needless to say, TypeScript is definitely worth looking into.

How Does it Work?

TypeScript actually looks much like modern JavaScript. At the most basic level, it

introduces a static typing paradigm to JavaScript, so instead of the following:

```
var name = "Susan",  
    age = 25,  
    hasCode = true;
```

We could write the following:

```
let name: string = "Susan",  
    age: number = 25,  
    hasCode: boolean = true;
```

As you can see, there's not a whole lot of difference here. All we're doing is explicitly telling the system what type each variable is; we're telling it from the get-go that `name` is a string and `age` is a number. But that just seems like we have to write more code. Why bother telling the system such specific information? Because it gives the system more information about our program, which in turn means it can catch errors that we might make further down the road.

Imagine, for instance, you have something like this in your code:

```
var age = 25;  
age = "twenty-five";
```

Mutating a variable like this and changing its type will likely end up breaking stuff somewhere else, especially in a really big program, so it's great if the compiler can catch this before we load this up in our browser and have to sit for half an hour looking for the issue ourselves. Basically, it makes our program safer and more secure from bugs.

There's more, though. Here's an example from the TypeScript website intro tutorial (which you can find [here](#)):


```
interface Person {
    firstname: string;
    lastname: string;
}

function greeter(person : Person):string {
    return "Hello, " + person.firstname + " " + person.lastname;
}

let user = {firstname: "Jane", lastname: "User"};

document.body.innerHTML = greeter(user);
```

Now there are a few more unusual things here than we had before. We've got a run-of-the-mill object, called `user`, containing a first and last name, and that's being passed to `greeter()` and the output inserted into the body of the document. But there is some bizarre-looking stuff in the arguments of the `greeter` function, as well as something called an `interface`.

Let's start with the `greeter` function:

```
function greeter(person: Person):string {
    return "Hello, " + person.firstname + " " + person.lastname;
}
```

We can see that `greeter` takes a `person` parameter and we expect it to be of type `Person`. In this way, we can be sure that when we ask for that person's first name, it will definitely be there and we won't induce headaches upon ourselves if it fails. The `:string` after the function parameters tells us what type we expect this function to return when we call it.

The body of the function is nothing complicated but, of course, by now you're probably wondering what on earth a `Person` type actually is. This is where the `interface` feature comes in:

```
interface Person {  
  firstname: string;  
  lastname: string;  
}
```

Interfaces are used in TypeScript to define the structure of objects (and only objects). In this example, we're saying that any variable of type `Person` must be an object containing a `firstname` and a `lastname` property, both of the string type. We're basically creating a custom type for our object.

This is useful because it tells the compiler, as well as yourself and any developer who will work on this in the future, exactly what type of data to expect. We're basically *modelling* the object properties, creating something we can reference if we need to debug later. This is often why you'll see interfaces at the top of TypeScript files, as they give us a good idea of the data the program is working with in the rest of the file.

In our example, if we use this `Person` interface with a variable at any point in the program and it *doesn't* contain either a `firstname` or `lastname`, both of type `string` (our `user` object thankfully does), then the compiler will moan at us and we will be forced to mend our ways.

Not only that, but having static typing means that an IDE or editor with support for TypeScript will be able to provide us with very good, very specific hinting and auto-completion so that we can develop code that is both faster and safer.

There are many more features that TypeScript allows us to use, such as generics and namespaces, so at least a quick read of their [documentation](#) is highly recommended.

How Do I Set it Up?

Because TypeScript is a superset of JavaScript, we'll need to transpile it into JavaScript if we want to use it in the browser. Thankfully, it integrates well with a

number of task runners and bundlers already.

If you're just looking to play around with it locally first in particular, you can install TypeScript globally via [npm](#) and use it from the command line with the `tsc` command, like so:

```
tsc your-typescript-file.ts
```

This will output a JavaScript file, in this case called `your-typescript-file.js`, which you can then use in the browser as per usual. Setting it up in a project, though, will almost certainly entail setting up a proper `tsconfig.json`.

This file denotes that the project is a TypeScript project, and allows us to set a number of configuration options. Here's a truncated example from the docs:

```
{
  "compilerOptions": {
    "module": "commonjs",
    "outFile": "./build/local/tsc.js",
    "sourceMap": true
  },
  "exclude": [
    "node_modules"
  ]
}
```

Here we're configuring the compiler in a number of ways. We're specifying a module system to compile to, where to put the compiled file when it's finished and to include a source map. We're also giving it an `exclude` option, which basically tells the compiler to compile any TypeScript files — those ending in `.ts` — it finds as long as they're not in the `node_modules` folder.

From here, we can integrate things into our favorite task runner or bundler. Both [Grunt](#) and [Gulp](#) have plugins for TypeScript which will expose the compiler options for your task runners. Webpack has an awesome [TypeScript loader](#), and

there's good support for some other setups as well. Basically, you can get TypeScript integrated into pretty much any workflow you currently have going on without too much effort.

External Typings

If you're using external libraries in your project (let's be honest, who isn't?) you'll likely also need some type definitions. These definitions — denoted by a `.d.ts` extension — give us access to interfaces that other people have written for a number of JavaScript libraries. By and large, these definitions are available in a gigantic repo called DefinitelyTyped, which is where we install them from.

To use them you'll need to install Typings, which is kind of like npm but for TypeScript type definitions. It has its own config file, called `typings.json`, where you can configure your bundles and paths for type definition installation.

We won't go into too much detail here, but if we wanted to use AngularJS 1.x types, for example, we could simply go `typings install angularjs --save` and have them downloaded into a path defined in `typings.json`. After that, you could use Angular's type definitions anywhere in your project simply by including this line:

```
/// <reference path="angularjs/angular.d.ts" />
```

Now we can use Angular type definitions like the following:

```
var http: ng.IHttpService;
```

Any developers who happen upon our code at a later stage (or ourselves, three months after we've written it) will be able to make more sense of what we've written by looking at them.

Okay, What About the Community?

The TypeScript community is continuing to grow, as is the language's adoption. Perhaps most importantly, it's what Angular is written in and the framework provides full support for it straight from the beginning. There is also fantastic support for its syntax baked into Microsoft Visual Studio IDE and Visual Studio Code, with packages and plugins for editors like [Atom](#), [Sublime Text](#) and [Emacs](#) readily available as well.

What this means is that there's plenty of activity going on around TypeScript, so this is something you'll want to keep your eye on.

Further Reading

- [Official TypeScript site](#)
- [DefinitelyTyped](#) — 3rd-party TypeScript Definitions

Conclusion

TypeScript is an interesting push toward improving on JavaScript's shortcomings by introducing a static typing system, complete with interfaces and type unions. This helps us write safer, more legible and declarative code.

It integrates well with virtually every mainstream build setup out there at the moment and even gives us the ability to create and use custom types as well. There are also a myriad IDEs and text editors that have great support for its syntax and compile process, so you can use it in your coding environment of choice with little pain or process.

Perhaps most importantly, TypeScript is a big part of Angular, which means we'll continue to see it well into the future. The more we know about it and how it works, the better equipped we'll be to deal with it when it arrives as a fully-fledged mainstream alternative to JavaScript.

Getting Past Hello World in Angular

Jason Aden

Chapter

4

This article is part of a web development series from Microsoft. Thank you for supporting the partners who make SitePoint possible.

So you've been through the basic Angular application and now you want a bit more. If you've been reading about Angular you've undoubtedly seen what might look like really odd syntax in templates. You may have heard something about an overhaul to dependency injection. Or maybe some of the features of ES7 (or ES2016, the version of JavaScript planned to come out next year) such as Decorators and Observables.

This chapter will give you a quick introduction to these concepts and how they apply to your Angular applications. We won't dive deep into these topics yet, but look for later posts that cover each area in more detail.

So, let's begin at the beginning—setting up an environment.

Environment Setup

Angular source is written in TypeScript, a superset of JavaScript that allows for type definitions to be applied to your JavaScript code (you can find more information about TypeScript at www.typescriptlang.org).

In order to use TypeScript, we need to install the compiler which runs on top of Node.js. I'm using Node 0.12.6 and NPM 2.9.1. TypeScript should be installed automatically by NPM, but you can also install TypeScript globally using:

```
npm install -g typescript
# Test using
tsc -v
```

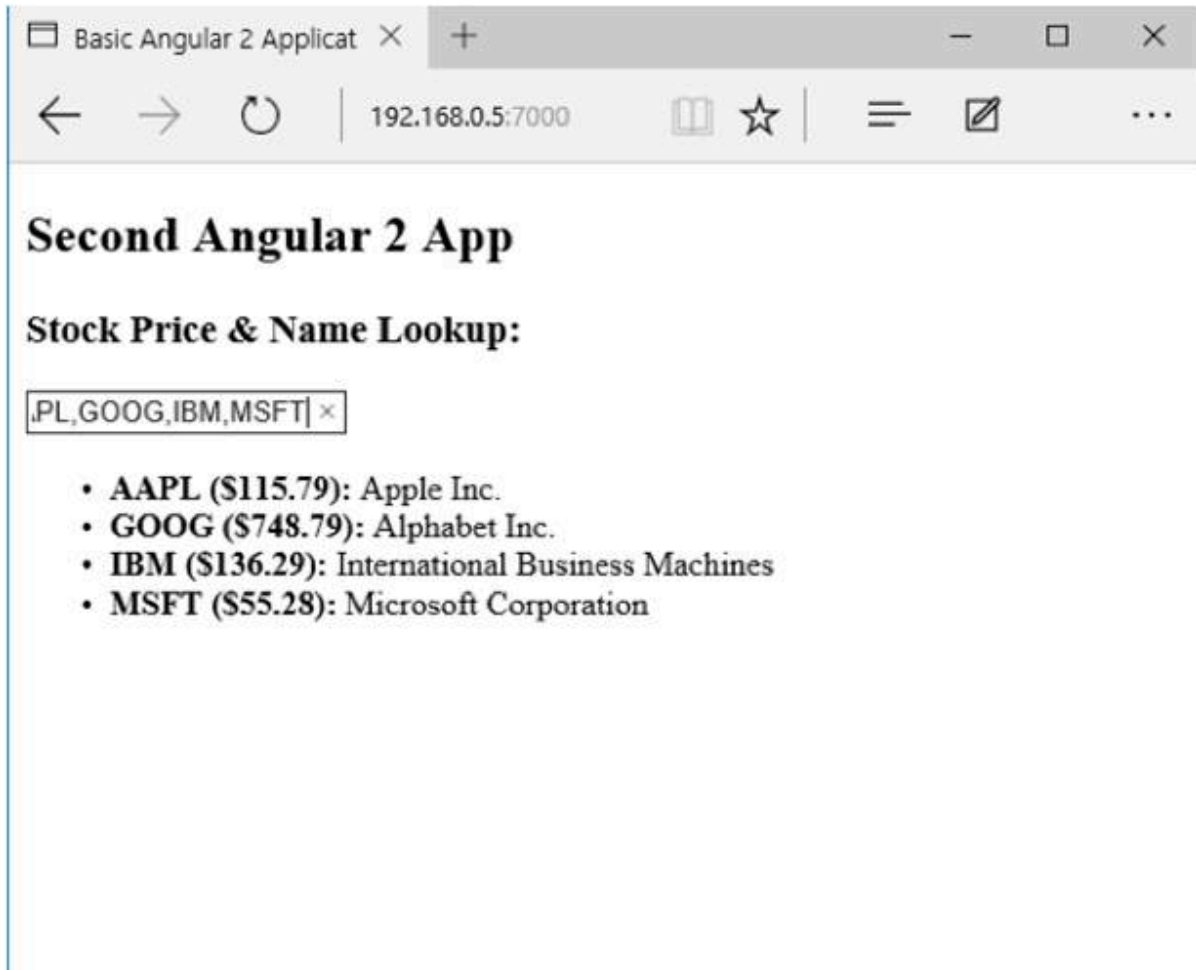
We are also using Gulp to build the project. If you don't have Gulp installed globally, use this command:

```
npm install -g gulp-cli
```

Once you have Node installed, use these instructions to set up and start your simple application:

```
git clone https://github.com/DevelopIntelligenceBoulder/second-angular-app
cd second-angular-app
npm install
gulp go
```

In a browser, navigate to <http://localhost:8080> (or the deployed version at <http://secondangular2.azurewebsites.net/>). You should see an extremely basic application like this:



4-1. Basic Angular application

This is a simple application that takes comma-separated ticker symbols as input and retrieves data including price and the stock name, displaying them in a simple list. This lacks styling, but we can use it to demonstrate a few ways to think about Angular applications.

Components

Angular applications are built using components. But what is a component? How do we set up a component in Angular?

At a high level, a component is a set of functionalities grouped together including

a view, styles (if applicable) and a controller class that manages the functionality of the component. If you're familiar with AngularJS, a component is basically a directive with a template and a controller. In fact, in Angular a component is just a special type of directive encapsulating a reusable UI building block (minimally a controller class and a view).

We will look at the pieces that make up a component by examining the StockSearch component. In our simple application, the StockSearch component displays the input box and includes a child component called StockList that renders the data returned from the API.

Decorators

Decorators are a new feature in ES7 and you will see them identified in source code by a leading "@" symbol. Decorators are used to provide Angular with metadata about a class (and sometimes methods or properties) so they can be wired into the Angular framework. Decorators always come before the class or property they are decorating, so you will see a codeblock like this:

```
@Component({  
  // ...  
})  
class MyClass {}
```

4-2. Decorators

This block uses the component decorator to tell Angular that `MyClass` is an Angular component.



Decorators vs Annotations

For an in-depth discussion of decorators and annotations (a special type of decorator used in Angular) see [Pascal Precht's post on the difference between annotations and decorators](#).

Examining a Component

Let's get into some code. Below is the StockSearch component, responsible for taking user input, calling a service to find stock data, and passing the data to the child StockList component. We will use this sample to talk about key parts of building an Angular application:

```

import {Component, View} from 'angular2/core'
import {StockList} from './stockList'
import {StocksService} from '../services/stocks'

@Component({
  selector: 'StockSearch',
  providers: [StocksService]
})
@View({
  template: `
    <section>
    <h3>Stock Price & Name Lookup:</h3>
    <form (submit)="doSearch()">
    <input [(ngModel)]="searchText"/>
    </form>
    <StockList [stocks]="stocks"></StockList>
    </section>
    `,
  directives: [StockList]
})
export class StockSearch {
  searchText: string;
  stocks: Object[];

```

```
constructor(public stockService:StocksService) {}

doSearch() {
    this.stockService.snapshot(this.searchText).subscribe(
        (data) => {this.stocks= data},
        (err) => {console.log('error!', err)}
    );
}
}
```

(Screenshots in this article are from [Visual Studio Code in Mac](#).)



Using TypeScript

While I'm relatively new to TypeScript, I've been enjoying it while writing Angular2 applications. TypeScript's ability to present type information in your IDE or text editor is extremely helpful, especially when working with something new like Angular2.

I've been pleasantly surprised with how well Visual Studio Code integrates with TypeScript. In fact, it's written in TypeScript and can parse it by default (no plugins required). When coding in Angular2 I found myself referring to the documentation frequently until I found the "Peek Definition" feature. This feature allows you to highlight any variable and use a keyboard shortcut (Opt-F12 on Mac) to look at the source of that variable. Since Angular2's documentation exists in the source, I find I rarely need to go to the online documentation. Using Peek Definition on the Component decorator looks like this:

```

app.ts src/app
1 //our root app component
2 import {Component, View} from 'angular2/angular2'

metadata.d.ts node_modules/angular2/src/core
511 export declare var Component: ComponentFactory;
512 /**
513  * Directives allow you to attach behavior to elements in the DOM.
514  *
515  * {@link DirectiveMetadata}s with an embedded view are called {@link ComponentMet
516  *
517  * A directive consists of a single directive annotation and a controller class. W
518  * directive's 'selector' matches
519  * elements in the DOM, the following steps occur:
520  *
521  * 1. For each directive, the 'ElementInjector' attempts to resolve the directive'
522  * arguments.
523  * 2. Angular instantiates directives for each matched element using 'ElementInjec
524  * depth-first order,
525  * as declared in the HTML.
526  *
527  *
528  *
529  *
530  *
531  *
532  *
533  *
534  *
535  *
536  *
537  *
538  *
539  *
540  *
541  *
542  *
543  *
544  *
545  *
546  *
547  *
548  *
549  *
550  *
551  *
552  *
553  *
554  *
555  *
556  *
557  *
558  *
559  *
560  *
561  *
562  *
563  *
564  *
565  *
566  *
567  *
568  *
569  *
570  *
571  *
572  *
573  *
574  *
575  *
576  *
577  *
578  *
579  *
580  *
581  *
582  *
583  *
584  *
585  *
586  *
587  *
588  *
589  *
590  *
591  *
592  *
593  *
594  *
595  *
596  *
597  *
598  *
599  *
600  *
601  *
602  *
603  *
604  *
605  *
606  *
607  *
608  *
609  *
610  *
611  *
612  *
613  *
614  *
615  *
616  *
617  *
618  *
619  *
620  *
621  *
622  *
623  *
624  *
625  *
626  *
627  *
628  *
629  *
630  *
631  *
632  *
633  *
634  *
635  *
636  *
637  *
638  *
639  *
640  *
641  *
642  *
643  *
644  *
645  *
646  *
647  *
648  *
649  *
650  *
651  *
652  *
653  *
654  *
655  *
656  *
657  *
658  *
659  *
660  *
661  *
662  *
663  *
664  *
665  *
666  *
667  *
668  *
669  *
670  *
671  *
672  *
673  *
674  *
675  *
676  *
677  *
678  *
679  *
680  *
681  *
682  *
683  *
684  *
685  *
686  *
687  *
688  *
689  *
690  *
691  *
692  *
693  *
694  *
695  *
696  *
697  *
698  *
699  *
700  *
701  *
702  *
703  *
704  *
705  *
706  *
707  *
708  *
709  *
710  *
711  *
712  *
713  *
714  *
715  *
716  *
717  *
718  *
719  *
720  *
721  *
722  *
723  *
724  *
725  *
726  *
727  *
728  *
729  *
730  *
731  *
732  *
733  *
734  *
735  *
736  *
737  *
738  *
739  *
740  *
741  *
742  *
743  *
744  *
745  *
746  *
747  *
748  *
749  *
750  *
751  *
752  *
753  *
754  *
755  *
756  *
757  *
758  *
759  *
760  *
761  *
762  *
763  *
764  *
765  *
766  *
767  *
768  *
769  *
770  *
771  *
772  *
773  *
774  *
775  *
776  *
777  *
778  *
779  *
780  *
781  *
782  *
783  *
784  *
785  *
786  *
787  *
788  *
789  *
790  *
791  *
792  *
793  *
794  *
795  *
796  *
797  *
798  *
799  *
800  *
801  *
802  *
803  *
804  *
805  *
806  *
807  *
808  *
809  *
810  *
811  *
812  *
813  *
814  *
815  *
816  *
817  *
818  *
819  *
820  *
821  *
822  *
823  *
824  *
825  *
826  *
827  *
828  *
829  *
830  *
831  *
832  *
833  *
834  *
835  *
836  *
837  *
838  *
839  *
840  *
841  *
842  *
843  *
844  *
845  *
846  *
847  *
848  *
849  *
850  *
851  *
852  *
853  *
854  *
855  *
856  *
857  *
858  *
859  *
860  *
861  *
862  *
863  *
864  *
865  *
866  *
867  *
868  *
869  *
870  *
871  *
872  *
873  *
874  *
875  *
876  *
877  *
878  *
879  *
880  *
881  *
882  *
883  *
884  *
885  *
886  *
887  *
888  *
889  *
890  *
891  *
892  *
893  *
894  *
895  *
896  *
897  *
898  *
899  *
900  *
901  *
902  *
903  *
904  *
905  *
906  *
907  *
908  *
909  *
910  *
911  *
912  *
913  *
914  *
915  *
916  *
917  *
918  *
919  *
920  *
921  *
922  *
923  *
924  *
925  *
926  *
927  *
928  *
929  *
930  *
931  *
932  *
933  *
934  *
935  *
936  *
937  *
938  *
939  *
940  *
941  *
942  *
943  *
944  *
945  *
946  *
947  *
948  *
949  *
950  *
951  *
952  *
953  *
954  *
955  *
956  *
957  *
958  *
959  *
960  *
961  *
962  *
963  *
964  *
965  *
966  *
967  *
968  *
969  *
970  *
971  *
972  *
973  *
974  *
975  *
976  *
977  *
978  *
979  *
980  *
981  *
982  *
983  *
984  *
985  *
986  *
987  *
988  *
989  *
990  *
991  *
992  *
993  *
994  *
995  *
996  *
997  *
998  *
999  *
1000  *
1001  *
1002  *
1003  *
1004  *
1005  *
1006  *
1007  *
1008  *
1009  *
1010  *
1011  *
1012  *
1013  *
1014  *
1015  *
1016  *
1017  *
1018  *
1019  *
1020  *
1021  *
1022  *
1023  *
1024  *
1025  *
1026  *
1027  *
1028  *
1029  *
1030  *
1031  *
1032  *
1033  *
1034  *
1035  *
1036  *
1037  *
1038  *
1039  *
1040  *
1041  *
1042  *
1043  *
1044  *
1045  *
1046  *
1047  *
1048  *
1049  *
1050  *
1051  *
1052  *
1053  *
1054  *
1055  *
1056  *
1057  *
1058  *
1059  *
1060  *
1061  *
1062  *
1063  *
1064  *
1065  *
1066  *
1067  *
1068  *
1069  *
1070  *
1071  *
1072  *
1073  *
1074  *
1075  *
1076  *
1077  *
1078  *
1079  *
1080  *
1081  *
1082  *
1083  *
1084  *
1085  *
1086  *
1087  *
1088  *
1089  *
1090  *
1091  *
1092  *
1093  *
1094  *
1095  *
1096  *
1097  *
1098  *
1099  *
1100  *
1101  *
1102  *
1103  *
1104  *
1105  *
1106  *
1107  *
1108  *
1109  *
1110  *
1111  *
1112  *
1113  *
1114  *
1115  *
1116  *
1117  *
1118  *
1119  *
1120  *
1121  *
1122  *
1123  *
1124  *
1125  *
1126  *
1127  *
1128  *
1129  *
1130  *
1131  *
1132  *
1133  *
1134  *
1135  *
1136  *
1137  *
1138  *
1139  *
1140  *
1141  *
1142  *
1143  *
1144  *
1145  *
1146  *
1147  *
1148  *
1149  *
1150  *
1151  *
1152  *
1153  *
1154  *
1155  *
1156  *
1157  *
1158  *
1159  *
1160  *
1161  *
1162  *
1163  *
1164  *
1165  *
1166  *
1167  *
1168  *
1169  *
1170  *
1171  *
1172  *
1173  *
1174  *
1175  *
1176  *
1177  *
1178  *
1179  *
1180  *
1181  *
1182  *
1183  *
1184  *
1185  *
1186  *
1187  *
1188  *
1189  *
1190  *
1191  *
1192  *
1193  *
1194  *
1195  *
1196  *
1197  *
1198  *
1199  *
1200  *
1201  *
1202  *
1203  *
1204  *
1205  *
1206  *
1207  *
1208  *
1209  *
1210  *
1211  *
1212  *
1213  *
1214  *
1215  *
1216  *
1217  *
1218  *
1219  *
1220  *
1221  *
1222  *
1223  *
1224  *
1225  *
1226  *
1227  *
1228  *
1229  *
1230  *
1231  *
1232  *
1233  *
1234  *
1235  *
1236  *
1237  *
1238  *
1239  *
1240  *
1241  *
1242  *
1243  *
1244  *
1245  *
1246  *
1247  *
1248  *
1249  *
1250  *
1251  *
1252  *
1253  *
1254  *
1255  *
1256  *
1257  *
1258  *
1259  *
1260  *
1261  *
1262  *
1263  *
1264  *
1265  *
1266  *
1267  *
1268  *
1269  *
1270  *
1271  *
1272  *
1273  *
1274  *
1275  *
1276  *
1277  *
1278  *
1279  *
1280  *
1281  *
1282  *
1283  *
1284  *
1285  *
1286  *
1287  *
1288  *
1289  *
1290  *
1291  *
1292  *
1293  *
1294  *
1295  *
1296  *
1297  *
1298  *
1299  *
1300  *
1301  *
1302  *
1303  *
1304  *
1305  *
1306  *
1307  *
1308  *
1309  *
1310  *
1311  *
1312  *
1313  *
1314  *
1315  *
1316  *
1317  *
1318  *
1319  *
1320  *
1321  *
1322  *
1323  *
1324  *
1325  *
1326  *
1327  *
1328  *
1329  *
1330  *
1331  *
1332  *
1333  *
1334  *
1335  *
1336  *
1337  *
1338  *
1339  *
1340  *
1341  *
1342  *
1343  *
1344  *
1345  *
1346  *
1347  *
1348  *
1349  *
1350  *
1351  *
1352  *
1353  *
1354  *
1355  *
1356  *
1357  *
1358  *
1359  *
1360  *
1361  *
1362  *
1363  *
1364  *
1365  *
1366  *
1367  *
1368  *
1369  *
1370  *
1371  *
1372  *
1373  *
1374  *
1375  *
1376  *
1377  *
1378  *
1379  *
1380  *
1381  *
1382  *
1383  *
1384  *
1385  *
1386  *
1387  *
1388  *
1389  *
1390  *
1391  *
1392  *
1393  *
1394  *
1395  *
1396  *
1397  *
1398  *
1399  *
1400  *
1401  *
1402  *
1403  *
1404  *
1405  *
1406  *
1407  *
1408  *
1409  *
1410  *
1411  *
1412  *
1413  *
1414  *
1415  *
1416  *
1417  *
1418  *
1419  *
1420  *
1421  *
1422  *
1423  *
1424  *
1425  *
1426  *
1427  *
1428  *
1429  *
1430  *
1431  *
1432  *
1433  *
1434  *
1435  *
1436  *
1437  *
1438  *
1439  *
1440  *
1441  *
1442  *
1443  *
1444  *
1445  *
1446  *
1447  *
1448  *
1449  *
1450  *
1451  *
1452  *
1453  *
1454  *
1455  *
1456  *
1457  *
1458  *
1459  *
1460  *
1461  *
1462  *
1463  *
1464  *
1465  *
1466  *
1467  *
1468  *
1469  *
1470  *
1471  *
1472  *
1473  *
1474  *
1475  *
1476  *
1477  *
1478  *
1479  *
1480  *
1481  *
1482  *
1483  *
1484  *
1485  *
1486  *
1487  *
1488  *
1489  *
1490  *
1491  *
1492  *
1493  *
1494  *
1495  *
1496  *
1497  *
1498  *
1499  *
1500  *
1501  *
1502  *
1503  *
1504  *
1505  *
1506  *
1507  *
1508  *
1509  *
1510  *
1511  *
1512  *
1513  *
1514  *
1515  *
1516  *
1517  *
1518  *
1519  *
1520  *
1521  *
1522  *
1523  *
1524  *
1525  *
1526  *
1527  *
1528  *
1529  *
1530  *
1531  *
1532  *
1533  *
1534  *
1535  *
1536  *
1537  *
1538  *
1539  *
1540  *
1541  *
1542  *
1543  *
1544  *
1545  *
1546  *
1547  *
1548  *
1549  *
1550  *
1551  *
1552  *
1553  *
1554  *
1555  *
1556  *
1557  *
1558  *
1559  *
1560  *
1561  *
1562  *
1563  *
1564  *
1565  *
1566  *
1567  *
1568  *
1569  *
1570  *
1571  *
1572  *
1573  *
1574  *
1575  *
1576  *
1577  *
1578  *
1579  *
1580  *
1581  *
1582  *
1583  *
1584  *
1585  *
1586  *
1587  *
1588  *
1589  *
1590  *
1591  *
1592  *
1593  *
1594  *
1595  *
1596  *
1597  *
1598  *
1599  *
1600  *
1601  *
1602  *
1603  *
1604  *
1605  *
1606  *
1607  *
1608  *
1609  *
1610  *
1611  *
1612  *
1613  *
1614  *
1615  *
1616  *
1617  *
1618  *
1619  *
1620  *
1621  *
1622  *
1623  *
1624  *
1625  *
1626  *
1627  *
1628  *
1629  *
1630  *
1631  *
1632  *
1633  *
1634  *
1635  *
1636  *
1637  *
1638  *
1639  *
1640  *
1641  *
1642  *
1643  *
1644  *
1645  *
1646  *
1647  *
1648  *
1649  *
1650  *
1651  *
1652  *
1653  *
1654  *
1655  *
1656  *
1657  *
1658  *
1659  *
1660  *
1661  *
1662  *
1663  *
1664  *
1665  *
1666  *
1667  *
1668  *
1669  *
1670  *
1671  *
1672  *
1673  *
1674  *
1675  *
1676  *
1677  *
1678  *
1679  *
1680  *
1681  *
1682  *
1683  *
1684  *
1685  *
1686  *
1687  *
1688  *
1689  *
1690  *
1691  *
1692  *
1693  *
1694  *
1695  *
1696  *
1697  *
1698  *
1699  *
1700  *
1701  *
1702  *
1703  *
1704  *
1705  *
1706  *
1707  *
1708  *
1709  *
1710  *
1711  *
1712  *
1713  *
1714  *
1715  *
1716  *
1717  *
1718  *
1719  *
1720  *
1721  *
1722  *
1723  *
1724  *
1725  *
1726  *
1727  *
1728  *
1729  *
1730  *
1731  *
1732  *
1733  *
1734  *
1735  *
1736  *
1737  *
1738  *
1739  *
1740  *
1741  *
1742  *
1743  *
1744  *
1745  *
1746  *
1747  *
1748  *
1749  *
1750  *
1751  *
1752  *
1753  *
1754  *
1755  *
1756  *
1757  *
1758  *
1759  *
1760  *
1761  *
1762  *
1763  *
1764  *
1765  *
1766  *
1767  *
1768  *
1769  *
1770  *
1771  *
1772  *
1773  *
1774  *
1775  *
1776  *
1777  *
1778  *
1779  *
1780  *
1781  *
1782  *
1783  *
1784  *
1785  *
1786  *
1787  *
1788  *
1789  *
1790  *
1791  *
1792  *
1793  *
1794  *
1795  *
1796  *
1797  *
1798  *
1799  *
1800  *
1801  *
1802  *
1803  *
1804  *
1805  *
1806  *
1807  *
1808  *
1809  *
1810  *
1811  *
1812  *
1813  *
1814  *
1815  *
1816  *
1817  *
1818  *
1819  *
1820  *
1821  *
1822  *
1823  *
1824  *
1825  *
1826  *
1827  *
1828  *
1829  *
1830  *
1831  *
1832  *
1833  *
1834  *
1835  *
1836  *
1837  *
1838  *
1839  *
1840  *
1841  *
1842  *
1843  *
1844  *
1845  *
1846  *
1847  *
1848  *
1849  *
1850  *
1851  *
1852  *
1853  *
1854  *
1855  *
1856  *
1857  *
1858  *
1859  *
1860  *
1861  *
1862  *
1863  *
1864  *
1865  *
1866  *
1867  *
1868  *
1869  *
1870  *
1871  *
1872  *
1873  *
1874  *
1875  *
1876  *
1877  *
1878  *
1879  *
1880  *
1881  *
1882  *
1883  *
1884  *
1885  *
1886  *
1887  *
1888  *
1889  *
1890  *
1891  *
1892  *
1893  *
1894  *
1895  *
1896  *
1897  *
1898  *
1899  *
1900  *
1901  *
1902  *
1903  *
1904  *
1905  *
1906  *
1907  *
1908  *
1909  *
1910  *
1911  *
1912  *
1913  *
1914  *
1915  *
1916  *
1917  *
1918  *
1919  *
1920  *
1921  *
1922  *
1923  *
1924  *
1925  *
1926  *
1927  *
1928  *
1929  *
1930  *
1931  *
1932  *
1933  *
1934  *
1935  *
1936  *
1937  *
1938  *
1939  *
1940  *
1941  *
1942  *
1943  *
1944  *
1945  *
1946  *
1947  *
1948  *
1949  *
1950  *
1951  *
1952  *
1953  *
1954  *
1955  *
1956  *
1957  *
1958  *
1959  *
1960  *
1961  *
1962  *
1963  *
1964  *
1965  *
1966  *
1967  *
1968  *
1969  *
1970  *
1971  *
1972  *
1973  *
1974  *
1975  *
1976  *
1977  *
1978  *
1979  *
1980  *
1981  *
1982  *
1983  *
1984  *
1985  *
1986  *
1987  *
1988  *
1989  *
1990  *
1991  *
1992  *
1993  *
1994  *
1995  *
1996  *
1997  *
1998  *
1999  *
2000  *
2001  *
2002  *
2003  *
2004  *
2005  *
2006  *
2007  *
2008  *
2009  *
2010  *
2011  *
2012  *
2013  *
2014  *
2015  *
2016  *
2017  *
2018  *
2019  *
2020  *
2021  *
2022  *
2023  *
2024  *
2025  *
2026  *
2027  *
2028  *
2029  *
2030  *
2031  *
2032  *
2033  *
2034  *
2035  *
2036  *
2037  *
2038  *
2039  *
2040  *
2041  *
2042  *
2043  *
2044  *
2045  *
2046  *
2047  *
2048  *
2049  *
2050  *
2051  *
2052  *
2053  *
2054  *
2055  *
2056  *
2057  *
2058  *
2059  *
2060  *
2061  *
2062  *
2063  *
2064  *
2065  *
2066  *
2067  *
2068  *
2069  *
2070  *
2071  *
2072  *
2073  *
2074  *
2075  *
2076  *
2077  *
2078  *
2079  *
2080  *
2081  *
2082  *
2083  *
2084  *
2085  *
2086  *
2087  *
2088  *
2089  *
2090  *
2091  *
2092  *
2093  *
2094  *
2095  *
2096  *
2097  *
2098  *
2099  *
2100  *
2101  *
2102  *
2103  *
2104  *
2105  *
2106  *
2107  *
2108  *
2109  *
2110  *
2111  *
2112  *
2113  *
2114  *
2115  *
2116  *
2117  *
2118  *
2119  *
2120  *
2121  *
2122  *
2123  *
2124  *
2125  *
2126  *
2127  *
2128  *
2129  *
2130  *
2131  *
2132  *
2133  *
2134  *
2135  *
2136  *
2137  *
2138  *
2139  *
2140  *
2141  *
2142  *
2143  *
2144  *
2145  *
2146  *
2147  *
2148  *
2149  *
2150  *
2151  *
2152  *
2153  *
2154  *
2155  *
2156  *
2157  *
2158  *
2159  *
2160  *
2161  *
2162  *
2163  *
2164  *
2165  *
2166  *
2167  *
2168  *
2169  *
2170  *
2171  *
2172  *
2173  *
2174  *
2175  *
2176  *
2177  *
2178  *
2179  *
2180  *
2181  *
2182  *
2183  *
2184  *
2185  *
2186  *
2187  *
2188  *
2189  *
2190  *
2191  *
2192  *
2193  *
2194  *
2195  *
2196  *
2197  *
2198  *
2199  *
2200  *
2201  *
2202  *
2203  *
2204  *
2205  *
2206  *
2207  *
2208  *
2209  *
2210  *
2211  *
2212  *
2213  *
2214  *
2215  *
2216  *
2217  *
2218  *
2219  *
2220  *
2221  *
2222  *
2223  *
2224  *
2225  *
2226  *
2227  *
2228  *
2229  *
2230  *
2231  *
2232  *
2233  *
2234  *
2235  *
2236  *
2237  *
2238  *
2239  *
2240  *
2241  *
2242  *
2243  *
2244  *
2245  *
2246  *
2247  *
2248  *
2249  *
2250  *
2251  *
2252  *
2253  *
2254  *
2255  *
2256  *
2257  *
2258  *
2259  *
2260  *
2261  *
2262  *
2263  *
```

import statements

The import statements at the top of the file bring in dependencies used by this component. When you create a component, you will almost always use `import {Component, View} from 'angular2/core'` at the top. This line brings in the component and view decorators.

Angular also requires that you explicitly tell the framework about any children components (or directives) you want to use. So the next line `(import {StockList} from './stockList')` pulls in the StockList component.

Similarly, we identify any services we will need. In this case we want to be able to request data using the StockService, so the last import pulls it in.

selector and template

The `selector` and `template` properties passed to the component and view decorators are the only two configuration options required to build an Angular component. Selector tells Angular how to find this component in HTML and it understands CSS selector syntax. Unlike regular HTML, Angular templates are case sensitive. So where in AngularJS we used hyphens to make camelCase into kebab-case, we don't need to do this in Angular. This means we can now use uppercase letters to start our component selectors, which distinguishes them from standard HTML. So:

```
selector: 'StockSearch' // matches <StockSearch></StockSearch>
selector: '.stockSearch' '.stockSearch' // matches <div class="stockSearch">
selector: '[stockSearch]' // matches <div stockSearch>
```

When the Angular compiler comes across HTML that matches one of these selectors, it creates an instance of the component class and renders the contents of the template property. You can use either `template` to create an inline template, or `templateUrl` to pass a URL, which contains the HTML template.

Dependency Injection ("providers" and "directives")

If you're familiar with AngularJS you know it uses dependency injection to pass things like services, factories and values into other services, factories and controllers. Angular also has dependency injection, but this version is a bit more robust.

In the case of the component above, the class (controller) is called `StockSearch`. `StockSearch` needs to use `StockService` to make a call to get stock data. Consider the following abbreviated code snippet:

```
import{StockService} from '../services/stocks'

@Component({
  selector: 'StockSearch',
  providers: [StockService]
})
export class StockSearch {
  constructor(public stockService:StockService) {}
}
```

As with other classes, the `import` statement makes the `StockService` class available. We then pass it into the `providers` property passed to the `Component` decorator, which alerts Angular that we want to use dependency injection to create an instance of the `StockService` when it's passed to the `StockSearch` constructor.

The constructor looks pretty bare-bones, but there's actually quite a bit happening in this single line.

public keyword

The `public` keyword tells TypeScript to put the `stockService` (camelCase, as opposed to the class name which is *PascalCase*) variable onto the instance of the `StockSearch` component. `StockSearch`'s methods will reference it as

```
this.stockService .
```

Type declaration

After `public stockService` we have `:StockService`, which tells the compiler that the variable `stockService` will be an instance of the `StockService` class. Because we used the component decorator, type declarations on the constructor cause Angular's dependency injection module to create an instance of the `StockService` class and pass it to the `StockSearch` constructor. If you're familiar with AngularJS, an analogous line of code would look like this:

```
app.controller('StockSearch', ['StockService', StockSearch]);

function StockSearch(StockService) {
  this.stockService = StockService;
})
```

4-4. Type declaration

One key difference between the AngularJS and Angular dependency injection systems is that in AngularJS there's just one large global namespace for all dependencies in your app. If you register a controller by calling `app.controller('MyController', ...)` in two different places in your code, the second one loaded will overwrite the first. (Note that this overwrite issue doesn't apply to directives in AngularJS. Registering a directive by calling `app.directive('myDirective', ...)` in multiple places will add to the behavior of the earlier definition, not overwrite it.)

In Angular the confusion is resolved by explicitly defining which directives are used in each view. So `directives: [StockList]` tells Angular to look for the `StockList` component inside our view. Without that property defined, Angular wouldn't do anything with the "`<StockList>`" HTML tag.

Properties and Events

Looking at the `template` passed to the View decorator, we see a new syntax where some attributes are surrounded by parentheses, some by brackets, and some by both.

Parentheses

Surrounding an attribute with parentheses tells Angular to listen for an event by the attribute name. So `<form (submit) = "doSearch()">` tells Angular to listen for the `submit` event on the `form` component, and when a submit occurs the `doSearch` method of the current component should be run (in this case, `StockSearch` is the current component).

```

<!-- Angular 2 capture submit -->
<form (submit)="doSearch()">
  <!-- Form code here -->
</form>

<!-- Angular 1 capture submit -->
<form ng-submit="doSearch()">
  <!-- Form code here -->
</form>

```

4-5. Properties and Events (Parentheses)

Square Brackets

Surrounding an attribute with square brackets tells Angular to parse the attribute value as an expression and assign the result of the expression to a property on the target component. So `<StockList [stocks] = "stocks"></StockList>` will look for a `stocks` variable on the current component (`StockSearch`) and pass its value to the `StockList` component in a property also named `"stocks"`. In AngularJS this required configuration using the Directive Definition Object, but looking just at the HTML there was no indication of how the attribute would be used.

```
<!-- Angular we know [stocks] causes "stocks" to
      parsed and passed to StockList component -->
<StockList[stocks] = "stocks"></StockList>

<!-- AngularJS doesn't use brackets. Looking just at
      the HTML we don't know how the directive is using
      the stocks attribute -->
<stock-list stocks = "stocks"></stock-list>
```

Square Brackets and Parentheses

This is a special syntax available to the `ngModel` directive. It tells Angular to create a two-way binding. So `<input [(ngModel)] = "searchText">` wires up an input element where text entered into the input gets applied to the `searchText` property of the `StockSearch` component. Similarly, changes to `this.searchText` inside the `StockSearch` instance cause an update to the value of the input.

```
<!-- Angular requires both event and attribute
      bindings to be explicitly clear we want 2-way binding -->
<input [(ngModel)] = "searchText">

<!-- AngularJS looks simpler without the [(...)]. -->
<input ng-model = "searchText">
```

Services

A more in-depth explanation of services will be the subject of a future post, but our application defines a `StockService` that's used to make an HTTP call to retrieve data. So let's take a look at the service and walk through the pieces:

```
//a simple service
import {Injectable} from 'angular2/core';
import {SearchParams} from 'angular2/http';
```

```

@Injectable()
export class StocksService {
  // TS shortcut "public" to put http on this
  constructor(public http:Http) {}

  snapshot(symbols:string): any {
    let params = new URLSearchParams();
    params.set('symbols', symbols);

    return this.http.get("/api/snapshot", {search: params})
      .map(res => res.json()) // convert to JSON
      .map(x => x.filter(y => y.name)); // Remove stocks w/no name
  }
}

```

@Injectable() decorator

In Angular we use the Injectable decorator to let Angular know a class should be registered with dependency injection. Without this, the providers property we used in the StockSearch component wouldn't have worked, and dependency injection wouldn't have created an instance of the service to pass into the controller. Therefore, if you want a class to be registered with dependency injection, use the Injectable decorator.

HTTP service

We'll deep-dive into Angular's HTTP library in the future, but the high-level summary is that it lets you call all your basic HTTP methods. Similar to the way we made StockService available to the StockSearch constructor, we are adding "http" on the "this" object using `public http:Http`.

If you take a look at the "snapshot" method, you'll see we call `/api/snapshot`, then pass a configuration object with the search params. This is pretty straightforward, but at the bottom pay attention to `.map(...)`. In AngularJS (as in most modern HTTP libraries), an HTTP call returns a "promise." But in Angular

we get an Observable.

An observable is like a promise but it can be resolved multiple times. There's a lot to know about observables and we'll cover them in upcoming posts, but for more information on observables, set aside some time to go through [this post](#) and exercises in order to get up to speed.

Tree of Components

We've looked at one component and a service it accesses. Angular applications are built by constructing a tree of components, starting with a root application component. The application component should contain very little logic, as the primary purpose is to lay out the top-level pieces of your application.

```
//our root app component
import {Component, View} from 'angular2/core'
import {S'angular2/core'm './components/stockSearch';

@Component({
  selector: 'App'
})
@View({
  template: '
    <header>
    <h2>Second Angular App</h2>
    </header>
    <StockSearch></StockSearch>',
  directives: [StockSearch]
})
export class AppComponent {}
```

Now that we are familiar with the syntax of wiring up a component, we can see that the template outputs the header, followed by the StockSearch component. StockSearch is in the directive list, so Angular renders the component when it comes across the `<StockSearch>` tag.

That's about all there is to an application component; it is simply the root of our application. There can be only a single root in any Angular application, and the final piece of the puzzle is telling Angular to look for it.

Bootstrapping our Application

In an Angular application, we need to tell Angular when to start up. This is done using the `bootstrap` method, passing in our `AppComponent` along with other module dependencies. For those familiar with AngularJS, this is similar to the main `angular.module(name, [dependencies...])` construct.

```
//our root app component
import {bootstrap} from 'angular2/platform/browser';
import {HTTP_PROVIDERS} from 'angular2/http';
import 'rxjs/Rx';
import {AppComponent} from './app';

bootstrap(AppComponent, [HTTP_PROVIDERS])
  .catch(err =>console.error(err));
```

Notice the second argument is `[HTTP_PROVIDERS]`. This is a special variable that references the classes defining Angular's HTTP functionality, and it's needed to let Angular know we can use the HTTP classes in our application. The AngularJS equivalent looks something like this:

```
// JavaScript code to set up the app
angular.module('App', ['ngResource']);

<!-- Corresp 'App'g HTML tag pointing to "App" module -->
<div ng-app="App"></div>
```

Also notice we have `import 'rxjs/Rx'` at the top. This pulls in the RxJs library so we get access to methods on Observables. Without adding this line we wouldn't be able to run `.map()` on the return from `http.get()` method since the returned Observable wouldn't have the map method available.

Once the application is bootstrapped, Angular looks for our root component in our markup. Looking back at the `AppComponent`, the selector is set to `app`, so Angular will be looking for an element `<app>` and will render the `AppComponent` there:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Basic Angular Application Demonstration</title>
  </head>
  <body>
    <App>
      loading...
    </App>
    <script src="/lib/angular2-polyfills.js"></script>
    <script src="/lib/system.js"></script>
    <script src="/lib/Rx.js"></script>
    <script src="/lib/angular2.dev.js"></script>
    <script src="/lib/http.dev.js"></script>
    <script>
      System.config({
        //packages defines our app package
        packages: {'app': {defaultExtension: 'js'}}
      });

      System.import('app/bootstrap')
        .catch(console.error.bind(console));
    </script>

  </body>
</html>
'app'
```

Summary

When considering Angular applications, think about components and their children. At any point in the application you are examining some component, and that component knows what its children are. It's a top-down approach where at

each level you define compartmentalized functionality that eventually leads to a complete application.

Angular Components: Inputs and Outputs

David Aden

Chapter

5

In this article, we'll take a look a bit closer at Angular components — how they're defined, and how to get data into them and back out of them.

In the previous chapter we covered the basic idea of components and decorators, and have specifically seen the `@Component` and `@View` decorators used to build an Angular application. This article dives in a bit deeper. However, we can't cover everything about components in a single article, so future articles will take up other aspects of Angular components.



Example Code

The code for this article and the other articles in the series is available as in the [angular2-samples](#) repo. You can also see the samples running at: <http://angular2-samples.azurewebsites.net/>.

Although it's possible to write Angular applications in ECMAScript 5 (the most common version of JavaScript supported by browsers), we prefer to write in TypeScript. Angular itself is written in TypeScript and it helps us at development time and includes features that make it easier for us to define Angular components.

In particular, TypeScript supports decorators (sometimes referred to as “annotations”) which are used to declaratively add to or change an existing “thing”. For example, class decorators can add metadata to the class's constructor function or even alter how the class behaves. For more information on decorators and the types of things you can do with them, see the proposal for JavaScript decorators. Angular includes several decorators.

As we've covered in an earlier article, Angular components are the key building block for Angular applications. They include a view, defined with HTML and CSS, and an associated controller that implements functionality needed by the view. The controller has three major responsibilities:

- Manage the model, i.e. the application data used by the view
- Implement methods needed by the view for things like submitting data or hiding/showing sections of the UI
- Managing data related to the state of the view, such as which item in a list is currently selected.

Depending on your background, the above list might sound familiar. In fact, the Angular component controller sounds very much like the original definition of a view model as defined by John Gossman in 2005:

The term means “Model of a View”, and can be thought of as abstraction of the view, but it also provides a specialization of the Model that the View can use for data-binding. In this latter role the ViewModel contains data-transformers that convert Model types into View types, and it contains Commands the View can use to interact with the Model. — [Source](#) (captured 11/27/2015)

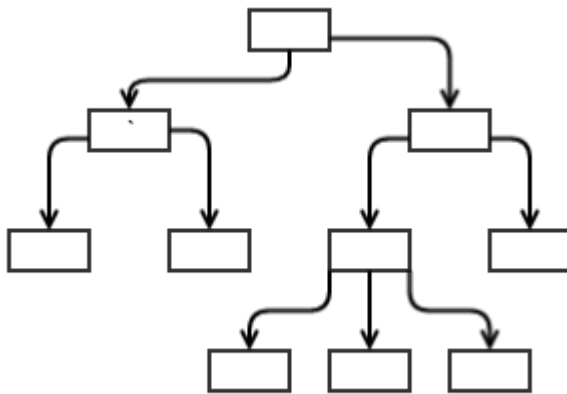
Because Angular components aren’t native JavaScript entities, Angular provides a way to define a component by pairing a constructor function with a view. You do this by defining a constructor function (in TypeScript it’s defined as a class) and using a decorator to associate your view with the constructor. The decorator can also set various configuration parameters for the component. This magic is accomplished using the `@Component` decorator we saw in the first article in this series.

Component Hierarchy

The above describes an individual component, but Angular applications are actually made up of a hierarchy of components – they begin with a root component that contains as descendants all the components used in the application. Angular components are intended to be self-contained, because we

want to encapsulate our component functions and we don't want other code to arbitrarily reach into our components to read or change properties. Also, we don't want our component to affect another component written by someone else. An obvious example is CSS: if we set CSS for one component, we don't want our CSS to "bleed out" to another components just as we don't want other CSS to "bleed in" to our component.

At the same time, components do need to exchange data. Angular components can receive data from their parent as long as the receiving component has specifically said it's willing to receive data. Similarly, components can send data to their parents by trigger an event the parent listens for. Let's look at how the component hierarchy behaves. To begin, we can draw it as follows:



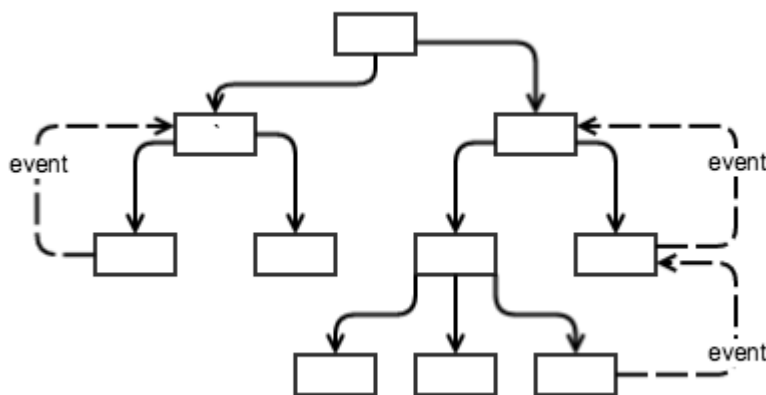
5-1. Angular components: Component Hierarchy behaviour

Each box is a component and technically this representation is called “graph” — a data structure consisting of nodes and connecting “edges.” The arrows represent the data flow from one component to another, and we can see that data flows in only one direction — from the top downwards to descendants. Also, note there are no paths that allow you to travel from one node, through other nodes and back to the one where you started. The official name for this kind of data structure is a “directed acyclic graph” — that is, it flows in only one direction and has no circular paths in it.

This kind of structure has some important features: it's predictable, it's simple to traverse, and it's easy to see what's impacted when a change is made. For Angular's purposes, when data changes in one node, it's easy to find the downstream nodes that could be affected.

A simple example of how this might be used is a table with rows containing customers and information about them, in which a table component contains multiple individual row components that represent each customer. The table component could manage a record set containing all the customers and pass the data on an individual customer to each of the row components it contains.

This works fine for simply displaying data, but in the real world data will need to flow the other way — back up the hierarchy — such as when a user edits a row. In that case, the row needs to tell the table component that the data for a row has changed so the change can be sent back to the server. The problem is that, as diagrammed above, data only flows down the hierarchy, not back up. To ensure we maintain the simplicity of one-way data flow down the hierarchy, Angular uses a different mechanism for sending data back up the hierarchy: events.



5-2. Angular components: One-way data flow down the hierarchy

Now, when a child component takes an action that a parent needs to know about, the child fires an event that's caught by the parent. The parent can take

any action it needs which might include updating data that will, through the usual one-way downwards data flow, update downstream components. By separating the downward flow of data from the upwards flow of data, things are kept simpler and data management performs well.

Angular Components: Inputs and Outputs

With that high-level look at Angular components under our belt, let's look at two properties that can be passed to the `@Component` decorator to implement the downward and upward flow of data: “inputs” and “outputs.” These have sometimes been confusing because, in earlier version of the Angular alpha, they were called “properties” (for “inputs”) and “events” (for “outputs”) and some developers were less than enthralled with the name change, though it does seem to make sense: <https://github.com/angular/angular/pull/4435>.

“Inputs”, as you might guess from the hierarchy discussion above, specifies which properties you can set on a component, whereas “outputs” identifies the events a component can fire to send information up the hierarchy to its parent.

```

• ParentComp.ts src/components/app/components
1 import {Component, View, Input, Output, EventEmitter} from 'angular2/core';
2
3 @Component({
4   selector: 'ParentComp',
5   inputs: ['myname'],
6   outputs: ['myevent']
7 })
8 @View({
9   template: `
10     <div (click)="fireMyEvent()">
11       Here is ParentComp and here's 'myname': {{myname}}
12     </div>
13   `
14 })
15
16 export class ParentComp {
17   public myname: String;
18   public myevent: EventEmitter = new EventEmitter();
19
20   constructor() {
21     console.log("ParentComp, myname not yet defined: ", this.myname);
22   }
23   fireMyEvent(evt) {
24     this.myevent.next(['abc', 'def']);
25   }
26 }
27

```

component-types* 1 0 Ln 19, Col 1 UTF-8 LF TypeScript

5-3. Angular components: Using inputs and outputs properties in the @Component decorator

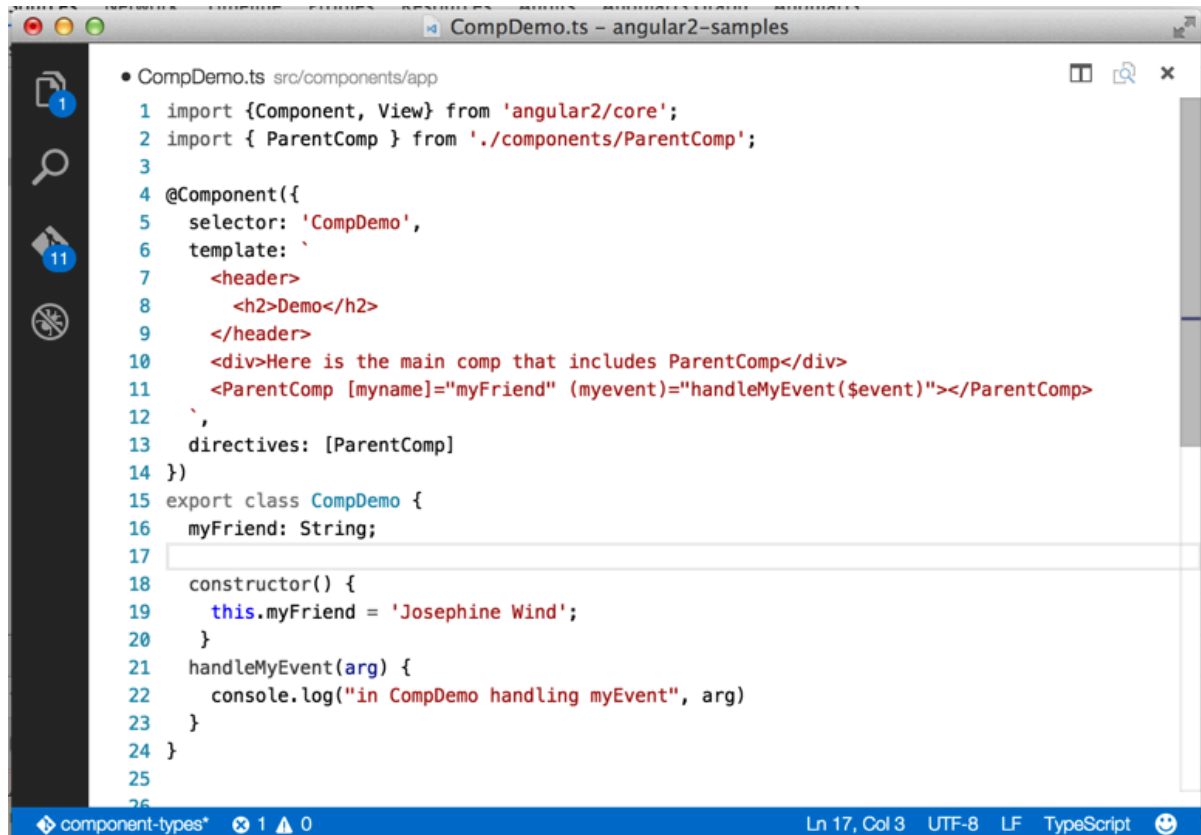
There are several things to notice with regards to inputs and outputs above:

- The “inputs” property passed to the `@Components` decorator lists “myname” as a component property that can receive data. We also declare “myname” as a public property within `ParentComp` class. If you don’t declare it, the TypeScript compiler might issue a warning.
- The “outputs” property lists “myevent” as a custom event that `ParentComp` can emit which its parent will be able to receive. Within the `ParentComp` class, “myevent” is declared as and set to be an `EventEmitter`. `EventEmitter` is a built-in class that ships with Angular that gives us methods for managing and firing custom events. Notice that we had to add `EventEmitter` to the import statement at the top of the file.
- This component displays the incoming “myname” in the view, but when we try

to access it in `ParentComp` constructor it's not yet defined. That's because input properties aren't available until the view has rendered, which happens after the constructor function runs.

- We added a “click” event handler to our template that invokes the `myeventEventEmitter`'s “`next()`” method and passes it the data we want to send with the event. This is the standard pattern for sending data up the component hierarchy — using “EventEmitter” to call the “`next()`” method.

Now that we've looked at how to define “inputs” and “outputs” in a component, let's see how to use them. The template for the `CompDemo` component uses the `ParentComp` component:



```

1 import {Component, View} from 'angular2/core';
2 import { ParentComp } from './components/ParentComp';
3
4 @Component({
5   selector: 'CompDemo',
6   template: `
7     <header>
8       <h2>Demo</h2>
9     </header>
10    <div>Here is the main comp that includes ParentComp</div>
11    <ParentComp [myname]="myFriend" (myevent)="handleMyEvent($event)"></ParentComp>
12  `,
13   directives: [ParentComp]
14 })
15 export class CompDemo {
16   myFriend: String;
17
18   constructor() {
19     this.myFriend = 'Josephine Wind';
20   }
21   handleMyEvent(arg) {
22     console.log("in CompDemo handling myEvent", arg)
23   }
24 }
25
26

```

5-4. Angular 2 components: Using the input and output defined by `ParentComp`

The syntax is pretty straightforward for using “`ParentComp`”:

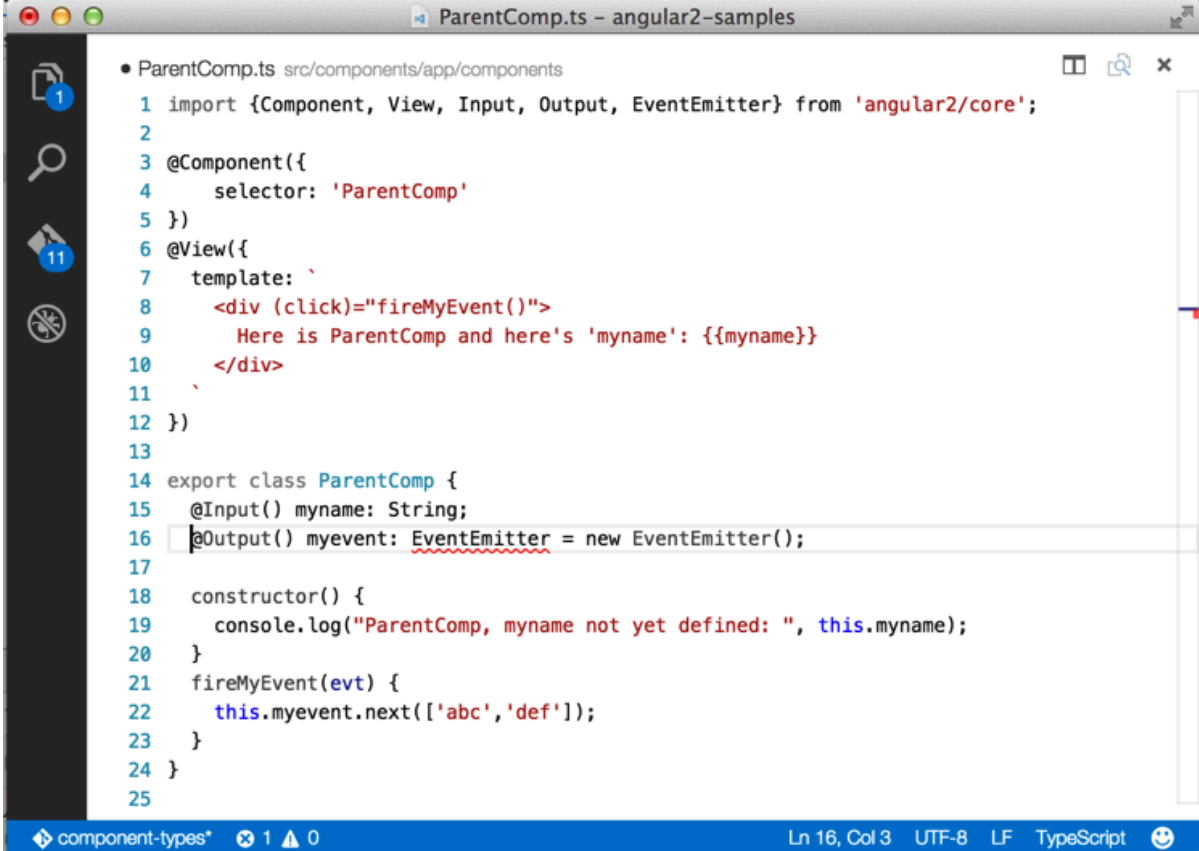
- `[myname] = "myFriend"` : This tells Angular to set the ParentComp input property “myname” to the value of “myFriend” interpolated as a property of CompDemo. Notice we set “myFriend” in the constructor
- `(myevent) = "handleMyEvent($event)"` : This tells Angular to invoke the CompDemo “`handleMyEvent($event)`” method when ParentComp fires “myevent.” The data that we passed to the “`next()`” method in ParentComp is available in CompDemo by passing “`$event`” as an argument to the “`handleMyEvent()`” method.

In both cases, the left side of the attribute refers to something in ParentComp (an input property or an output event) and the right side refers to something that’s interpreted in the context of CompDemo (an instance property or a method).

If you try to set a property on ParentComp without specifying it as an input property, Angular won’t throw an error, but it also won’t set the property. The above pattern — passing data in through an “input” property and sending data out through an “output” event — is the primary way to share data between Angular components. We’ll see in a future article that we can also share data between components by defining services that can be injected into components, effectively giving us a way to share data or functions between components.

@Input() and @Output()

There is an alternative syntax available to define input properties and output events in a component. In the example above we used the “inputs” and “outputs” properties of the object passed to the `@Component` decorator. Angular also lets us use an `@Input` and `@Output` decorator to get the same result:



```

ParentComp.ts src/components/app/components
1 import {Component, View, Input, Output, EventEmitter} from 'angular2/core';
2
3 @Component({
4   selector: 'ParentComp'
5 })
6 @View({
7   template: `
8     <div (click)="fireMyEvent()">
9       Here is ParentComp and here's 'myname': {{myname}}
10    </div>
11  `
12 })
13
14 export class ParentComp {
15   @Input() myname: String;
16   @Output() myevent: EventEmitter = new EventEmitter();
17
18   constructor() {
19     console.log("ParentComp, myname not yet defined: ", this.myname);
20   }
21   fireMyEvent(evt) {
22     this.myevent.next(['abc', 'def']);
23   }
24 }
25
component-types* 1 0 Ln 16, Col 3 UTF-8 LF TypeScript

```

5-5. Angular components: Using the @Input and @Output decorator

In the above version of ParentComp, we dispensed with the “inputs” and “outputs” properties of the `@Component` definition object. Instead, we added “Input” and “Output” to the import command on line 2 and used the `@Input` and `@Output` decorator in the ParentComp class to declare “myname” and “myevent.”

Whether you use inputs/outputs or `@Input` / `@Output`, the result is the same, so choosing which one to use is largely a stylistic decision.

Wrapup

In this article, we’ve looked in more depth at Angular components, how they relate, and how you pass data into them and how to get data back out. We’re still only scratching the surface in terms of components; they’re arguably the major

feature of Angular and are involved in every aspect of designing and building Angular applications. In future articles, we'll continue to explore Angular components by looking in more detail at Angular services as a way to re-use code and encapsulate key functionality.

This article is part of a web development series from Microsoft. Thank you for supporting the partners who make SitePoint possible.

A Practical Guide to Angular Directives

Claudio Ribeiro

Chapter

6

This article focuses on Angular directives — what are they, how to use them, and to build our own.

Directives are perhaps the most important bit of an Angular application, and if we think about it, the most-used Angular unit, the component, is actually a directive.

An Angular component isn't more than a directive with a template. When we say that components are the building blocks of Angular applications, we're actually saying that directives are the building blocks of Angular applications.

Basic overview

At the core, a directive is a function that executes whenever the Angular compiler finds it in the DOM. Angular directives are used to extend the power of the HTML by giving it new syntax. Each directive has a name — either one from the Angular predefined like `ng-repeat`, or a custom one which can be called anything. And each directive determines where it can be used: in an `eLement`, `attribute`, `class` or `comment`.

By default, from Angular versions 2 and onward, Angular directives are separated into three different types:

Components

As we saw earlier, components are just directives with templates. Under the hood, they use the directive API and give us a cleaner way to define them.

The other two directive types don't have templates. Instead, they're specifically tailored to DOM manipulation.

Attribute directives

Attribute directives manipulate the DOM by changing its behavior and

appearance.

We use attribute directives to apply conditional style to elements, show or hide elements or dynamically change the behavior of a component according to a changing property.

Structural directives

These are specifically tailored to create and destroy DOM elements.

Some attribute directives — like `hidden`, which shows or hides an element — basically maintain the DOM as it is. But the structural Angular directives are much less DOM friendly, as they add or completely remove elements from the DOM. So, when using these, we have to be extra careful, since we're actually changing the HTML structure.

Using the Existing Angular Directives

Using the existing directives in Angular is fairly easy, and if you've written an Angular application in the past, I'm pretty sure you've used them. The `ngClass` directive is a good example of an existing Angular attribute directive:

```
<p [ngClass]="{'blue'=true, 'yellow'=false}">
  Angular Directives Are Cool!
</p>

<style>
  .blue{color: blue}
  .yellow{color: yellow}
</style>
```

So, by using the `ngClass` directive on the example below, we're actually adding the `blue` class to our paragraph, and explicitly not adding the `yellow` one. Since we're changing the appearance of a class, and not changing the actual HTML structure, this is clearly an attribute directive. But Angular also offers out-of-the-

box structural directives, like the `ngIf`:

```
@Component({
  selector: 'ng-if-simple',
  template: `
    <button (click)="show = !show">{{show ? 'hide' : 'show'}}</button>
    show = {{show}}
    <br>
    <div *ngIf="show">Text to show</div>
  `
})

class NgIfSimple {
  show: boolean = true;
}
```

In this example, we use the `ngIf` directive to add or remove the text using a button. In this case, the HTML structure itself is affected, so it's clearly a structural directive.

For a complete list of available Angular directives, we can check the [official documentation](#).

As we saw, using Angular directives is quite simple. The real power of Angular directives comes with the ability to create our own. Angular provides a clean and simple API for creating custom directives, and that's what we'll be looking at in the following sections.

Creating an attribute directive

Creating a directive is similar to creating a component. But in this case, we use the `@Directive` decorator. For our example, we'll be creating a directive called "my-error-directive", which will highlight in red the background of an element to indicate an error.

For our example, we'll be using the [Angular quickstart package](#). We just have to

clone the repository, then run `npm install` and `npm start`. It will provide us a boilerplate app that we can use to experiment. We'll build our examples on top of that boilerplate.

Let's start by creating a file called `app.myerrordirective.ts` on the `src/app` folder and adding the following code to it:

```
import {Directive, ElementRef} from '@angular/core';

@Directive({
  selector: '[my-error]'
})

export class MyErrorDirective{
  constructor(elr:ElementRef){
    elr.nativeElement.style.background='red';
  }
}
```

After importing the `Directive` from `@angular/core` we can then use it. First, we need a selector, which gives a name to the directive. In this case, we call it `my-error`.

Best practice dictates that we always use a prefix when naming our Angular directives. This way, we're sure to avoid conflicts with any standard HTML attributes. We also shouldn't use the `ng` prefix. That one's used by Angular, and we don't want to confuse our custom created Angular directives with Angular predefined ones. In this example, our prefix is `my-`.

We then created a class, `MyErrorDirective`. To access any element of our DOM, we need to use `ElementRef`. Since it also belongs to the `@angular/core` package, it's a simple matter of importing it together with the `Directive` and using it.

We then added the code to actually highlight the constructor of our class.

To be able to use this newly created directive, we need to add it to the declarations on the `app.module.ts` file:

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { MyErrorDirective } from './app.myerrordirective';

import { AppComponent }  from './app.component';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent, MyErrorDirective ],
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```

Finally, we want to make use of the directive we just created. To do that, let's navigate to the `app.component.ts` file and add the following:

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `<h1 my-error>Hello {{name}}</h1>`,
})
export class AppComponent { name = 'Angular'; }
```

The final result looks similar to this:



6-1. Angular directives: attribute directive example

Creating a structural directive

In the previous section, we saw how to create an attribute directive using Angular. The approach for creating a structural behavior is exactly the same. We create a new file with the code for our directive, then we add it to the declarations, and finally, we use it in our component.

For our structural directive, we'll implement a copy of the `ngIf` directive. This way, we'll not only be implementing a directive, but also taking a look at how Angular directives handle things behind the scenes.

Let's start with our `app.mycustomifdirective.ts` file:

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({
  selector: '[myCustomIf]'
})

export class MyCustomIfDirective {

  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef) { }

  @Input() set myCustomIf(condition: boolean) {
    if (condition) {
      this.viewContainer.createEmbeddedView(this.templateRef);
    } else {
      this.viewContainer.clear();
    }
  }
}
```

As we can see, we're using a couple of different imports for this one, mainly: `Input`, `TemplateRef` and `ViewContainerRef`. The `Input` decorator is used to pass data to the component. The `TemplateRef` one is used to instantiate

embedded views. An embedded view represents a part of a layout to be rendered, and it's linked to a template. Finally, the `ViewContainerRef` is a container where one or more Views can be attached. Together, these components work as follows:

Directives get access to the view container by injecting a `ViewContainerRef`. Embedded views are created and attached to a view container by calling the `ViewContainerRef`'s `createEmbeddedView` method and passing in the template. We want to use the template our directive is attached to so we pass in the injected `TemplateRef`. — from Rangle.io's [Angular Training](#)

Next, we add it to our declarators:

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { MyErrorDirective } from './app.myerrordirective';
import { MyCustomIfDirective } from './app.mycustomifdirective';

import { AppComponent }  from './app.component';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent, MyErrorDirective, MyCustomIfDirective ],
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```

And we use it in our component:

```
import { Component } from '@angular/core';
```

```

@Component({
  selector: 'my-app',
  template: `<h1 my-error>Hello {{name}}</h1>
    <h2 *myCustomIf="condition">Hello {{name}}</h2>
    <button (click)="condition = !condition">Click</button>`,
})

export class AppComponent {
  name = 'Angular';
  condition = false;
}

```

The kind of approach provided by structural directives can be very useful, such as when we have to show different information for different users based on their permissions. For example, a site administrator should be able to see and edit everything, while a regular user shouldn't. If we loaded private information into the DOM using an attribute directive, the regular user and all users for that matter would have access to it.

Angular Directives: Attribute vs Structural

We've looked at attribute and structural directives. But when should we use one or the other?

The answer might be confusing and we can end up using the wrong one just because it solves our problems. But there's a simple rule that can help us choose the right one. Basically, if the element that has the directive will still be useful in the DOM when the DOM is not visible, then we should definitely keep it. In this case, we use an attribute directive like `hidden`. But if the element has no use, then we should remove it. However, we have to be careful to avoid some common pitfalls. We have to avoid the pitfall of always hiding elements just because it's easier. This will make the DOM much more complex and probably have an impact on overall performance. The pitfall of always removing and recreating elements should also be avoided. It's definitely cleaner, but comes at the expense of performance.

All in all, each case should be carefully analyzed, because the ideal solution is always the one that has the least overall impact on your application structure, behavior and performance. That solution might be either attribute directives, structural directives or, in the most common scenario, a compromise between both of them.

Conclusion

In this article, we took a look at Angular directives, the core of Angular applications. We looked at the different types of directives and saw how to create custom ones that suit our needs.

Angular Components and Providers: Classes, Factories & Values

David Aden

Chapter

7

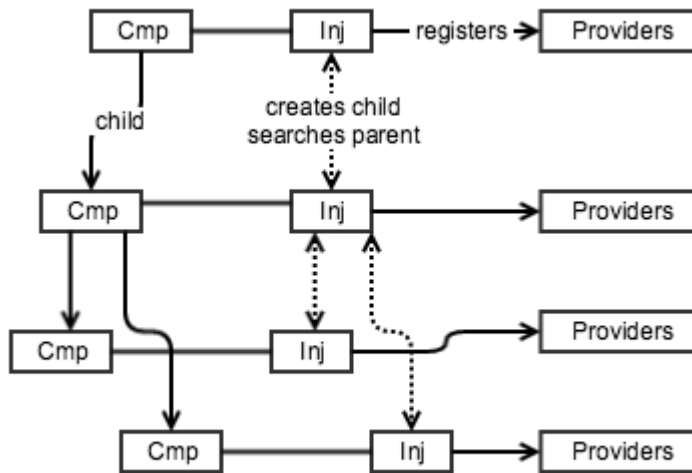
In the Chapter 5, we looked at how to get data into and out of components using the `@Input` and `@Output` annotations. In this article, we'll look at another fundamental aspect of Angular components — their ability to use *providers*.

You may have seen “providers” in a list of properties you can use to configure components, and you might have realized that they allow you to define a set of injectable objects that will be available to the component. That’s nice, but it of course begs the question, “what is a provider?”

Answering that question gets us into an involved discussion of Angular’s Dependency Injection (DI) system. We may specifically cover DI in a future blog post, but it’s well covered in a series of articles by Pascal Precht, beginning with: <http://blog.thoughttram.io/angular/2015/05/18/dependency-injection-in-angular-2.html>. We’ll assume you are familiar with DI and Angular’s DI system in general, as covered in Pascal’s article, but in brief the DI system is responsible for:

- Registering a class, function or value. These items, in the context of dependency injection, are called “providers” because they result in something. For example, a class is used to provide or result in an instance. (See below for more details on provider types.)
- Resolving dependencies between providers — for example, if one provider requires another provider.
- Making the provider’s result available in code when we ask for it. This process of making the provider result available to a block of code is called “injecting it.” The code that injects the provider results is, logically enough, called an “injector.”
- Maintaining a hierarchy of injectors so that if a component asks for a provider result from a provider not available in its injector, DI searches up the hierarchy of injectors.

In the previous article, we included a diagram showing that components form a hierarchy beginning with a root component. Let’s add to that diagram to include the injectors and the resources (providers) they register:



7-1. Component hierarchy with injectors and resources

We can see from the above that while components form a downwards directed graph, their associated injectors have a two-way relationship: parent injectors create children (downwards) and when a provider is requested, Angular searches the parent injector (upwards) if it can't find the requested provider in the component's own injector. This means that a provider with the same identifier at a lower level will shadow (hide) the same-named provider at a higher level.

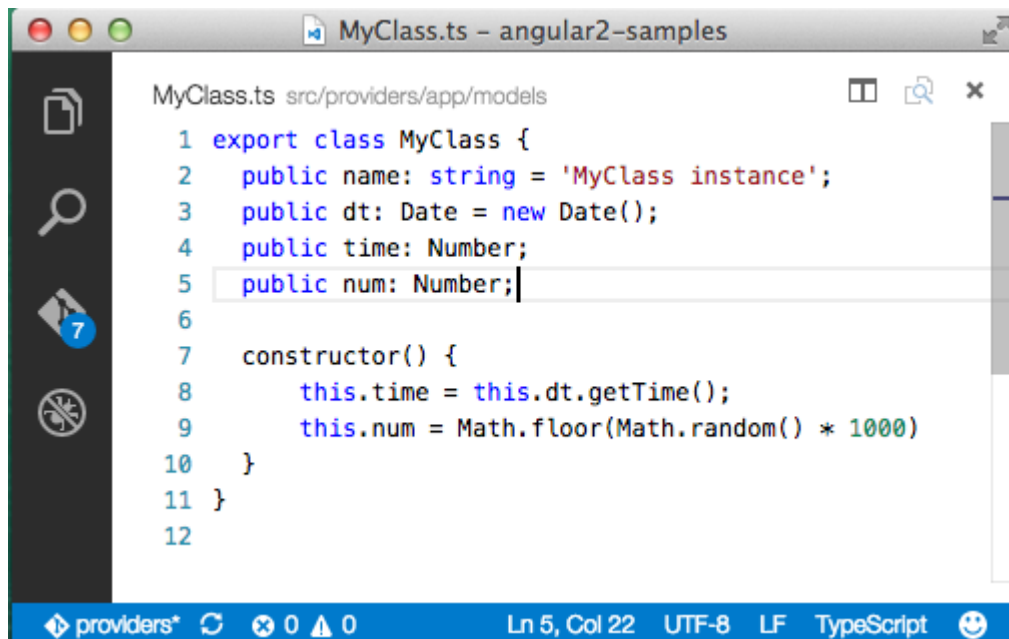
What are Providers?

So, what are these “providers” that the injectors are registering at each level? Actually, it's simple: a provider is a resource or JavaScript “thing” that Angular uses to provide (result in, generate) something we want to use:

- A class provider generates/provides an instance of the class.
- A factory provider generates/provides whatever returns when you run a specified function.
- A value provider doesn't need to take an action to provide the result like the previous two, it just returns its value.

Unfortunately, the term “provider” is sometimes used to mean both the class, function or value and the thing that results from the provider — a class instance, the function’s return value or the returned value.

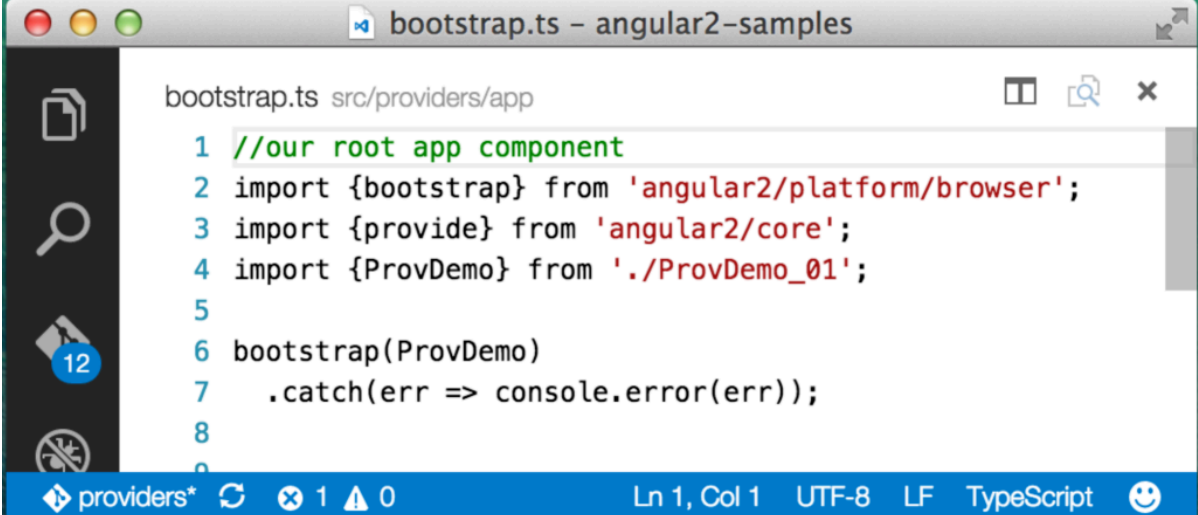
Let’s see how we can add a provider to a component by creating a class provider using `MyClass`, a simple class that will generate the instance we want to use in our application.



```
MyClass.ts src/providers/app/models
1 export class MyClass {
2   public name: string = 'MyClass instance';
3   public dt: Date = new Date();
4   public time: Number;
5   public num: Number;
6
7   constructor() {
8     this.time = this.dt.getTime();
9     this.num = Math.floor(Math.random() * 1000)
10  }
11 }
12
```

7-2. A simple class with four properties

Okay, that’s the class. Now let’s instruct Angular to use it to register a class provider so we can ask the dependency injection system to give us an instance to use in our code. We’ll create a component, `ProvDemo_01.ts`, that will serve as the root component for our application. We load this component and kick-off our application in the `bootstrap.ts`:



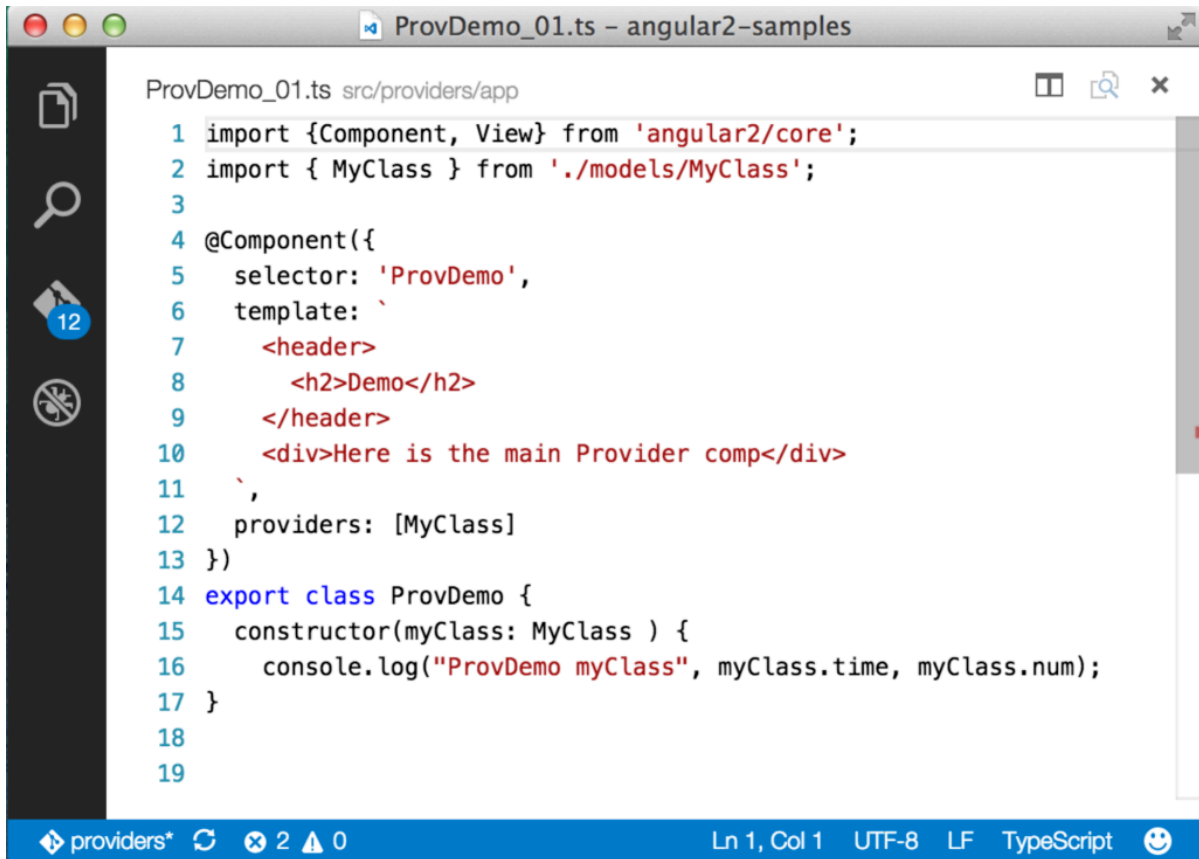
```
bootstrap.ts  src/providers/app

1 //our root app component
2 import {bootstrap} from 'angular2/platform/browser';
3 import {provide} from 'angular2/core';
4 import {ProvDemo} from './ProvDemo_01';
5
6 bootstrap(ProvDemo)
7   .catch(err => console.error(err));
8
9
```

providers* 1 0 Ln 1, Col 1 UTF-8 LF TypeScript

7-3. Our application's bootstrap.ts file that instantiates the root component

If the above doesn't make sense, then take a look at our earlier post that walks through building a simple Angular application. Our root component is called `ProvDemo`, and the repository contains several numbers versions of it. You can change the version that's displayed by updating the line that imports `ProvDemo` above. Our first version of the root component looks like this:



```

ProvDemo_01.ts src/providers/app
1 import {Component, View} from 'angular2/core';
2 import { MyClass } from './models/MyClass';
3
4 @Component({
5   selector: 'ProvDemo',
6   template: `
7     <header>
8       <h2>Demo</h2>
9     </header>
10    <div>Here is the main Provider comp</div>
11  `,
12   providers: [MyClass]
13 })
14 export class ProvDemo {
15   constructor(myClass: MyClass ) {
16     console.log("ProvDemo MyClass", myClass.time, myClass.num);
17   }
18
19

```

7-4. CompDemo with MyClass imported

Adding the `MyClass` provider to this component is straightforward:

- Import `MyClass`
- Add it to the `@Component` providers property
- Add an argument of type “`MyClass`” to the constructor.

Under the covers, when Angular instantiates the component, the DI system creates an injector for the component which registers the `MyClass` provider. Angular then sees the `MyClass` type specified in the constructor’s argument list and looks up the newly registered `MyClass` provider and uses it to generate an instance which it assigns to “myClass” (initial small “m”).

The process of looking up the `MyClass` provider and generating an instance to

assign to “myClass” is all Angular. It takes advantage of the TypeScript syntax to know what type to search for but Angular’s injector does the work of looking up and returning the `MyClass` instance.

Given the above, you might conclude that Angular takes the list of classes in the “providers” array and creates a simple registry used to retrieve the class. But there’s a slight twist to make things more flexible. A key reason why a “twist” is needed is to help us write unit tests for our components that have providers we don’t want to use in the testing environment. In the case of `MyClass`, there isn’t much reason not to use the real thing, but if `MyClass` made a call to a server to retrieve data, we might not want to or be able to do that in the test environment. To get around this, we need to be able to substitute within `ProvDemo` a mock `MyClass` that doesn’t make the server call.

How do we make the substitution? Do we go through all our code and change every `MyClass` reference to `MyClassMock`? That’s not efficient and is a poor pattern for writing tests.

We need to swap out the provider implementation without changing our `ProvDemo` component code. To make this possible, when Angular registers a provider it sets up a map to associate a key (called a “token”) with the actual provider. In our example above, the token and the provider are the same thing: `MyClass`. Adding `MyClass` to the providers property in the `@Component` decorator is shorthand for:

```
providers: [ provide(MyClass, { useClass: MyClass } ) ]
```

This says “register a provider using ‘MyClass’ as the token (key) to find the provider and set the provider to `MyClass` so when we request the provider, the dependency injection system returns a `MyClass` instance.” Most of us are used to thinking of keys as being either numbers or strings. But in this case the token (key) is the class itself. We could have also registered the provider using a string for the token as follows:

```
providers: [ provide("aStringNameForMyClass", { useClass: MyClass } )
```

So, how does this help us with testing? It means in the test environment we can override the provider registration, effectively doing:

```
provide(MyClass, { useClass: MyClassMock })
```

This associates the token (key) `MyClass` with the class provider `MyClassMock`. When our code asked the DI system to inject `MyClass` in testing, we get an instance of `MyClassMock` which can fake the data call. The net effect is that all our code remains the same and we don't have to worry about whether the unit test will make a call to a server that might not exist in the test environment.

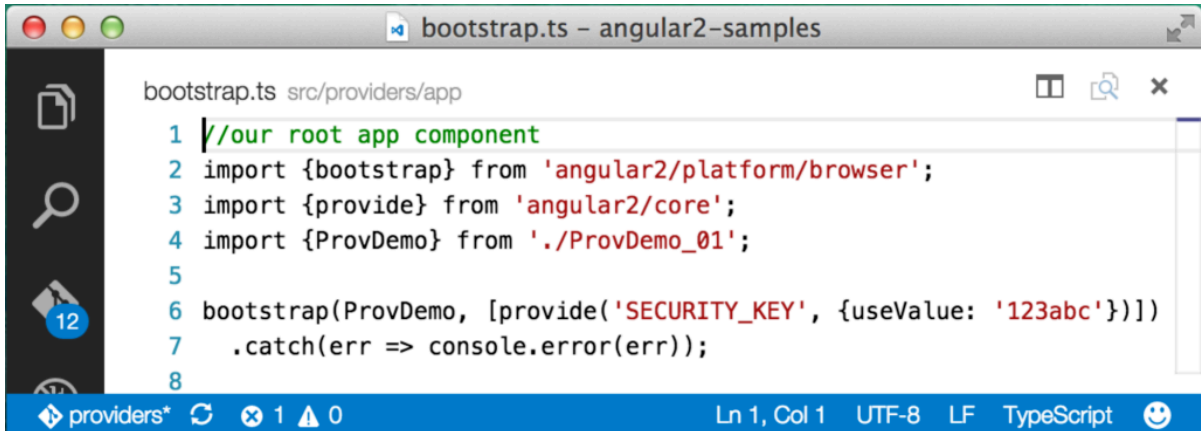
Injecting Non-Class Providers

In the above, we injected our class provider instance into the constructor by writing:

```
constructor( myClass: MyClass ) {...}
```

TypeScript lets us specify that the `myClass` argument needs to be of type `MyClass` and the DI system does the work to give us the `MyClass` instance.

But how do we tell Angular to inject our provider result if we use a string token instead of a class? Let's edit our `bootstrap.ts` file to add a new value provider and register it using a string token. Remember value providers are a type of provider that returns the value associated with the token. In the example above we told Angular to register a provider by adding to the `@Component` providers property but we can also register providers by passing them into the bootstrap function as follows (the same thing could be added to the providers property):



```
bootstrap.ts  src/providers/app
1  //our root app component
2  import {bootstrap} from 'angular2/platform/browser';
3  import {provide} from 'angular2/core';
4  import {ProvDemo} from './ProvDemo_01';
5
6  bootstrap(ProvDemo, [provide('SECURITY_KEY', {useValue: '123abc'})])
7    .catch(err => console.error(err));
8
```

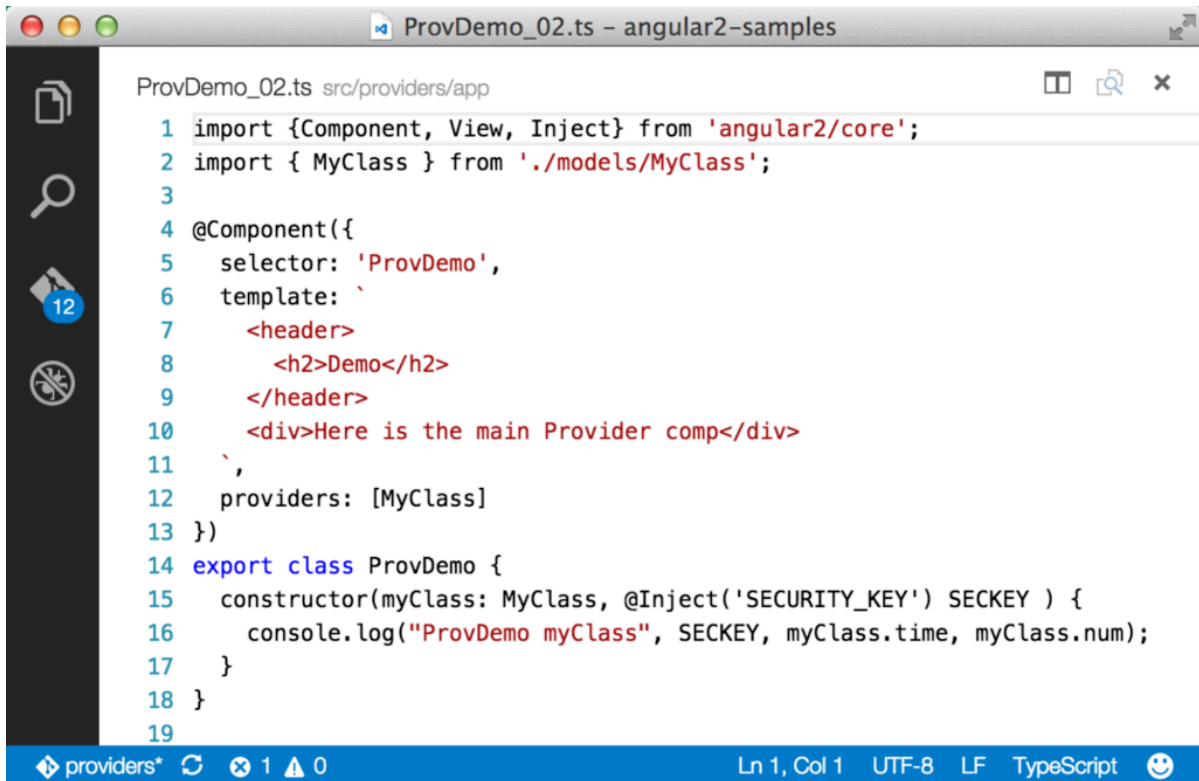
providers* 1 0 Ln 1, Col 1 UTF-8 LF TypeScript

7-5. bootstrap.ts with a value provider added

Here we've added a provider by invoking the `provide` function and passed in a string token ("SECURITY_KEY") and an object which specifies we want to create a value provider and the provider itself — in this case a simple value. Now, we'd like to inject the value generated by the value provider into our constructor, but this isn't going to work ...

```
constructor( SECKEY: "SECURITY_KEY" ) { ... }
```

This is because "SECURITY_KEY" is not a type. To make it possible to inject providers with non-class tokens, Angular gives us the `@Inject` parameter decorator. As with all other decorators, we need to import it and then we use it to tell Angular to inject a provider associated with our string token. To do this we adjust create `ProvDemo_02.ts` :



```

ProvDemo_02.ts src/providers/app
1 import {Component, View, Inject} from 'angular2/core';
2 import { MyClass } from './models/MyClass';
3
4 @Component({
5   selector: 'ProvDemo',
6   template: `
7     <header>
8       <h2>Demo</h2>
9     </header>
10    <div>Here is the main Provider comp</div>
11  `,
12   providers: [MyClass]
13 })
14 export class ProvDemo {
15   constructor(myClass: MyClass, @Inject('SECURITY_KEY') SECKEY ) {
16     console.log("ProvDemo myClass", SECKEY, myClass.time, myClass.num);
17   }
18 }
19

```

7-6. Importing the “Inject” decorator and using it to inject a value provider

We could use the same syntax to inject the `MyClass` provider:


```
constructor( @Inject(MyClass) myClass, @Inject('SECURITY_KEY') SECKEY ) {...}
```

Okay, we’ve seen how to register and use providers, but let’s learn a little bit more about what providers return.

Providers and Singletons

As we saw above, providers are responsible for generating the thing that gets injected. A class provider generates an instance and the instance gets injected. But it’s important to understand that you don’t get a new instance each time the class provider result is injected. Instead, the DI system generates the instance once, caches it and each subsequent injection receives the same instance as long as you use the same provider.

The last is important because each component gets its own injector with its own registered providers. `MyClass` has a `time` property set to the current time in milliseconds and a random number to help us see if we're getting the same instance each time. We're going to add a `ChildComp` component to our application.



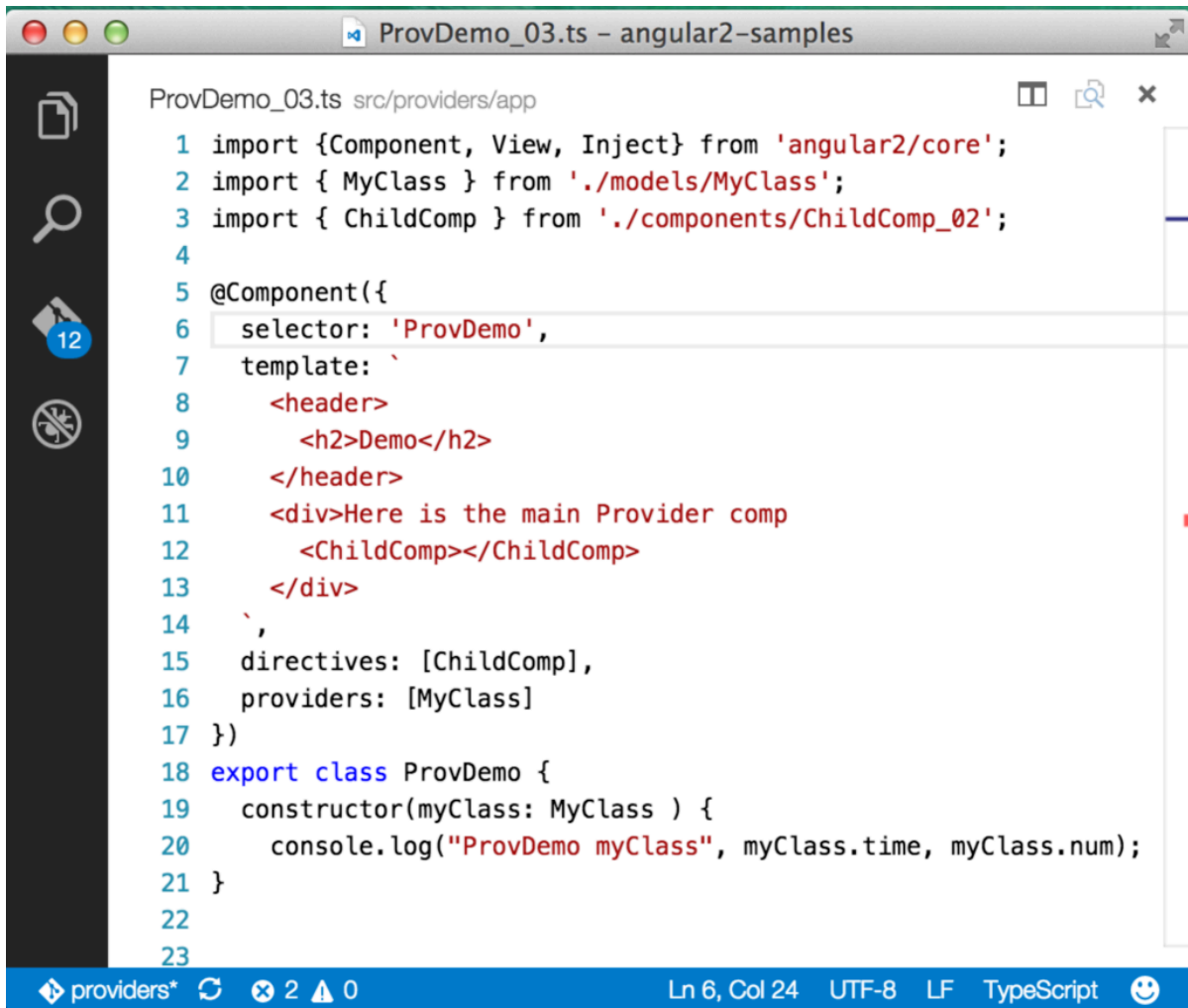
```

ChildComp_01.ts src/providers/app/components
1 import {Component, View, Input} from 'angular2/core';
2 import { MyClass } from '../models/MyClass';
3
4 @Component({
5   selector: 'ChildComp',
6   template: `
7     <div>
8       This is ChildComp.
9     </div>
10 `
11 })
12
13 export class ChildComp {
14   constructor(myClass: MyClass) {
15     console.log("ChildComp myClass", myClass.time, myClass.num);
16   }
17 }
18

```

7-7. ChildComp with MyClass injected into the constructor

Notice we import `MyClass` and use it to set the type in the constructor's argument list. Important: *The only purpose the imported `MyClass` serves in `ChildComp` is as a token the DI system uses, to look for a registered provider.* Because `ChildComp` does not have its own provider registered using that token, Angular looks up the injector hierarchy to find one. To make this work, we need to add `ChildComp` to the `ProvDemo` component:



```

ProvDemo_03.ts src/providers/app

1 import {Component, View, Inject} from 'angular2/core';
2 import { MyClass } from './models/MyClass';
3 import { ChildComp } from './components/ChildComp_02';
4
5 @Component({
6   selector: 'ProvDemo',
7   template: `
8     <header>
9       <h2>Demo</h2>
10    </header>
11    <div>Here is the main Provider comp
12      <ChildComp></ChildComp>
13    </div>
14  `,
15   directives: [ChildComp],
16   providers: [MyClass]
17 })
18 export class ProvDemo {
19   constructor(myClass: MyClass ) {
20     console.log("ProvDemo myClass", myClass.time, myClass.num);
21   }
22 }
23

```

7-8. ProvDemo with ChildComp added to the template

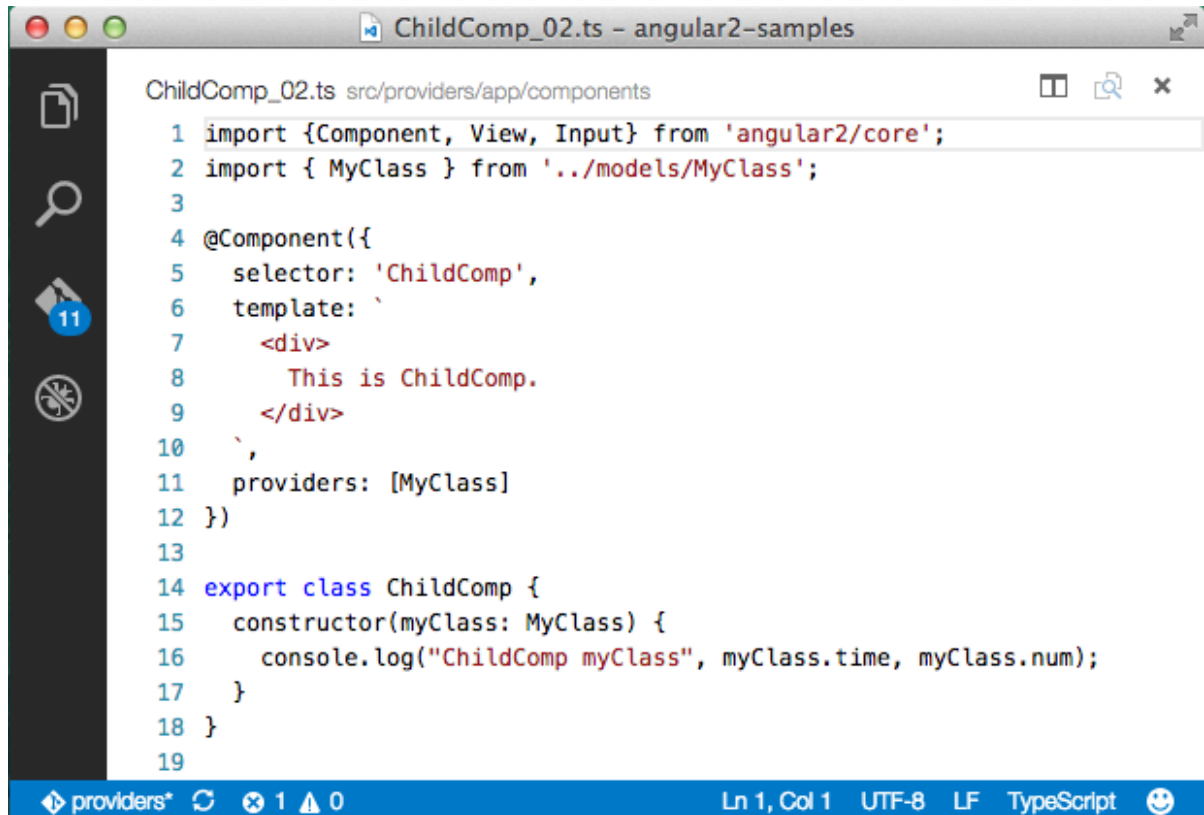
We import `ChildComp`, add a directives property to `@Component` to tell `ProvDemo` we're going to use the `ChildComp` component and add the `ChildComp` element to the template. When the application runs, the console output shows that both `ProvDemo` and `ChildComp` receive the same instance of `MyClass`:

```

ProvDemomyClass 1453033148406 390
ChildCompmyClass 1453033148406 390

```

Now let's change `ChildComp` to add a `MyClass` provider to its injector:



```

ChildComp_02.ts src/providers/app/components
1 import {Component, View, Input} from 'angular2/core';
2 import { MyClass } from '../models/MyClass';
3
4 @Component({
5   selector: 'ChildComp',
6   template: `
7     <div>
8       This is ChildComp.
9     </div>
10  `,
11   providers: [MyClass]
12 })
13
14 export class ChildComp {
15   constructor(myClass: MyClass) {
16     console.log("ChildComp myClass", myClass.time, myClass.num);
17   }
18 }
19

```

7-9. ParentComp with its own MyClass provider defined

All we've changed is to add the providers property to the @Component annotation. And, sure enough, we can see that two different `MyClass` instances are created:

```

ProvDemomyClass 1453033681877 263
ChildCompmyClass 1453033681881 761

```

This feature of Angular gives a lot of flexibility over the results generated by any one provider and whether we are going to work with a single instance or multiple instances. For example, you might put a component inside a repeater so the component is generated multiple times. If this repeated component registers its own provider, each one gets unique providers. But, if you only register provider in the parent component, each repeated instance shares the parent's provider.

Wrapup

In this article, we defined what a provider is and covered the three different types of providers. We then looked at how you can register a provider for a component and inject the result generated by the provider into the component. We also took a look at how the hierarchy of injectors is used by Angular to find a requested provider. Angular gives you additional control over how the dependency injection system works and where it looks for providers but the above should get you started creating and working with providers in your Angular applications.

Quickly Create Simple Yet Powerful Angular Forms

Kaloyan Kolev

Chapter

8

Forms are an essential part of many web applications, being the most common way to enter and edit text-based data. Front-end JavaScript frameworks such as Angular, often have their own idiomatic ways of creating and validating forms that you need to get to grips with to be productive.

Angular allows you to streamline this common task by providing two types of forms that you can create:

- **Template-driven forms** - simple forms that can be made rather quickly.
- **Reactive forms** - more complex forms that give you greater control over the elements in the form.

In this article, we'll make a simple example form with each method to see how it's done.

Prerequisites

You do not need to know all the details of how to create an Angular application to understand the framework's usefulness when it comes to forms. However, if you want to get a better grasp of Angular, you can take a look at this SitePoint article series on [building a CRUD app with Angular](#).

Requirements

We will use [Bootstrap](#) in this tutorial. It is not an integral part of an Angular application, but it will help us streamline our efforts even further by providing ready-made styles.

This is how you can add it to your application:

- 1 Open the command prompt and navigate to the folder of your project
- 2 Type `npm install bootstrap@next`. This will add the latest version of bootstrap to the project

- 3 Edit the `.angular-cli.json` file and add a link to the Bootstrap CSS file

```
"apps": [
  "styles": [
    "../node_modules/bootstrap/dist/css/bootstrap.css"
  ]
]
```

We will not use the Bootstrap JavaScript file in this application.

- 4 Both template-driven Forms and Reactive Forms require the `FormsModule`. It should be added to the application in `app.module`:

```
import { FormsModule } from '@angular/forms';
@NgModule({
  imports: [
    BrowserModule,
    FormsModule
  ]
})
```

With that out of the way, we can proceed with the forms themselves.

Template-Driven Forms

Let us assume you want to create a simple form as quickly as possible. For example, you need a company registration form. How can you create the form?

The first step is to create the `<form>` tag in your view.

```
<form #companyForm="ngForm">
```

We need to modify this tag in two ways in order to submit the form and use the

information from the input fields in our component:

- We will declare a template variable using the `ngForm` directive.
- We will bind the `ngSubmit` event to a method we will create in our component

```
<form #companyForm="ngForm" (ngSubmit)="submitCompany(companyForm.form);">
```

We will create the `submitCompany` method in the component a bit later. It will be called when the form is submitted and we will pass it the data from the form via `companyForm.form`.

We also need a submit button, regardless of the content of the form. We will use a few Bootstrap classes to style the button. It is good practice to disable the button before all the data validation requirements are met. We can use the template variable we created for the form to achieve this. We will bind the disabled property to the valid property of the `companyForm` object. This way the button will be disabled if the form is not valid.

```
<button class="btn btn-primary" [disabled]="!companyForm.valid">Submit</button>
```

Let us assume our simple form will have two fields - an input field for the name of the company and a drop-down field for the company's industry.

Creating form inputs

First, we create an input field for the name:

```
<input type="text"  
  class="form-control"  
  name="company-name">
```

Right now we have a standard input with the type, name and class attributes. What do we need to do to use the Angular approach on our input?

We need to apply the `ngModel` directive to it. Angular will create a control object and associate it with the field. Essentially, Angular does some of the work for you behind the scenes.

This is a good time to mention that `ngModel` requires the input field to have a name or the form control must be defined as standalone in `ngModelOptions`. This is not a problem because our form already has a name. Angular will use the name attribute to distinguish between the control objects.

In addition, we should specify a template variable for the input: `#nameField` in this case. Angular will set `nameField` to the `ngModel` directive that is applied to the input field. We will use this later for the input field's validation. This variable will also allow us to perform an action based on the value of the field while we are typing in it.

Now our input looks like this:

```
<input type="text"
      class="form-control"
      name="company-name"
      ngModel
      #nameField="ngModel">
```

It is almost the same, but with a few key changes.

Validation

Let us assume we want the company name field to be required and to have a minimum length of 3 characters. This means we have to add the `required` and `minLength` attributes to our input:

```
<input type="text"
      class="form-control"
      name="company-name"
      required
      minLength="3">
```

```
ngModel
#nameField="ngModel"
required
minlength="3">
```

Sounds simple enough, right? We will also need to display an error message if any of these two requirements are not met. Angular allows us to check the input's value and display the appropriate error message before the form is submitted.

We can perform such a check while the user is typing in the form. First of all, it is a good idea to display an error only after the user has started to interact with the form. There is no use in displaying an error message right after we load the page. This is why we will insert all the error messages for this input inside the following div:

```
<div *ngIf="nameField.touched && nameField.errors"></div>
```

The `ngIf` directive allows us to show the div only when a specific condition is true. We will use the `nameField` template variable again here because it is associated with the input. In our case, the div will be visible only, if the input has been touched and there is a problem with it. Alright, what about the error messages themselves?

We will place another div inside the aforementioned one for each error message we want. We will create a new div for the error message and use the `nameField` template variable again:

```
<div class="alert alert-danger"
  *ngIf="nameField.errors.required">
  The company name is required
</div>
```

We are using the "alert alert-danger" bootstrap classes to style the text field.

The `nameField` variable has the property `errors`, which contains an object with key-value pairs for all the current errors. The `ngIf` directive allows us to show this error message only when the 'required' condition is not met. We will use the same approach for the error message about the minimum length.

```
<div class="alert alert-danger"
  *ngIf="nameField.errors.minLength">
  The company name should be at least 3 characters long
</div>
```

This div will be visible only when the `minLength` requirements are not met. here we can make the error message a little bit more dynamic.

Currently, we have specified the minimum length in two locations - in the input's attribute and the text field. We can improve this by replacing the hardcoded "3" with the `requiredLength` property of the `minLength` object like so:

```
<div class="alert alert-danger"
  *ngIf="nameField.errors.minLength">
  The company name should be at least {{ nameField.errors.minLength.
    ↳requiredLength }} characters long
</div>
```

This way the number of the minimum length in the error message will depend on the input's `minLength` attribute.

Now, we will do the same thing with the dropdown field for the company's industry:

```
<select class="form-control"
  name="company-industry"
  ngModel
  #industryField="ngModel"
  required>
```

We will create a list of the options for the dropdown in the component associated with this view in order to avoid hard-coding values in the HTML.

```
export class ContactFormComponent implements OnInit {
  industries = [
    {id: 1, name: "Agriculture"},
    {id: 2, name: "Manufacturing"},
    {id: 3, name: "Energy"},
    {id: 4, name: "Transportation"},
    {id: 5, name: "Finance"}
  ];
}
```

Now we can list all the options in the view via the `ngFor` directive. It will create an option tag for every element in the `industries` array from the component.

```
<option *ngFor="let industry of industries"
  [value]="industry.id">
  {{ industry.name }}
</option>
```

The validation for this field is quite easy and similar to that for the company name field:

```
<div class="alert alert-danger"
  *ngIf="industryField.touched && !industryField.valid">
  The industry is required
</div>
```

Now our form is ready for submission. Earlier we bound the `ngSubmit` event to a method called `submitCompany`; let's go to the component and add that now:

```
export class ContactFormComponent implements OnInit {
  submitCompany(form){
    console.log(form.value);
    alert("The form was submitted");
  }
}
```

```

    form.reset();
  }
}

```

The `form` parameter will contain all the data from the form. On the other hand, `form.value` will contain just an object with the values of the fields in the form.

Here I will just log the result in the console, but you can do whatever you want with it. I have added an alert with a message to inform the user the form was submitted. This is not required, but it is a good practice to show some sort of notification. `form.reset()` will reset the form to its initial state after submission, which means the fields will be emptied.

Alright let us see what our form should look like: <https://sitepoint-editors.github.io/company-registration-form/>

Reactive Forms

The other kind of form you can create is a reactive form, which allows you to explicitly create control objects for the form fields yourself. This approach is a good choice when you are building a more complex form and you want to have more control over its behavior.

Let us assume that we need to create an account registration form, which will have two fields for an email and a password. We will use Bootstrap to style this form as well.

The first step is to import the `ReactiveFormsModule` class in `app.module` because it is necessary for all reactive forms:

```

import { ReactiveFormsModule } from "@angular/forms";

@NgModule({

```

```
imports: [
  ReactiveFormsModule
]
```

Then, we need to import the `FormGroup` and `FormControl` classes in the component for our page in order to explicitly define our control objects:

```
import { FormGroup, FormControl } from "@angular/forms";
```

Now we should create an instance of the `FormGroup` class and specify all the fields in our form. To put it simply, we will list key-value pairs. The keys will be the names of the fields and the values will be the form objects.

```
accountForm = new FormGroup({
  email: new FormControl(),
  password: new FormControl();
```

Next, we should create the form. We will once again need the `<form>` tag. We will add the `FormGroup` directive to it and associate the HTML form with the `accountForm` form group object we created in the component:

```
<form [formGroup]="accountForm"></form>
```

Next, we will create the email input field. We will apply the `formControlName` directive to it and set it to the corresponding key in the list of controls we created in the components, `email`.

```
<input type="text"
  class="form-control"
  id="email"
  formControlName="email">
```

We will do the same for the password field:

```
<input type="text"
      id="password"
      class="form-control"
      formControlName="password">
```

Validation

The next step is to add validation to the form. We will not use any HTML attributes like "required" as with the template-driven forms. Instead, we have to assign all the validators when we create the form control objects.

We will go back to the component where we defined our `accountForm`. All the validator methods for reactive forms are defined in the `Validators` class, which we have to import:

```
import { FormGroup, FormControl, Validators } from "@angular/forms";
```

Then we will assign the validators to the controls in our controller. The format is the following :

```
form = new FormGroup({
  fieldname: new FormControl(
    initial value,
    synchronous validators,
    asynchronous validators)
});
```

Let us assume that both the email and the password fields will be required. We should also check if the email is valid. In addition, the password should contain at least one uppercase letter, one lowercase letter, and one number. Thus, we will use the `required` and `pattern` validators from the `Validators` class for both fields. We will leave their initial values as an empty string.

```

form = new FormGroup({
  email: new FormControl("",
    [Validators.required,
     Validators.pattern('[a-zA-z0-9_\.]+@[a-zA-Z]+\.[a-zA-Z]+')]),
  password: new FormControl("",
    [Validators.required,
     Validators.pattern('^(?=.*[0-9])(?=.*[a-z])(?=.*[A-Z]).{8,}$')])
});

```

Now we need to go to the template and add the validation messages. We will do this in the same way we did it with the template-driven forms. However, we will access the control objects in a different way. In our component we can define a property that gives us access to the control in the form like so:

```

get email(){
  return this.accountForm.get("email");
}

```

We can access this property in our template. This means that instead of writing `this.accountForm.get("email")` every time we want to specify a validation message, we can use just `email`.

```

<div *ngIf="email.touched && email.errors">
  <div class="alert alert-danger" *ngIf="email.errors.required">
    The email is required
  </div>
</div>
<div *ngIf="email.errors">
  <div class="alert alert-danger" *ngIf="email.errors.pattern">
    The email is not valid
  </div>
</div>

```

This way, the message "The email is required" will appear after the user touched the form and left it empty, while the message "The email is not valid" will appear as the user is typing. We can display the validation messages for the password field in the same way.

Let us move on to submitting our reactive form. Firstly, we can disable the submit button in a similar way to the one we used with the template-driven form:

```
<button class="btn btn-primary" type="submit"
  [disabled]="!accountForm.valid">Sign up</button>
```

We also need to bind the `ngSubmit` event to a function, which will be called on submit.

```
<form [formGroup]="accountForm" (ngSubmit)="signup()">
```

Then we need to define that function in the controller:

```
signup(){
  console.log(this.accountForm.value);
  alert('The form was submitted');
  this.accountForm.reset();
}
```

For now, we will show the submitted data in the console. We will clear the form fields after we display a confirmation message.

Asynchronous validation

It will be great if we can check if the email the user is trying to submit is already in use. We can perform such a check even as the user is typing in if we use an asynchronous validator.

We will use a fake API for the purposes of this demo - [JSON Placeholder](#). This is a useful tool for testing an application because it provides various kinds of data. For example, it can provide a list of users with emails, which we will pretend is the list of existing users for our demo application. You can send get and post requests to it just as you would with a real API.

We will create a service in our application that connects to this JSON API, and attaches an asynchronous validator to the email field. This way, we will be able to check if the email is already in use.

First, we will create the service. We can do that via the Angular CLI

```
ng g service server.service
```

Then, we have to add the service to `app.module` so that we can use it in the application:

```
import { ServerService } from "./server.service";
@NgModule({
  providers: [
    ServerService
  ],
  bootstrap: [AppComponent]
})
```

In our service, we need to import the `Injectable`, `Http` and `Observable` classes as well as the `map` and `filter` RxJS operators. Then we will specify the URL to our test API. After we get the results we will filter them to see if there is a user with an email that matches the one the user typed, which we will pass to it when we perform the request.

```
@Injectable()
export class ServerService {
  private url = "http://jsonplaceholder.typicode.com/users";

  constructor(private http: Http) { }

  checkUsers(email: string) {
    return this.http
      .get(this.url)
      .map(res => res.json())
      .map(users => users.filter(user => user.email === email))
  }
}
```



```

        .map(users => !users.length);
    }
}

```

Now we have to create the validator, which will use this service to check the email. We will create a new typescript file, **custom.validators.ts**. This will allow us to separate our code in a more effective way and reuse the validator. There we will import the `AbstractControl` and `ValidationErrors` classes as well as the `ServerService`.

```

import { AbstractControl, ValidationErrors } from '@angular/forms';
import { ServerService } from './server.service';

export class Customvalidators{
    static checkDuplicateEmail(serverService: ServerService) {
        return (control: AbstractControl) => {
            return serverService.checkUsers(control.value).map(res => {
                return res ? null : { duplicateEmail: true };
            });
        };
    }
}

```

We create an instance of our `serverService` and call the `checkUsers` method we created in it. Custom validators are supposed to return `null` if everything is OK, or an object with key-value pairs that describe the error otherwise.

Now we will go to our component to apply the asynchronous validator to the email field. We will have to import the `ServerService` into the component as well and create an instance of it in order to perform the request to our test API.

```

import { ServerService } from "../server.service";

constructor(private serverService: ServerService){

}

```

```
accountForm = new FormGroup({  
  email: new FormControl("", synchronous validators,  
    Customvalidators.checkDuplicateEmail(this.serverService))  
});
```

The only thing left to do is add a validation message

```
<div *ngIf="email.errors">  
  <div class="alert alert-danger" *ngIf="email.errors.duplicateEmail">  
    The email is already in use  
  </div>  
</div>
```

Now let's see what our form looks like. <https://sitepoint-editors.github.io/account-registration-form/>

Wrapping Up

As you can see, Angular allows you to do a few neat tricks with forms. Not only can you create simple forms quickly by making them template-driven, but you can also implement complex features in them if you need to.

Using Angular NgModules for Reusable Code and More

Jeremy Wilken

Chapter

9

NgModules are a core concept in Angular that are part of every application and help to wire up some important details for the compiler and application runtime. They're especially useful for organizing code into features, lazy loading routes, and creating reusable libraries.

In this guide, we're going to cover the primary uses of NgModules with some examples to show you how to use them in your Angular projects! This guide assumes you have a working knowledge of Angular.

JavaScript Modules Aren't NgModules

Let's clear the air first about what JavaScript modules are (sometimes called ES6 modules). They're a language construct that makes it easier to organize your code.

At their most basic, JavaScript modules are JavaScript files that contain either the `import` or `export` keywords, and which cause the objects defined inside of that file to be private unless you export it. I encourage you to review the link above for a deeper understanding, but essentially this is a way to organize your code and easily share it, without relying on the dreaded global scope.

When you create an Angular application with TypeScript, any time you use `import` or `export` in your source, it's treated as a JavaScript module. TypeScript is able to handle the module loading for you.



Naming

To help keep things clear in this article, I'll always refer to JavaScript modules and NgModules by their full names.

The Basic NgModule, the AppModule

Let's start by looking at a basic NgModule that exists in every Angular

application, the `AppModule` (which is generated by default in any new Angular application). It looks something like you see here:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Angular uses decorators to define metadata it needs to know about during compile time. To define an NgModule, you simply add the `@NgModule()` decorator above a class. The class may not always be empty, but often it is. However, you'll need to define an object with some properties for the NgModule to do anything.

When the application bootstraps, it needs to be given an NgModule to instantiate. If you look in the main file of your application (also typically called `main.ts`), you'll see `platformBrowserDynamic().bootstrapModule(AppModule)`, which is how the application registers and initiates the `AppModule` (which can be named anything, but is almost always named this).

The Properties of NgModule

The NgModule API documentation page outlines the properties that you can pass when defining an NgModule, but we'll cover them here as well. They're all optional, but you'll need to define values for at least one of them for the

NgModule to do anything.

`providers`

The `providers` is an array that contains the list of any providers (injectable services) that are available for this NgModule. Providers have a scope, and if they're listed in a lazy-loaded NgModule, they're not available outside of that NgModule.

`declarations`

The `declarations` array should contain a list of any directives, components, or pipes that this NgModule defines. This makes it possible for the compiler to find these items and ensure they're bundled properly. If this is the root NgModule, then declarations are available for all NgModules. Otherwise, they're only visible to the same NgModule.

`imports`

If your NgModule depends on any other objects from another NgModule, you'll have to add it to the `imports` array. This ensures that the compiler and dependency injection system know about the imported items.

`exports`

Using the `exports` array, you can define which directives, components, and pipes are available for any NgModule that imports this NgModule. For example, in a UI library you'd export all of the components that compose the library.

`entryComponents`

Any component that needs to be loaded at runtime has to be added to the list of `entryComponents`. Essentially, this will create the component factory and store it

for when it needs to be loaded dynamically. You can learn more about how to [dynamically load components](#) from the documentation.

bootstrap

You can define any number of components to bootstrap when the app is first loaded. Usually you only need to bootstrap the main root component (usually called the `AppComponent`), but if you had more than one root component, each would be declared here. By adding a component to the `bootstrap` array, it's also added to the list of `entryComponents` and precompiled.

schemas

Schemas are a way to define how Angular compiles templates, and if it will throw an error when it finds elements that aren't standard HTML or known components. By default, Angular throws an error when it finds an element in a template that it doesn't know, but you can change this behavior by setting the schema to either `NO_ERRORS_SCHEMA` (to allow all elements and properties) or `CUSTOM_ELEMENTS_SCHEMA` (to allow any elements or properties with a `-` in their name).

id

This property allows you to give an NgModule a unique ID, which you can use to retrieve a module factory reference. This is a rare use case currently.

NgModule Examples

To illustrate the way NgModule is used with Angular, let's look at a set of examples that show you how to handle various use cases easily.

Feature NgModules

The most basic use case for NgModules besides the `AppModule` is for Feature NgModules (usually called feature modules, but trying to keep the terms consistent). They help separate individual parts of your application, and are highly recommended. In most ways, they're the same as the main App NgModule. Let's take a look at a basic Feature NgModule:

```
@NgModule({
  declarations: [
    ForumComponent,
    ForumsComponent,
    ThreadComponent,
    ThreadsComponent
  ],
  imports: [
    CommonModule,
    FormsModule,
  ],
  exports: [
    ForumsComponent
  ],
  providers: [
    ForumsService
  ]
})
export class ForumsModule { }
```

This simple Feature NgModule defines four components, one provider, and imports two modules that are required by the components and service. Together, these comprise the necessary pieces for the forums section of an application.

The items in `providers` are available to any NgModule that imports the `ForumsModule` to be injected, but it's important to understand that each NgModule will get its own instance of that service. This is different from providers listed in the root NgModule, from which you'll always get the same

instance (unless its reprovided). This is where it's important to understand dependency injection, particularly hierarchical dependency injection. It's easy to think you'll get the same instance of a service and change properties on it, but never see the changes elsewhere in the application.

As we learned earlier, the items in `declarations` are not actually available to be used in other NgModules, because they're private to this NgModule. To fix this, you can optionally export those declarations you wish to consume in other NgModules, like in this snippet where it exports just the `ForumsComponent`. Now, in any other Feature NgModules, you could put `<app-forums></app-forums>` (or whatever the selector for the component is) to display the `ForumsComponent` in a template.

Another key difference is that `ForumsModule` imports the `CommonModule` instead of the `BrowserModule`. The `BrowserModule` should only be imported at the root NgModule, but the `CommonModule` contains the core Angular directives and pipes (such as `NgFor` and the `Date` pipe). If your Feature NgModule doesn't use any of those features, it wouldn't actually need the `CommonModule`.

Now, when you want to consume the `ForumsModule` in your project, you need to import it into your `AppModule` like you see here:

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    ForumsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

This NgModule is then imported into the main `AppModule` to load it properly, which includes the items in the `ForumsModule` providers array and any exported items for consumption in your application.

When you use the Angular CLI, you can easily generate Feature NgModules by running the generator for a new NgModule:

```
ng generate module path/to/module/feature
```

You can organize your Feature NgModules any way you see fit, but the general recommendation is to group similar things that are used on the same view. I try to make a small number of Feature NgModules to hold the commonly shared things, and then focus more on NgModules for each major feature of the application.

Lazy Loading NgModules with Routes

Sometimes you want to load code only when the user needs it, and with Angular this is currently possible by using the router and Feature NgModules together. The router has the ability to lazy load NgModules when a user requests a specific route. See this [primer on routing with Angular](#) if you're new to routing.

The best way to start is to create a Feature NgModule for the unique parts of a route. You might even want to group more than one route, if they're almost always used together. For example, if you have a customer account page with several subpages for managing the account details, more than likely you'd declare them as part of the same NgModule.

There's no difference in the way you define the NgModule itself, except you'll need to define some routes with `RouterModule.forChild()`. You should have one route that has an empty path, which will act like the root route for this Feature NgModule, and all other routes hang from it:

```

@NgModule({
  declarations: [
    ForumComponent,
    ForumsComponent,
  ],
  imports: [
    CommonModule,
    FormsModule,
    RouterModule.forChild([
      {path: '', component: ForumsComponent},
      {path: ':forum_id', component: ForumComponent}
    ])
  ],
  providers: [
    ForumsService
  ]
})
export class ForumsModule { }

```

There is an important change in behavior that isn't obvious related to the way the providers are registered with the application. Since this is a lazy loaded NgModule, providers are *not available* to the rest of the application. This is an important distinction, and should be considered when planning your application architecture. Understanding how Angular dependency injection works is very important here.

To load the lazy route, the main `AppModule` defines the path that goes to this Feature NgModule. To do this, you'll have to update your root router config for a new route. This example shows how to define a lazy loaded route, by giving it a `path` and `loadChildren` properties:

```

const routes: Routes = [
  {
    path: 'forums',
    loadChildren: 'app/forums/forums.module#ForumsModule'
  },
  {

```

```
    path: '',  
    component: HomeComponent  
  }  
];
```

The syntax of the `loadChildren` property is a string that has the path to the NgModule file (without the file extension), a `#` symbol, and then the name of the NgModule class: `loadChildren: 'path/to/module#ModuleName'`. Angular uses this to know where to load the file at runtime, and to know the name of NgModule.

The path to the lazy loaded route is defined at the root level of routes, so the lazy loaded NgModule doesn't even know specifically what the path for its route will be. This makes them more reusable, and makes it possible for the application to know when to lazy load that NgModule. Think of the lazy loaded NgModule defining all routes as relative paths, and the full path is provided by combining the root route and lazy loaded routes.

For example, if you visit the `/` route in this application, it will load the `HomeComponent` and the `ForumsModule` will not be loaded. However, once a user clicks a link to view the forums, it will notice that the `/forums` path requires the `ForumsModule` to be loaded, downloads it, and registers the defined routes from it.

Routing NgModules

A common pattern for Angular is to use a separate NgModule to host all of your routes. It's done for separation of concerns, and is entirely optional. The Angular CLI has support for automatically generating a Routing NgModule when you create a new module by passing the `--routing` flag:

```
ng generate module path/to/module/feature --routing
```

What happens is that you create a standalone NgModule that defines your routes, and then your Feature NgModule imports it. Here's what a routing

NgModule could look like:

```
const routes: Routes = [
  { path: '', component: ForumsComponent }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class ForumsRoutingModule { }
```

Then you just import it to your `ForumsModule` like you see here:

```
@NgModule({
  declarations: [
    ForumComponent,
    ForumsComponent,
  ],
  imports: [
    CommonModule,
    FormsModule,
    ForumsRoutingModule,
  ],
  providers: [
    ForumsService
  ]
})
export class ForumsModule { }
```

This is largely preference, but it's a common pattern you should consider. Essentially, it's another way NgModules are used for code separation.

Singleton services

We've seen a couple of concerns about providers where you couldn't be guaranteed that you'd get the same instance of a service across NgModules unless you only provided it in the root NgModule. There's a way to define your

NgModule so that it can declare providers only for the root NgModule, but not redeclare them for all other NgModules.

In fact, the Angular router is a good example of this. When you define a route in your root NgModule, you use `RouterModule.forRoot(routes)`, but inside of Feature NgModules you use `RouterModule.forChild(routes)`. This pattern is common for any reusable library that needs a single instance of a service (singleton). We can do the same with any NgModule by adding two static methods to our NgModule like you see here:

```
@NgModule({
  declarations: [
    ForumComponent,
    ForumsComponent,
    ThreadComponent,
    ThreadsComponent
  ],
  imports: [
    CommonModule,
    FormsModule,
  ],
  exports: [
    ForumsComponent
  ]
})
export class ForumsModule {
  static forRoot(): ModuleWithProviders {
    return {
      ngModule: ForumsModule,
      providers: [ForumsService]
    };
  }

  static forChild(): ModuleWithProviders {
    return {
      ngModule: ForumsModule,
      providers: []
    };
  }
}
```

```

    }
  }
}

```

Then in our `AppModule` you'd define the import with the `forRoot()` method, which will return the NgModule with providers. In any other NgModule that imports `ForumsModule`, you'd use the `forChild()` method so you don't declare the provider again (thus creating a new instance):

```

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    ForumsModule.forRoot()
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

NgModules for grouping NgModules

You can combine a number of other NgModules into a single one, to make it easier to import and reuse. For example, in the [Clarity](#) project I work on, we have a number of NgModules that only export other NgModules. For instance, this is the main `ClarityModule` which actually reexports the other individual NgModules that contain each of the components:

```

@NgModule({
  exports: [
    ClrEmphasisModule, ClrDataModule, ClrIconModule, ClrModalModule,
    ↪ClrLoadingModule, ClrIfExpandModule, ClrConditionalModule,
    ↪ClrFocusTrapModule, ClrButtonModule, ClrCodeModule, ClrFormsModule,
    ↪ClrLayoutModule, ClrPopoverModule, ClrWizardModule
  ]
})

```

```
    ]  
  })  
  export class ClarityModule { }
```

This makes it easy to import many NgModules at once, but it does make it harder for the compiler to know which NgModules are used or not for tree shaking optimizations.

Summary

We've gone through a whirlwind tour of NgModules in Angular, and covered the key use cases. The [Angular documentation](#) about NgModules is quite in-depth as well, and if you get stuck I suggest reviewing the [FAQ](#).

Angular Testing: A Developer's Introduction

Michael Wanyoike

Chapter

10

In this guide, we'll look at how we can write automated tests in Angular 5 projects. Angular Testing is a core feature available in every project that was set up with either the [Angular CLI](#) or the [Angular quick start project](#).

The subject of Angular testing is vast, as it's a complex and very involved topic. It would require several chapters or a full-length course to cover it fully. So in this guide, I'll show you just the basics to get you started.

Prerequisites

At the time of writing, Angular 5.2 is the current stable version — which is what we'll be using here. This guide assumes you at least have a solid grasp of Angular 4+ fundamentals. It's also assumed you at least understand the concept or have some skills writing automated tests.

We'll base our testing examples on [Angular's official beginner tutorial](#) to demonstrate how to write tests for components and services. You can find the completed code with tests on our [GitHub repository](#). At the end of this guide, you should have the skills to implement several passing tests in Angular 5.



10-1. Angular testing: Passing Tests

Angular Testing Technologies

As you already know, an Angular project is made up of templates, components, services and modules. They all run inside what's known as the Angular environment. While it's possible to write isolated tests, you won't really know how your code will interact with other elements within the Angular environment.

Luckily, we have several technologies that can help us write such unit tests with the least amount of effort.

1. Angular Testing Utilities

This is a set of classes and functions that are needed to build a test environment for Angular code. You can find them on Angular's [api documentation](#). The most important of all is the `TestBed`. It's used to configure an Angular module just the same way as the `@NgModule` — except that it prepares the module for testing. It has a `configureTestingModule` function where you provide all the necessary dependencies for your component to function in a test environment. Here's an example of the `dashboard component` being prepared to run in a test environment. Several dependencies are needed by this component for the test to run:

```
TestBed.configureTestingModule({
  imports: [ RouterTestingModule ],
  declarations: [ DashboardComponent ],
  schemas: [ NO_ERRORS_SCHEMA ],
  providers: [
    {
      provide: HeroService,
      useClass: MockHeroService
    }
  ],
})
.compileComponents();
```

We'll look more closely at what's going on here a little further below.

2. Jasmine

Jasmine is the de facto framework for writing Angular tests. Basically, it's a testing framework that uses the behavior-driven notation. Writing a test in Jasmine is quite simple:

```
describe('createCustomer' () => {  
  
  it('should create new customer', (customer) => {  
    ...  
    expect(response).toEqual(newCustomer)  
  });  
  
  it('should not create customer with missing fields', () => {  
    ...  
    expect(response.error.message).toEqual('missing parameters')  
  });  
  
  it('should not create customer with existing record', () => {  
    ...  
    expect(response.error.message).toEqual('record already exists')  
  });  
});
```

The anatomy of a Jasmine test is made up of at least two elements: a `describe` function, which is a suite of tests, and an `it` function, which is the test itself. We normally use `describe` to indicate the function we're focusing on — for example, `createCustomer()`. Then, within the suite, we create multiple `it` tests. Each test puts the target function under a different condition in order to ensure it behaves as expected. You can refer to the [Jasmine docs](#) for more information.

3. Karma

Karma is a tool for executing source code against test code inside a browser environment. It supports the running of tests in each browser it's configured for. Results are displayed on both the command line and on the browser for the developer to inspect which tests have passed or failed. Karma also watches the

files and can trigger a test rerun whenever a file changes. At the root of the Angular project, we have the file `karma.conf` that's used to configure Karma. The contents should look something like this:

```
module.exports = function (config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine', '@angular/cli'],
    plugins: [
      require('karma-jasmine'),
      require('karma-chrome-launcher'),
      require('karma-jasmine-html-reporter'),
      require('karma-coverage-istanbul-reporter'),
      require('@angular/cli/plugins/karma')
    ],
    client:{
      clearContext: false // leave Jasmine Spec Runner output visible in browser
    },
    coverageIstanbulReporter: {
      reports: [ 'html', 'lcovonly' ],
      fixWebpackSourcePaths: true
    },
    angularCli: {
      environment: 'dev'
    },
    reporters: ['progress', 'kjhtml'],
    port: 9876,
    colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['Chrome'],
    singleRun: false
  });
};
```

Do check out [Karma's configuration documentation](#) to learn how to customize it. As you can see, Chrome is listed as the browser to use for running tests. You'll need to define an environment variable called `CHROME_BIN` that points to the location of your Chrome browser executable. If you're using Linux, just add this line to your `.bashrc` file:

```
export CHROME_BIN="/usr/bin/chromium-browser"
```

In order for Karma to run your tests, you must ensure the test files end with `.spec.ts`. You should note that Karma was designed to mostly run unit tests. To run end-to-end tests, we'll need another tool, Protractor, which we'll look into next.

4. Protractor

Protractor is an end-to-end test framework for Angular. It runs your tests inside a real browser, interacting with it as real person would. Unlike unit tests, where we test individual functions, here we test the entire logic. Protractor is able to fill in forms, click buttons and confirm that the expected data and styling is displayed in the HTML document. Just like Karma, Protractor has its own configuration file at the root of your Angular project, `protractor.conf`:

```
const { SpecReporter } = require('jasmine-spec-reporter');

exports.config = {
  allScriptsTimeout: 11000,
  specs: [
    './e2e/**/*.e2e-spec.ts'
  ],
  capabilities: {
    'bro'. './e2e/**/*.e2e-spec.ts'
  },
  directConnect: true,
  baseUrl: 'http://localhost:4200/',
  framework: 'jasmine',
  jasmineNodeOpts: {
    showColors: true,
    defaultTimeoutInterval: 30000,
    print: function() {}
  },
  onPrepare() {
    require('ts-node').register({
      project: 'e2e/tsconfig.e2e.json'
    });
  }
};
```

```
jasmine.getEnv().addReporter(new SpecReporter({ spec: { displayStacktrace:  
  ↪ true } }));  
}  
};
```

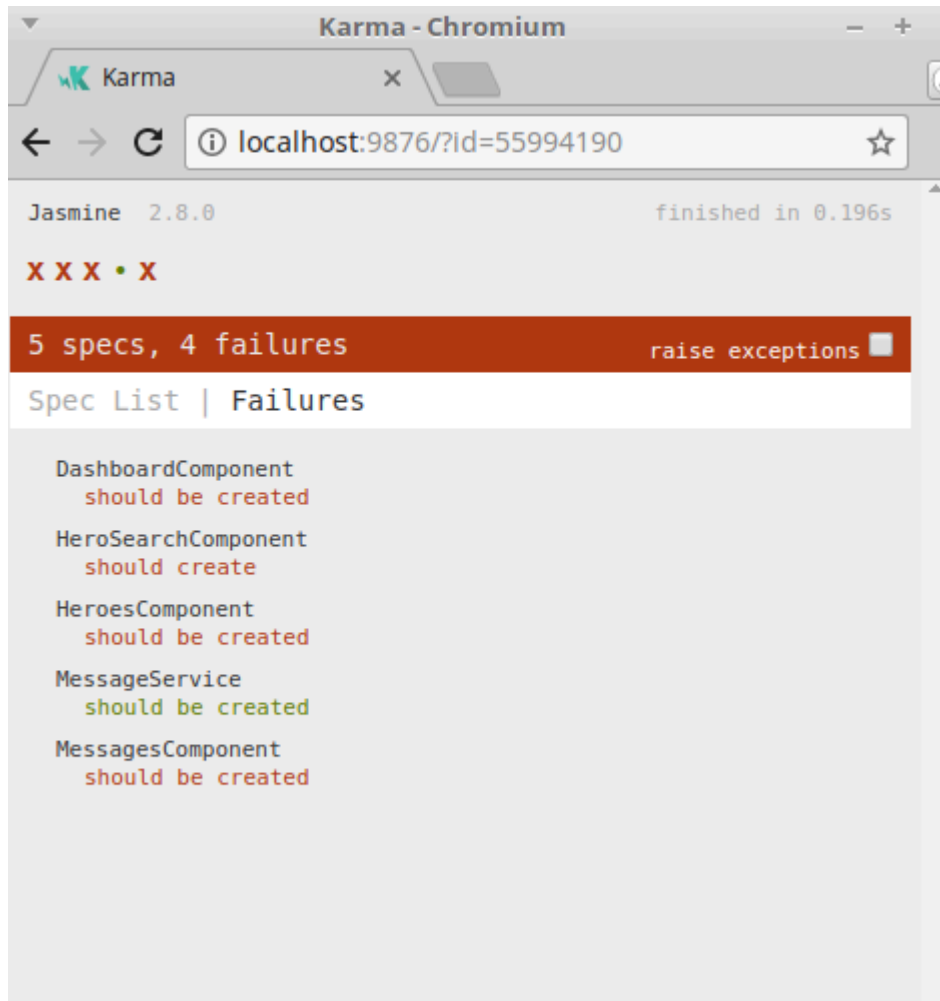
You can find the documentation for its configuration [here](#). Unlike Jasmine/Karma tests, Protractor tests are located outside the `src` folder, in a folder called `e2e`. We'll look into writing end-to-end tests later down the road. For now, let's start writing unit tests.

Writing Unit Tests

As mentioned earlier, Angular comes with just about everything you need to write automated tests for your project. To start testing, just run this:

```
ng test
```

Karma will spin up and run all available tests. Assuming you just completed the “Tour of Heroes” tutorial, you should have a similar report like this:



10-2. Angular testing: Failed tests

These tests get created when you generate components, services and classes using the `Angular CLI` tool. At the point of creation, the code in these tests were correct. However, as you added code to your component and services, the tests got broken. In the next section, we'll see how we can solve the broken tests.

Testing a Component

Unit testing a component can go in two ways. You can test it in isolation, or you can test it within an Angular environment to see how it interacts with its template and dependencies. The latter sounds hard to implement, but using

Angular Testing Utilities makes creating the test easier. Here's an example of the test code that's generated for you when you create a component using the `Angular CLI` tool:

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';

import { HeroesComponent } from './heroes.component';

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

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

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

  it('should be created', () => {
    expect(component).toBeTruthy();
  });
});
```

In the first `beforeEach()` function, we're using the `TestBed.configureTestingModule` function to create a module environment for testing the component. It's similar to `NgModules`, except that in this case we're creating a module for testing. In the second `beforeEach()` function, we create an instance of the `component-under-test`. Once we do this we can't configure the `TestBed` again, as an error will be thrown.

Finally we have the spec, `should be created`, where we confirm that the

`component` has been initialized. If this test passes, it means the component should run properly within an Angular environment. However, if it fails, it's likely the component has a certain dependency we haven't included in the test configuration. Let's look at how we can deal with different issues.

Testing a Component that Uses Another Component

While building a user interface in Angular, we often reference other components in a template file via the selector. Take a look at this example of

`dashboard.component.html`:

```
<h3>Top Heroes</h3>
...
</div>

<app-hero-search></app-hero-search>
```

In this example, we're referencing another component that has the selector `app-hero-search`. If you try to run the initial test as is, it will fail. This is because we haven't declared the referenced component in the test environment. In a unit test, we put all our focus on the component that we're testing. Other components are not of interest to us in a unit test. We have to assume they're working as expected. Including referenced components in our test may contaminate the results. To solve this problem, we can either mock the referenced component or simply ignore it using the `NO_ERRORS_SCHEMA` directive. Here's an example:

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { NO_ERRORS_SCHEMA } from '@angular/core';

import { DashboardComponent } from './dashboard.component';

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

```

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

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

it('should be created', () => {
  expect(component).toBeTruthy();
});
});

```

Now this test shouldn't have an issue with component dependencies. However, this test won't pass just yet, as there's another situation we have to deal with ...

Testing a Component that uses a Module

Let's examine `hero-detail.component.html` this time:

```

<div *ngIf="hero">
  <h2>{{ hero.name | uppercase }} Details</h2>
  <div><span>id: </span>{{hero.id}}</div>
  <div>
    <label>name:
      <input [(ngModel)]="hero.name" placeholder="name" />
    </label>
  </div>
  <button (click)="goBack()">go back</button>
  <button (click)="save()">save</button>
</div>

```

Here we're using the `ngModel` directive, which comes from the `FormsModule` library. In order to write a test that supports this module, we only need to import the `FormsModule` and include it in the `TestBed` configuration:

```
import { FormsModule } from '@angular/forms';
...
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ HeroDetailComponent ],
    imports: [ FormsModule ],
  })
  .compileComponents();
}));
...
```

That should fix the issue with the `FormsModule`. However, there are a couple more dependencies we need to specify in our test environment.

Testing a Component that Uses Routing Module

Let's examine `hero-detail.component.ts` constructor:

```
constructor(
  private route: ActivatedRoute,
  private location: Location,
  private heroService: HeroService
) {}
```

The component has `ActivatedRoute` and `Location` dependencies which deal with routing. In our test code, `hero-detail.component.spec.ts`, we could implement mock versions of the classes. However, I found the best solution was to import the `RouterTestingModule` like this:

```
import { RouterTestingModule } from '@angular/router/testing';
...
```

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ HeroDetailComponent ],
    imports: [ FormsModule, RouterTestingModule ],
  })
  .compileComponents();
}));
```

The `RouterTestingModule` easily solves the `ActivateRoute` and `Location` dependencies in our test code. The `RouterTestingModule` also handles other situations where routing is involved. Take a look at this code in

`dashboard.component.html` :

```
<h3>Top Heroes</h3>
<div class="grid grid-pad">
  <a *ngFor="let hero of heroes" class="col-1-4" routerLink="/detail/{{hero.id}}">
    <div class="module hero">
      <h4>{{hero.name}}</h4>
    </div>
  </a>
</div>
```

Notice we have a directive called `routerLink` . It's a directive provided by `AppRoutingModule` library. If run the dashboard test, it will fail due to this dependency. To fix it, just implement `RouterTestingModule` in `dashboard.component.spec.ts` the same way we've done for `hero-detail.component.spec.ts` .

Let's now look at how we can test components that depend on services.

Testing a Component that Uses Services

Every component needs at least a service to handle logic. There are a couple of ways of testing components that use services. Let's look at `message.service.ts` , which is being used by `message.component.ts` :

```
import { Injectable } from '@angular/core';

@Injectable()
export class MessageService {
  messages: string[] = [];

  add(message: string) {
    this.messages.push(message);
  }

  clear() {
    this.messages = [];
  }
}
```

The `MessageService` has a very simple implementation. It doesn't use any external dependency. While it's recommended to exclude external logic from unit tests, we'll make an exception here. I don't see the need for complicating our tests. For that reason, I think it's best to include the service in the test. Here's the test code for `message.component.spec.ts`:

```
import { MessageService } from '@services/message.service';
...

beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ MessagesComponent ],
    providers: [ MessageService ]
  })
  .compileComponents();
})))
```

Now let's look at another service, `hero-service.ts`:

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import { of } from 'rxjs/observable/of';
```

```

import { HttpClient, HttpHeaders } from '@angular/common/http';
import { catchError, map, tap } from 'rxjs/operators';
...
@Injectable()
export class HeroService {

  private heroesUrl = 'api/heroes';

  constructor(
    private http: HttpClient,
    private messageService: MessageService) { }

  /** GET heroes from the server */
  getHeroes (): Observable<Hero[]> {
    return this.http.get<Hero[]>(this.heroesUrl)
      .pipe(
        tap(heroes => this.log(`fetched ${heroes.length} heroes`)),
        catchError(this.handleError('getHeroes', []))
      );
  }

  getHero(id: number): Observable<Hero> {
    const url = `${this.heroesUrl}/${id}`;
    return this.http.get<Hero>(url).pipe(
      tap(_ => this.log(`fetched hero id=${id}`)),
      catchError(this.handleError<Hero>(`getHero id=${id}`))
    );
  }
  ...
}

```

The `HeroService` class contains quite a bit of logic — about 104 lines in total. It has multiple dependencies, including one to another service. Also, all its functions are asynchronous. Such complex code has high potential of contaminating our unit tests. For that reason, we should exclude its logic. We do that by creating a mock version of `hero.service.ts`. Just create a new file and call it `hero.service.mock.ts`. Mock its functions such that its core logic is stripped away:


```
import { Observable } from 'rxjs/Observable';
import { of } from 'rxjs/observable/of';
import { Hero } from '@models/hero.model';

export class MockHeroService {
  getHeroes(): Observable<Hero[]> {
    return of([]);
  }

  getHero() {
    return of({});
  }
}
```

You can see how much simpler the mock version is. It now has zero chances of contaminating our unit tests. To include it in our component spec files, we implement it like this:

```
import { HeroService } from '@services/hero.service';
import { MockHeroService } from '@services/hero.service.mock';
...

TestBed.configureTestingModule({
  declarations: [ HeroDetailComponent ],
  imports: [ FormsModule, RouterTestingModule ],
  providers: [
    {
      provide: HeroService,
      useClass: MockHeroService
    },
  ],
})
.compileComponents();
});
...

```

We use the `providers` option to inject the `MockHeroService` as our service. Implement this for all components' test code using the service.

Testing a Service

Now that we've dealt with some of the common scenarios that occur while testing components, let's look at how we can test services. Services perform the core logic of our applications, so it's very important we test their functions thoroughly. As mentioned earlier, Angular testing is a deep subject, so we're just going to scratch the surface here.

Open `hero.service.ts` and examine the functions. Let me highlight a couple:

```
...
/** GET heroes from the server */
getHeroes (): Observable<Hero[]> {
  return this.http.get<Hero[]>(this.heroesUrl)
    .pipe(
      tap(heroes => this.log(`fetched ${heroes.length} heroes`)),
      catchError(this.handleError('getHeroes', []))
    );
}

/** UPDATE: update selected hero on the server */
updateHero (hero: Hero): Observable<any> {
  return this.http.put(this.heroesUrl, hero, httpOptions).pipe(
    tap(_ => this.log(`updated hero id=${hero.id}`)),
    catchError(this.handleError<any>('updateHero'))
  );
}
...
```

Each function is made up of a few lines of code, but a lot is going on. To fully test each, we need to consider a number of scenarios. When we execute

`getHeroes()`, the server may possibly

- send back list of heroes
- send back an empty list
- throw an error
- fail to respond.

You may be able to think of more possible scenarios to add to the list. Now that we've considered possible scenarios, it's time to write the tests. Here's an example of how to write a *spec* for *HeroService*:

```
import { TestBed, inject } from '@angular/core/testing';
import { HttpClientModule, HttpClient, HttpResponse } from '@angular/common/http';
import { HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';

import { HeroService } from './hero.service';
import { MessageService } from './message.service';
import { Hero } from '@models/hero.model';

const mockData = [
  { id: 1, name: 'Hulk' },
  { id: 2, name: 'Thor'},
  { id: 3, name: 'Iron Man'}
] as Hero[];

describe('HeroService', () => {

  let service;
  let httpTestingController: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [
        HttpClientTestingModule
      ],
      providers: [HeroService, MessageService]
    });
    httpTestingController = TestBed.get(HttpTestingController);
  });

  beforeEach(inject([HeroService], s => {
    service = s;
  }));

  beforeEach(() => {
    this.mockHeroes = [...mockData];
    this.mockHero = this.mockHeroes[0];
```

```

    this.mockId = this.mockHero.id;
  });

  const apiUrl = (id: number) => {
    return `${service.heroesUrl}/${this.mockId}`;
  };

  afterEach(() => {
    // After every test, assert that there are no more pending requests.
    httpTestingController.verify();
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });

  describe('getHeroes', () => {

    it('should return mock heroes', () => {
      service.getHeroes().subscribe(
        heroes => expect(heroes.length).toEqual(this.mockHeroes.length),
        fail
      );
      // Receive GET request
      const req = httpTestingController.expectOne(service.heroesUrl);
      expect(req.request.method).toEqual('GET');
      // Respond with the mock heroes
      req.flush(this.mockHeroes);
    });
  });

  describe('updateHero', () => {

    it('should update hero', () => {
      service.updateHero(this.mockHero).subscribe(
        response => expect(response).toEqual(this.mockHero),
        fail
      );
      // Receive PUT request
      const req = httpTestingController.expectOne(service.heroesUrl);
      expect(req.request.method).toEqual('PUT');
    });
  });

```

```

        // Respond with the updated hero
        req.flush(this.mockHero);
    });
});

describe('deleteHero', () => {

    it('should delete hero using id', () => {
        const mockUrl = apiUrl(this.mockId);
        service.deleteHero(this.mockId).subscribe(
            response => expect(response).toEqual(this.mockId),
            fail
        );
        // Receive DELETE request
        const req = httpTestingController.expectOne(mockUrl);
        expect(req.request.method).toEqual('DELETE');
        // Respond with the updated hero
        req.flush(this.mockId);
    });

    it('should delete hero using hero object', () => {
        const mockUrl = apiUrl(this.mockHero.id);
        service.deleteHero(this.mockHero).subscribe(
            response => expect(response).toEqual(this.mockHero.id),
            fail
        );
        // Receive DELETE request
        const req = httpTestingController.expectOne(mockUrl);
        expect(req.request.method).toEqual('DELETE');
        // Respond with the updated hero
        req.flush(this.mockHero.id);
    });
});
});
});

```

This is just a sample of how we should write a test for a service that interacts with the `HttpClientModule`. Examine each test and take note that we're using `HttpTestingController` class to intercept requests. In this test, we're controlling the inputs and outputs to create different scenarios. The main purpose of these tests is to ensure that our service methods are able to handle each scenario

gracefully. Note that we haven't fully implemented all the tests required for `hero.service.spec.ts`, as it's beyond the scope of this guide.

There are more topics that we still need to look at before the end of this guide.

End-to-end Angular Testing

Unit tests ensure components and services run correctly in a controlled test environment. However, there's no guarantee that components and services will interact with each other within the Angular environment. That's why we need to perform end-to-end testing. An end-to-end test is one that simulates human testing. In other words, the tests are designed to interact with our application the same way we do — via the browser interface.

For our tour of heroes application, there's a number of use cases we can test for, such as ensuring that —

- five heroes are displayed on dashboard component
- all heroes are displayed on heroes component
- navigation links aren't broken
- a new hero can be created
- the hero can be updated
- the hero can be deleted.

And you can keep on adding to this list as more features get implemented. An end-to-end test ideally has two parts.

The first part is a helper file that provides helper functions specific to a component. Here's an example of `app.po.ts`:

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

export class AppPage {
```

```

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

getParagraphText() {
  return element(by.css('app-root h1')).getText();
}
}

```

Once you've defined your helper functions, you can easily access them while writing an e2e test. Here's an example of `e2e/app.e2e.spec.ts`:

```

import { AppPage } from './app.po';

describe('angular-tour-of-heroes App', () => {
  let page: AppPage;

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

  it('should display welcome message', () => {
    page.navigateTo();
    expect(page.getParagraphText()).toEqual('Welcome to app!');
  });
});

```

To run this test, simply execute the following command:

```
ng e2e
```

You might need an internet connection if this is the first time you're executing this command. Once the test is complete, you'll most likely get a failed message that looks something like this:

```
angular-tour-of-heroes App
```

```

X should display welcome message
  - Expected 'Tour of Heroes' to equal 'Welcome to app!'.

```

Let's fix the error as follows. I've also added one more test just to make sure the redirection we specified in `app-routing.module.ts` works:

```

import { AppPage } from './app.po';
import { browser } from 'protractor';

describe('angular-tour-of-heroes App', () => {
  let page: AppPage;

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

  it('should redirect to dashboard', async () => {
    page.navigateTo();
    const url = await browser.getCurrentUrl();
    expect(url).toContain('/dashboard');
  });

  it('should display welcome message', () => {
    page.navigateTo();
    expect(page.getParagraphText()).toEqual('Tour of Heroes');
  });
});

```

Run the test again. We should now have passing tests:

```

angular-tour-of-heroes App
  ✓ should redirect to dashboard
  ✓ should display welcome message

```

Watching `e2e` tests run is an awesome feeling. It gives you the confidence that your application will run smoothly in production. Now that you have had a taste of `e2e`, it's time to move on to another cool testing feature.

Code Coverage

One of our biggest question as developers is “have we tested enough code?” Luckily, we have tools that can generate “code coverage” to determine how much of our code is tested. To generate the report, just run the following:

```
ng test --watch=false --code-coverage
```

A coverage folder will be created at the root of your Angular project. Navigate inside the folder and you'll find `index.html`. Open it using a web browser. You should see something like this:

All files

82.29% Statements 144/175 37.5% Branches 3/8 70.18% Functions 49/57 82.98% Lines 117/141

| File ▲ | | Statements | | Branches | | Functions | | Lines | |
|--------------------------------|------------------------|------------|-------|----------|-----|-----------|-------|--------|-------|
| src | <div><div></div></div> | 100% | 8/8 | 100% | 0/0 | 100% | 0/0 | 100% | 8/8 |
| src/app | <div><div></div></div> | 100% | 6/6 | 100% | 0/0 | 100% | 2/2 | 100% | 4/4 |
| src/app/components/dashboard | <div><div></div></div> | 100% | 14/14 | 100% | 0/0 | 100% | 5/5 | 100% | 11/11 |
| src/app/components/hero-detail | <div><div></div></div> | 84.62% | 22/26 | 100% | 0/0 | 62.5% | 5/8 | 86.36% | 19/22 |
| src/app/components/hero-search | <div><div></div></div> | 87.5% | 14/16 | 100% | 0/0 | 60% | 3/5 | 84.62% | 11/13 |
| src/app/components/heroes | <div><div></div></div> | 64% | 16/25 | 0% | 0/2 | 55.56% | 5/9 | 68.42% | 13/19 |
| src/app/components/messages | <div><div></div></div> | 100% | 8/8 | 100% | 0/0 | 100% | 3/3 | 100% | 6/6 |
| src/app/models | <div><div></div></div> | 100% | 3/3 | 100% | 0/0 | 100% | 1/1 | 100% | 2/2 |
| src/app/services | <div><div></div></div> | 76.81% | 53/69 | 50% | 3/6 | 66.67% | 16/24 | 76.79% | 43/56 |

10-3. Angular testing: Code coverage report

I won't go into much detail here, but you can see some classes have been tested fully while others not completely. Due to time and availability of resources, it often isn't always possible to implement 100% test coverage. However, you can decide with your team on what should be the minimum. To specify the minimum, use `karma.conf` to configure your code coverage settings like this:

```
coverageIstanbulReporter: {  
  reports: [ 'html', 'lcovonly' ],  
  fixWebpackSourcePaths: true,  
  thresholds: {  
    statements: 80,  
    lines: 80,  
    branches: 80,  
    functions: 80  
  }  
}
```

The above threshold value specifies a minimum of 80% to be covered by unit tests.

Additional Utilities

We've now covered the basics of Angular testing. However, we can improve our code quality by going a few steps further.

1. Linting

Angular comes with a tool for performing code linting. Just execute the following code to do a lint check on your project:

```
ng lint
```

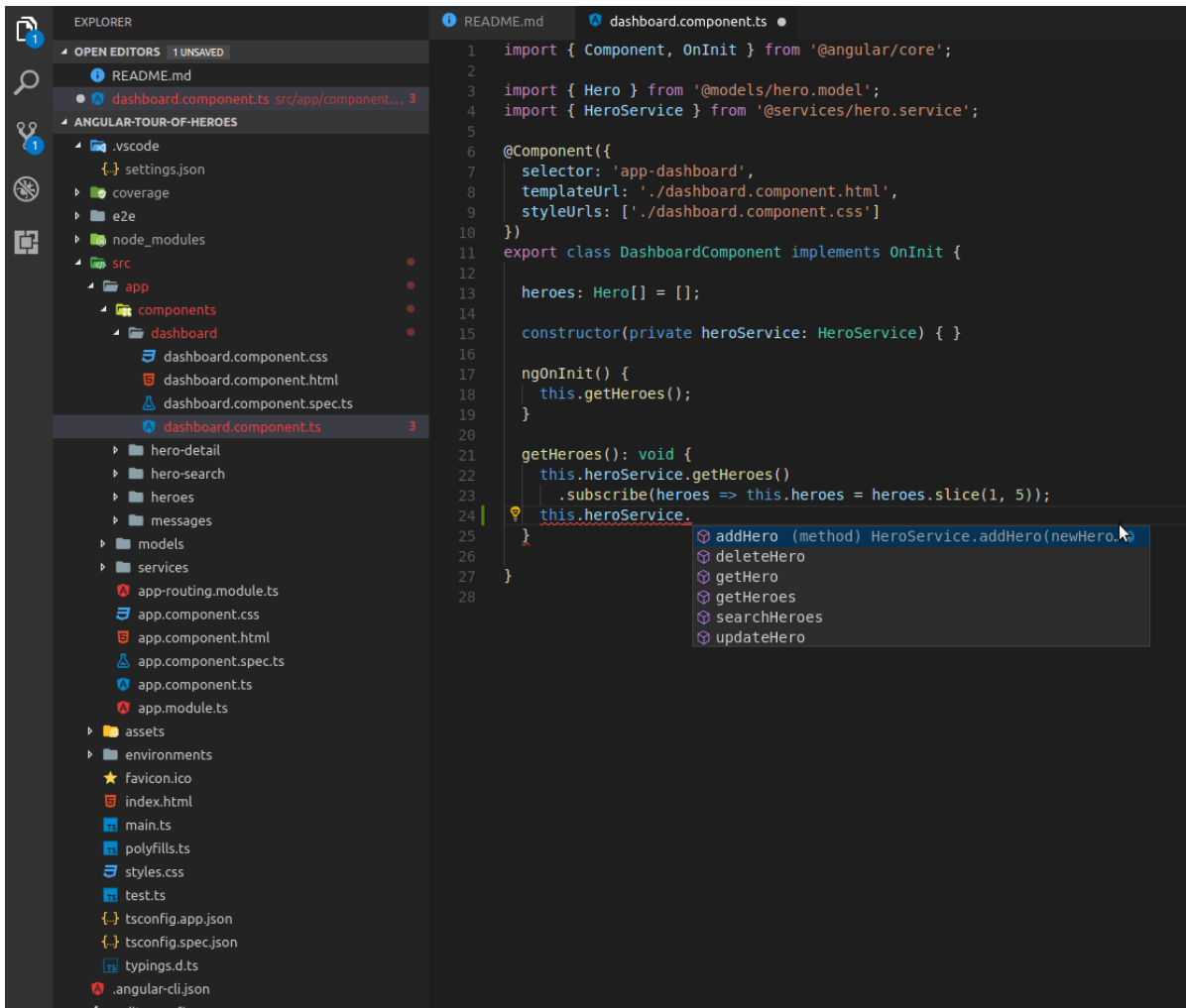
This command will spit out warnings about your code — for example, where you forgot to use a semicolon, or you used too many spaces. The command will also help identify unused code and certain mistakes in your statements. Using this command often will ensure that everyone in your team writes code using a consistent style. You can further customize the lint options in the `tslint.json` file.

2. Intelligent Code Editors

When it comes to code editors and IDEs, my personal favorites are Atom and Sublime Text. However, I recently discovered Visual Studio Code, which has more appealing features. It's a free code editor that can run in Windows, macOS and Linux. It borrows a lot from Atom, except it has additional features I'd like to highlight:

- Intellisense
- Error highlighting
- Modern Angular extensions

Currently, neither Atom nor Sublime Text have these features, while they're built into VSCode. You only need to install the required language extension. The Intellisense feature lists options for you as you type your code. It's like autocomplete but with a specific list of syntactically correct options. With this feature it's hard to make a syntax mistake. You also get to see a function's documentation, allowing you to see the return type and the required inputs.



10-4. Angular testing: Visual Studio Code

Visual Studio Code also has a proper error highlighting feature. It not only checks for syntax errors, but also ensures assignments have the right type. For example, if you try to assign an array to the result of an Observable function, an error will be highlighted for you. VSCode also has Angular extensions compatible with Angular 5.

Having an IDE that checks your code for errors as you type is great for productivity. It helps you spend less time fixing errors that you would otherwise have made. There may be other code editors that can accomplish the same, but for now I'm recommending Visual Studio Code for Angular projects.

3. Continuous Integration

Continuous Integration (CI) is the process of automating testing and builds. As developers, we often work in isolation for a couple of weeks or more. By the time we merge changes into the master branch, a lot of errors and conflicts are produced. This can take a lot of time to fix.

CI encourages developers to write tests and commit tasks often in smaller bits. The CI server will automatically build and run tests, helping developers catch errors early, leading to less conflicts and issues. There are many CI solutions available for Angular developers. Check out SitePoint's tutorial on [testing Jasmine and Karma on Travis](#).

Wrapping Up

We have access to tons of information about automated tests, along with frameworks for test-driven development, that help us to write tests. However, there are a couple reasons why we shouldn't always write tests:

- 1 Don't write tests for a new application. The scope of the project will change rapidly, depending on what the client wants or how the market responds.
- 2 Writing tests requires more time in addition to implementing features. It also requires time to maintain when the feature scope changes. If your budget is low, it's okay to skip writing tests. Be practical with the resources you have.

So that leaves the question of when it is the right time to write tests. Here are some pointers:

- 1 You've completed the prototype phase and you've pinned down the core features of your application.

- 2 Your project has sufficient funding.

Now, assuming you've decided to enforce TDD, there are plenty of benefits to be reaped:

- 1 Writing code that can be tested means you're writing better quality code.
- 2 As a developer, you'll have more confidence releasing your latest version into production.
- 3 Writing tests is a way of documenting your code. This means future developers will have an easier time upgrading legacy code.
- 4 You don't need to hire someone for quality control, as your CI server will do that work for you.

If you decide to skip tests completely for your product-ready application, be prepared to face angry and disappointed customers in the future. The number of bugs will increase exponentially as your codebase increases in size.

Hopefully this has been a useful introduction to Angular testing for you. If you want to learn more, I recommend you stick first to the official Angular 5 documentation. The majority of information out there is for older versions of Angular, unless stated otherwise.

Creating UIs with Angular Material Design Components

Ahmed Bouchefra

Chapter

11

In this tutorial, I'll introduce you to Material Design in Angular, then we'll look at how to create a simple Angular application with a UI built from various Angular Material components.

The widespread adoption of component-based frameworks such as Angular, React and Vue.js has seen a growing number of pre-built UI component collections become available. Using such collections can help developers to quickly create professional-looking applications.

What's Material Design?

Material Design (codenamed **Quantum Paper**) is a visual language that can be used to create digital experiences. It's a set of principles and guidelines across platforms and devices for interactivity, motion and components that simplify the design workflow for teams designing their product.

The Material components allow you to create professional UIs with powerful modularity, theming and customization features.

Introduction to Angular Material

Angular Material is the implementation of Material Design principles and guidelines for Angular. It contains various UI components, such as:

- form controls (input, select, checkbox, date picker and sliders etc.),
- navigation patterns (menus, sidenav and toolbar)
- layout components (grids, cards, tabs and lists)
- buttons
- indicators (progress bars and spinners)
- popups and modals
- data tables with headers and pagination etc.

Requirements

Before you can continue with this tutorial, you need to make sure you have a development machine with Node (6.9.0+) and NPM (3+) installed.

You also need to have the Angular CLI installed. If that's not the case, simply run the following command in your terminal or command prompt:

```
npm install -g @angular/cli
```

Create the Project with the Angular CLI

Let's now create the Angular project using the Angular CLI. Simply run the following command:

```
ng new angularmaterialdemo
```

You can then serve your application by running:

```
cd angularmaterialdemo  
ng serve
```

The application will be running at `http://localhost:4200`.

Since we're going to demonstrate different Angular Material components, we need to create a few Angular components and routing configurations for a simple demo application, so go ahead and use the CLI to generate the components:

```
ng g component login  
ng g component CustomerList  
ng g component CustomerCreate
```

Next, open `src/app/app.module.ts` and add the router configuration:

```
/*...*/

import { RouterModule, Routes } from '@angular/router';

/*...*/

const appRoutes: Routes = [
  { path: 'customer-list', component: CustomerListComponent },
  { path: 'customer-create', component: CustomerCreateComponent },
  {
    path: 'login',
    component: LoginComponent
  },
  { path: '',

  redirectTo: '/login',

  pathMatch: 'full'

},
];
```

Getting Started with Angular Material

Now that we have a basic application, let's get started by installing Angular Material and its different dependencies to enable the different features such as gestures and animations.

Installing Angular Material and Angular CDK

Let's start by installing Angular Material and Angular CDK from npm.

Head back to your terminal and run the following command:

```
npm install --save @angular/material @angular/cdk
```

Adding HammerJS for Gestures Support

Components such as `mat-slide-toggle`, `mat-slider` and `matTooltip` require the HammerJS library for gestures support, so you need to install it for getting the full features of these components. Simply run the following command in your terminal:

```
npm install --save hammerjs
```

Next, open `src/main.js` (the entry point of your application) and import `hammerjs`

```
import 'hammerjs';
```

Adding a Theme

Angular Material has a bunch of pre-built themes. To use a theme, you simply need to import it in `styles.css`:

```
@import "~@angular/material/prebuilt-themes/indigo-pink.css"
```

You can find more information about theming in this [guide](#).

Adding Angular Material Icons

Angular Material comes with a `mat-icon` component for icons, so you need to load the icon font before you can use it.

Add the following tag to your `index.html` file:

```
<link href="https://fonts.googleapis.com/icon?family=Material+Icons"
      rel="stylesheet">
```

Using Animations with Angular Material Components

The last thing is enabling animations. Some components rely on the Angular animations module for advanced transitions, so you need to install the `@angular/animations` module and include the `BrowserAnimationsModule` in your application module configuration.

First, head back to your terminal and run the following command:

```
npm install --save @angular/animations
```

Next, open `src/app/app.module.ts` and add the following code:

```
/* ... */
import {BrowserAnimationsModule} from '@angular/platform-browser/animations';

@NgModule({
  /*...*/
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
  ],
})
export class AppModule { }
```

Importing Angular Material Components

Before you can use any Angular Material component, you'll have to import its module. Each component has its own module so you can include only the components you're going to use.

Another approach is to create a separate module and import all the Angular Material components you need to use and then simply include this module in your application module.

So go ahead and create a `src/app/material.module.ts` file, and then add the following content:

```
import { NgModule } from '@angular/core';
import { MatNativeDateModule, MatSnackBarModule, MatIconModule, MatDialogModule,
  ↳ MatButtonModule, MatTableModule, MatPaginatorModule, MatSortModule, MatTabsModule,
  ↳ MatCheckboxModule, MatToolbarModule, MatCard, MatCardModule, MatFormField,
  ↳ MatFormFieldModule, MatProgressSpinnerModule, MatInputModule }
  ↳ from '@angular/material';
import { MatDatepickerModule } from '@angular/material/datepicker';
import { MatRadioModule } from '@angular/material/radio';
import { MatSelectModule } from '@angular/material/select';
import { MatSliderModule } from '@angular/material/slider';
import { MatDividerModule } from '@angular/material/divider';

@NgModule({
  imports: [MatTabsModule, MatDividerModule, MatSliderModule, MatSelectModule,
    ↳ MatRadioModule, MatNativeDateModule, MatDatepickerModule, MatSnackBarModule,
    ↳ MatIconModule, MatDialogModule, MatProgressSpinnerModule, MatButtonModule,
    ↳ MatSortModule, MatTableModule, MatTabsModule, MatCheckboxModule, MatToolbarModule,
    ↳ MatCardModule, MatFormFieldModule, MatProgressSpinnerModule, MatInputModule,
    ↳ MatPaginatorModule],
  exports: [MatTabsModule, MatDividerModule, MatSliderModule, MatSelectModule,
    ↳ MatRadioModule, MatNativeDateModule, MatDatepickerModule, MatSnackBarModule,
    ↳ MatIconModule, MatDialogModule, MatProgressSpinnerModule, MatButtonModule,
    ↳ MatSortModule, MatCheckboxModule, MatToolbarModule, MatCardModule, MatTableModule,
    ↳ MatTabsModule, MatFormFieldModule, MatProgressSpinnerModule, MatInputModule,
    ↳ MatPaginatorModule],
})

export class MyMaterialModule { }
```

Next, include this module in `src/app/app.module.ts` :

```
import { MyMaterialModule } from './material.module';

/*...*/

@NgModule({

/*...*/

imports: [

/*...*/
MyMaterialModule,
],
/*...*/
})

export class AppModule { }
```

That's it: you can now use the imported Angular Material components in your Angular application.

Create the UI for the Application

You've previously created the application components and added the router configuration. Now let's build the UI for the different components using Angular Material components.

Building the UI for AppComponent

Go ahead and open `src/app/app.component.html`, then add:

- a Material toolbar with three Material buttons (`mat-button`) for links to the app components
- a router outlet `<router-outlet>` where the components matching a router path will be inserted.

This is the HTML code for the component:

```
<mat-toolbar color="primary" class="fixed-header">
  <mat-toolbar-row>
    <span></span>
    <a mat-button routerLink="/customer-list" routerLinkActive="active">Customers</a>
    <a mat-button routerLink="/customer-create">Create Customer</a>
    <a mat-button routerLink="/login">Login</a>
  </mat-toolbar-row>
</mat-toolbar>
<main>
  <router-outlet style="margin-top: 80px;"></router-outlet>
</main>
```

By adding some styling, this is how the toolbar looks:



11-1. Angular Material: a first look at the toolbar

To create a toolbar, you use the `<mat-toolbar>` component and then you create one or multiple rows inside the toolbar using the `<mat-toolbar-row>` component.

Notice that we have imported the `MatToolbarModule` and `MatButtonModule` modules from `@angular/material` in the `material.module.ts`.

You can also make use of other navigation components such as menus and sidebars.

Building the UI for the Login Component

After creating the UI for the root component of our application, let's create the UI for the login component.

Open `src/app/login/login.component.html` and add the following HTML code.

Add an Angular Material card to contain the login form:

```
<mat-card class="login-card">
  <mat-card-header>
    <mat-card-title>Login</mat-card-title>
  </mat-card-header>
  <mat-card-content>
    <!-- The form goes here -->
  </mat-card-content>
```

A `<mat-card>` component serves as a single-object container for text, photos, and actions. Find more details about cards from the [docs](#).

Next in the content section of the card, add the HTML form:

```
<form class="login-form">
  <!-- Form elements are here -->
</form>
```

Now let's add two inputs to get the user's username and password credentials:

```
<mat-form-field>
  <input matInput placeholder="Username" [(ngModel)]="username" name="username"
  required>
</mat-form-field>
<mat-form-field>
  <input matInput placeholder="Password" [(ngModel)]="password" type="password"
  name="password" required>
</mat-form-field>
```

Many Angular Material components need to be wrapped inside a `<mat-form-field></mat>` component to have common [Text field](#) styles such as the underline, floating label, and hint messages.

These are the components that are designed to be wrapped inside

`<mat-form-field>`:

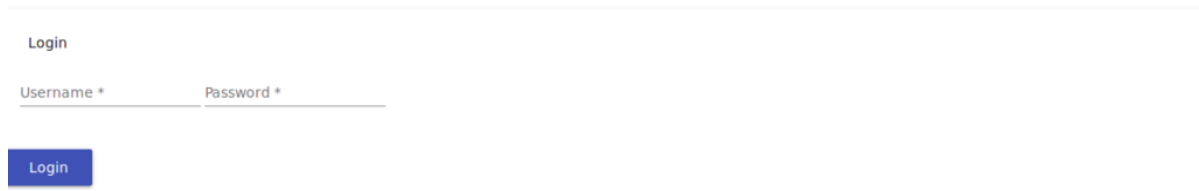
- `<input matInput>` and `<textarea matInput>`
- `<mat-select>`
- `<mat-chip-list>`.

Finally, in the actions section of the card, let's add an Angular Material button for the login action:

```
<mat-card-actions>
  <button mat-raised-button (click)="login()" color="primary">Login</button>
</mat-card-actions>
</mat-card>
```

To create an Angular Material button, you simply use native HTML `<button>` and `<a>` elements and add Material attributes such as `mat-button` and `mat-raised-button`. For more details, check the [docs](#).

This is how the login form looks:



11-2. The login form

If the login is successful (*username: demo, password: demo*) the user will be redirected to the *CustomerList* component. Otherwise, the user will get an error message: “Your login information are incorrect!”

Create the Error Modal Dialog

`MatDialog` can be used to create and open modal dialogs. The dialog requires a

component to be loaded, so first create an Angular component that displays the error message that gets passed.

Create `src/app/error.component.ts`, then add the following code:

```
import {Component, Inject, Injectable} from '@angular/core';

import {MatDialogRef, MAT_DIALOG_DATA, MatDialog} from '@angular/material';

@Component({
  templateUrl: 'error.component.html'
})
export class ErrorComponent {
  constructor(private dialogRef: MatDialogRef<ErrorComponent>,
    ↪@Inject(MAT_DIALOG_DATA) public data: any) {
  }
  public closeDialog() {
    this.dialogRef.close();
  }
}
```

This component will be created and opened with `MatDialog` so it can inject `MatDialogRef`, which provides a handle on the opened dialog and can be used to close the dialog and receive notifications when the dialog gets closed. (The `closeDialog()` method uses this reference to close the dialog.)

Our error component needs a way to get the error message to be displayed for the user. (The message will be passed when the component is opened using `MatDialog` open method.) You can access the data in your dialog component using the `MAT_DIALOG_DATA` injection token that can be imported from `'@angular/material'` and then injected into the component using `@Inject()`.

Next, create `src/app/error.component.html` and add the following code:

```
<h2 mat-dialog-title>Error</h2>
```

```
<mat-dialog-content class="error">{{data.message}}</mat-dialog-content>
<mat-dialog-actions>
<button mat-raised-button (click)="closeDialog()">Ok</button>
</mat-dialog-actions>
```

We're using many available directives for structuring the dialog components:

- `mat-dialog-title` : used for the dialog title, and needs to be applied to heading elements `<h1>` , `<h2>` etc.
- `<mat-dialog-content>` : used for the scrollable content of the dialog
- `<mat-dialog-actions>` : used as a container for dialog actions.

Next, open `src/app/app.module.ts` and add this component to the module:

```
@NgModule({
  declarations: [
    /*...*/
    ErrorComponent
  ],
  imports: [/*...*/
  ],
  entryComponents: [ErrorComponent],
  /*...*/
})

export class AppModule { }
```

Now let's add the `Login()` method (to trigger the error dialog) to the `LoginComponent` :

```
import { Component, OnInit } from '@angular/core';
import { MatDialog, MatDialogRef } from '@angular/material';
```

```

/* ... */

@Component({
  /* ... */
})
export class LoginComponent{

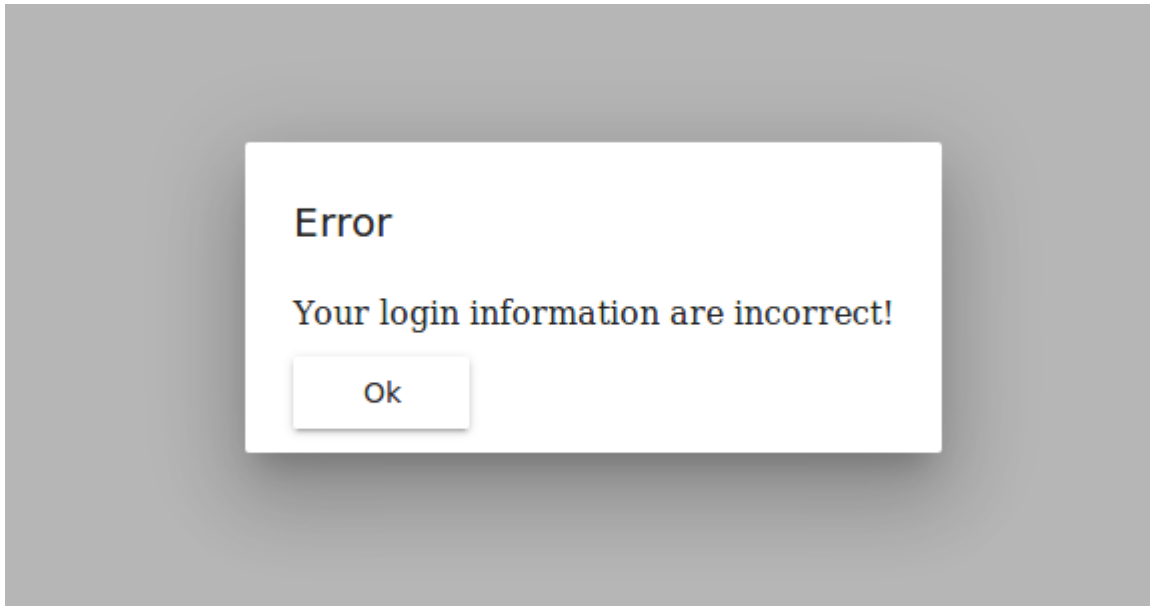
  public username: string = "";
  public password: string = "";

  constructor(private dialog: MatDialog, private router: Router) { }
  login(){
    if(this.username === "demo" && this.password === "demo")
    {
      this.router.navigate(['customer-list']);
    }
    else
    {
      this.dialog.open(ErrorComponent,{ data: {
        message: "Your login information are incorrect!"
      }});
    }
  }
}

```

We simply inject the `MatDialog` component and use it to open a dialog (if the user credentials aren't correct) with the `ErrorComponent` component and a config object holding the data that will be passed. The `open()` method returns an instance of `MatDialogRef`.

This is a screenshot of the error dialog:



11-3. A screenshot of the error dialog

You can find more information about dialogs from the [docs](#).

Building the UI for the CustomerList Component

For the `CustomerList` component we'll make use of different Angular Material components, most importantly:

- [Paginator](#)
- [Table](#)
- [Progress spinner](#)

Open `src/app/customer-list/customer-list.component.html` and add the following markup:

```
<div class="mat-elevation-z8">
  <!-- Other contents here -->
</div>
```

the `mat-elevation-z8` class is an Angular Material elevation class that allows

you to add separation between elements along the z-axis. You can find more details from this [link](#).

Adding a Loading Spinner

We use `<mat-spinner>` to display a spinner when data is still loading, which we're simulating using a `setTimeout()` function:

```
<mat-spinner [style.display]="loading ? 'block' : 'none'"></mat-spinner>
```

If the `loading` variable equates to `true`, the CSS `display` property gets assigned the `block` value. Otherwise, it gets assigned `none` which makes it disappear.

You need to add a loading variable to the component class, which initially takes a value of `true`.

```
loading = true;
/*...*/
constructor() {}
ngOnInit(): void {

  setTimeout(() => {

    this.loading = false;

  }, 2000);

  /*...*/
}
```

Adding a Data Table with a Data Source

We then create a Material data table using `<mat-table>` that gets displayed only if `loading` equals `false`:

```
<mat-table [style.display]="loading === false ? 'block' : 'none'" #table
  ↳ [dataSource]="dataSource">
<!-- Other contents here -->
</mat-table>
```

The data table takes a `dataSource` property that's used to provide data for the table. You can simply pass a data array to the table, but for real-world use cases you need to use an advanced data source such as `MatTableDataSource`, provided by Angular Material, that encapsulates logic for advanced operations such as pagination, sorting and filtering:

```
export class CustomerListComponent implements OnInit {

  customers: Customer[] = [
    { id:1, name:'Customer 001',job:'Programmer'},
    /*...*/
  ];

  dataSource = new MatTableDataSource<Customer>(this.customers);
```

Adding Column Templates

For each column definition, you need to provide a unique name and the content for its header and row cells. For example, this is the column for displaying the customer ID:

```
<ng-container matColumnDef="id">
<mat-header-cell *matHeaderCellDef> CustomerId </mat-header-cell>
<mat-cell *matCellDef="let customer"> {{customer.id}} </mat-cell>
</ng-container>
```

This is a simple column definition with the name `id`. The header cell contains the text `CustomerId` and each row cell will render the `id` property of each row's data.

In a similar way, you can add the other column definitions.

For the operations column, we're using two Angular Material icons: `delete` and `edit` wrapped with Material buttons:

```
<ng-container matColumnDef="operations">
  <mat-header-cell *matHeaderCellDef mat-sort-header> Operations </mat-header-cell>

  <mat-cell *matCellDef="let element"> <button mat-button color="primary"
    ↳(click)="deleteCustomer(element.id)"><mat-icon>delete</mat-icon>

    </button> <button mat-button color="primary" (click)="editCustomer(element.id)"><mat-icon>edit</mat-icon>

  </mat-cell>
</ng-container>
```

Adding Row Templates

After defining column templates, you need to define the row templates, so first you need to add a variable that contains the list of columns you have to the component:

```
displayedColumns = ['id', 'name', 'job', 'operations'];
```

Next you need to add `<mat-header-row>` and `<mat-row>` to the content of your `<mat-table>` and provide your column list as inputs:

```
<mat-header-row *matHeaderRowDef="displayedColumns"></mat-header-row>
<mat-row *matRowDef="let row; columns: displayedColumns;"></mat-row>
```

Adding Pagination

You can add pagination to your data table by simply adding a `<mat-paginator>`

component after `<mat-table>` :

```
<mat-paginator #paginator [style.display]="loading === false ? 'block' : 'none'"
  ↳[pageSize]="5"></mat-paginator>
```

One benefit of using `MatTableDataSource` is that you get pagination out of the box by simply providing `MatPaginator` to your data source.

First make sure you import the `MatPaginatorModule` in `material.module.ts`. Then you need to get the paginator directive:

```
@ViewChild(MatPaginator) paginator: MatPaginator;
```

Finally, you just pass the paginator to the table's data source:

```
ngAfterViewInit() {
  this.dataSource.paginator = this.paginator;
}
```

Adding Sorting

Using `MatTableDataSource` you can also have sorting out of the box by only adding a few things.

First, make sure you have `MatSortModule` imported in your module (`material.module.ts`).

Next, add the `matSort` directive to the `<mat-table>` component and then add `mat-sort-header` to each column header cell that needs to have sorting.

For example, let's add sorting to the `name` column:

```
<ng-container matColumnDef="name">
<mat-header-cell *matHeaderCellDef mat-sort-header> Name </mat-header-cell>
<mat-cell *matCellDef="let customer"> {{customer.name}} </mat-cell>
</ng-container>
```

Finally, you need to provide the `MatSort` directive to the data source and it will automatically have sorting:

```
import {MatTableDataSource, MatPaginator, MatSort} from '@angular/material';
/*...*/
export class CustomerListComponent implements OnInit {

  @ViewChild(MatSort) sort: MatSort;









  ngAfterViewInit() {
    /*...*/
    this.dataSource.sort = this.sort;
  }
}
```

You can also add filtering and selection to your table. Check the [docs](#) for more information.

Using `MatTableDataSource` provides you with many built-in features out of the box, but it only supports client-side pagination. For server-side pagination or other custom features, you need to create your [custom data source](#).

In the case of a custom data source, you'll need to listen to the paginator's `(page)` event to implement pagination and to the sort's `(matSortChange)` event for implementing data sorting.

This is a screenshot of the `CustomerList` component:

| Customerid ↑ | Name | Job | Operations | |
|--------------|--------------|------------|---|---|
| 11 | Customer 011 | Programmer |  |  |
| 12 | Customer 012 | Programmer |  |  |
| 13 | Customer 013 | Programmer |  |  |
| 14 | Customer 014 | Programmer |  |  |

Items per page: 5 11 - 14 of 14 < >

11-4. The CustomerList component

Adding MatSnackBar Notifications

First make sure you've imported `MatSnackBarModule` into your module. Next, import and inject `MatSnackBar` into your component, then simply call the `open()` method of the `MatSnackBar` instance:

```
import {MatSnackBar} from '@angular/material';
/*...*/
constructor(public snackBar: MatSnackBar) {}

deleteCustomer(id){
  let snackBarRef = this.snackBar.open(`Deleting customer #${id}`);
}
editCustomer(id){
  let snackBarRef = this.snackBar.open(`Editing customer #${id}`);
}
```

Building the UI for the CustomerCreate Component

For the `CustomerCreate` component, we'll use a bunch of Angular Material components, such as:

- the card component (`<mat-card>`)
- the tabs component (`<mat-tab>` and `<mat-tab-group>`)
- the label (`<mat-label>`) and input (`<input matInput>` and `<textarea`

`matInput` components

- the form field (`<mat-form-field>`) component
- the checkbox (`<mat-checkbox>`) component
- the date picker (`<mat-datepicker>`) component
- the radio button (`<mat-radio-button>`) component
- the select (`<mat-select>`) component.

So open `src/app/customer-create/customer-create.component.html` and start by adding the card component that holds the customer creation form:

```
<mat-card class="my-card">
  <mat-card-header>
    <mat-card-title>Create Customer</mat-card-title>
  </mat-card-header>

  <mat-card-content>
    <!-- The form goes here -->
  </mat-card-content>
  <mat-card-actions>
    <!-- Actions go here -->
  </mat-card-actions>
</mat-card>
```

In the card content section, let's add an HTML form:

```
<form class="my-form">
  <!-- Form fields here -->
</form>
```

Next, let's organize the form fields into horizontal tabs using the Angular Material tabs components. Inside the form element, add the following code to create two tabs with *General Information* and *Other Information* labels:

```
<mat-tab-group>
  <mat-tab label="General Information" class="my-tab">
```

```

<!-- Form fields here -->
</mat-tab>
<mat-tab label="Other Information" class="my-tab">
<!-- Form fields here -->
</mat-tab>
</mat-tab-group>

```

In each tab, we'll add an HTML table to organize the form fields into table cells:

```

<table style="width: 100%" cellspacing="10">
<tr>
  <td> <!-- Form fields here --> </td>
</tr>
</table>

```

Add the following content for the first tab. Inside the table row/cell, add a field for the customer name:

```

<mat-form-field>
  <mat-label>Name</mat-label>
  <input matInput placeholder="Name" [(ngModel)]="name" name="name" required>
</mat-form-field>

```

Next create an input for the customer address:

```

<mat-form-field>
  <mat-label>Address</mat-label>
  <textarea [(ngModel)]="address" matInput></textarea>
</mat-form-field>

```

Finally, create an input field for the customer email:

```

<mat-form-field>
  <mat-label>Email</mat-label>
  <input matInput placeholder="Email" [(ngModel)]="email" name="email">
</mat-form-field>

```

We're using `<mat-label>` to create labels, `<input matInput />` to create an input field, and `<textarea matInput>` to create a text area field.

For the second tab, add the following content. In the first row/cell of the table, add the *Is company?* checkbox:

```
<mat-checkbox [checked]="isCompany">Is company?</mat-checkbox>
```

The `checked` property determines if the checkbox is checked or not.

Next, add a date input with a date picker wrapped inside an Angular Material form field:

```
<mat-form-field>
  <mat-label>Created At</mat-label>
  <input [value]="createdAt.value" matInput [matDatepicker]="picker"
    placeholder="Date of creation">
  <mat-datepicker-toggle matSuffix [for]="picker"></mat-datepicker-toggle>
  <mat-datepicker #picker></mat-datepicker>
</mat-form-field>
```

As explained in the [Angular mMaterial docs](#):

The datepicker allows users to enter a date either through text input, or by choosing a date from the calendar. It is made up of several components and directives that work together.

Next, add a group of radio buttons to choose the gender of the customer:

```
<mat-label>Gender</mat-label>
<mat-radio-group [(value)]="selectedGender">
  <mat-radio-button value="male">Male</mat-radio-button>
```

```
<mat-radio-button value="female">Female</mat-radio-button>
</mat-radio-group>
```

To get the currently selected radio button inside the group, simply bind a variable to the `value` property.

Finally, add a select field to select the source of the lead/customer:

```
<mat-form-field>
  <mat-label>Lead Source</mat-label>
  <mat-select [(value)]="selectedSource" placeholder="Source">
    <mat-option>None</mat-option>
    <mat-option value="email">Email Marketing</mat-option>
    <mat-option value="social">Social Media</mat-option>
    <mat-option value="affiliate">Affiliate</mat-option>
  </mat-select>
</mat-form-field>
```

The `<mat-select>` component supports two-way binding to the `value` property without the need for Angular forms.

Now let's add a button to create the customer from the form information:

```
<mat-card-actions>
  <button mat-raised-button (click)="createCustomer()" color="primary">Create
</button>
</mat-card-actions>
```

Now you need to create variables that are bound to these controls. In `src/app/customer-create/customer-create.component.ts`, add the following code to the component class:

```
name: string = "";
email: string = "";
job: string = "";
```

```

address: string = "";
selectedSource: string = "email";
selectedGender: string = "male";
isCompany : boolean = false;
createdAt = new FormControl(new Date());
public createCustomer(){
/* logic to create a customer from the form information*/
}

```

This is a screenshot of the component UI:

The screenshot shows a web form titled "Create Customer". It features two tabs: "General Information" and "Other Information". The "General Information" tab is currently selected. Below the tabs, there are four input fields: "Name *" (with an asterisk indicating it's required), "Address", "Job *" (also required), and "Email". At the bottom left of the form is a blue button labeled "Create".

11-5. The component UI

You can see a live demo of this [here](#).

Conclusion

With internationalization, accessibility, freedom from bugs and high performance, Angular Material aims to build a set of high-quality UI components using Angular and based on the Material design specification.

In this tutorial, we've built a simple UI using Angular Material with a variety of UI components. You can find the complete set of available Angular Material components from the [Angular Material docs](#).



Example code

You can find the source code of the demo we've built throughout this tutorial in [this GitHub repository](#).

Developing Angular Apps without a Back End Using MockBackend

Vildan Softic

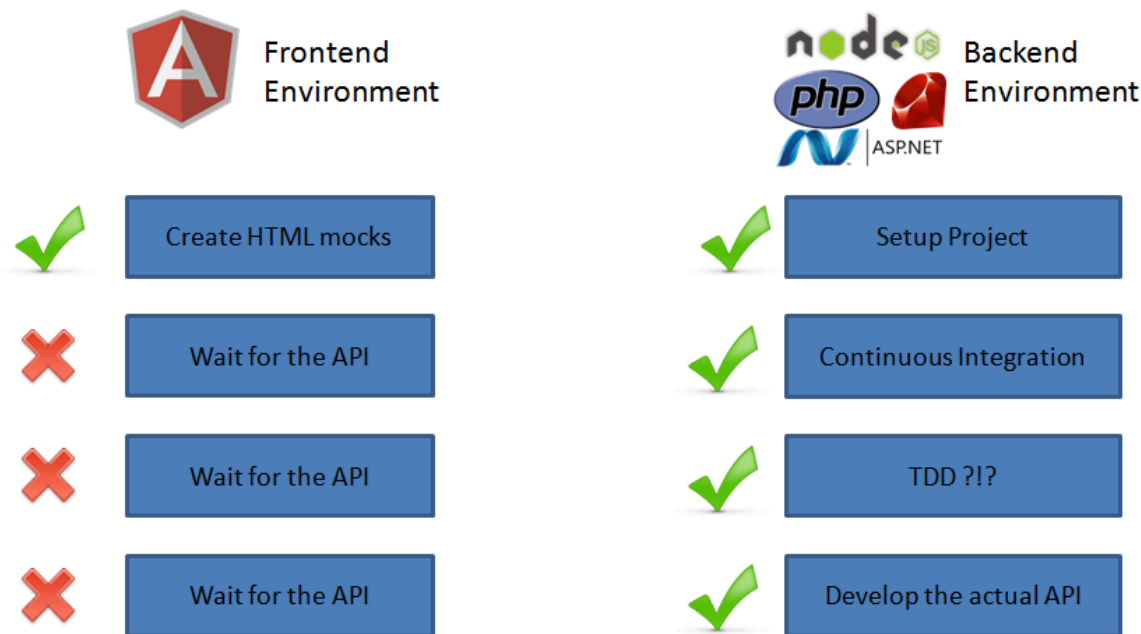
Chapter

12

In this article, we show how to develop apps with the Angular MockBackend class, providing a way for front-end teams to become independent of the back end, and a useful interface that reduces the risk of structural changes.

Getting your front-end and back-end teams up to full speed is certainly something each company is looking for. Often, though, teams fall into the pit of blocking dependencies. Those are situations where the upcoming work of one team is blocked by a user story owned by the other team.

One of those examples is the communication process between the front- and back-end. Over recent times, REST APIs have ascended the throne of so-called communication standards. The benefit of using JSON, a simple yet effective data transfer format, is that front-end workers don't need to care about the actual back end anymore. Whatever crosses the wire is directly consumable and may be leveraged to bring data into your application. So it's not surprising that those elementary entities often don't get modeled at all on the front end and are consumed as they arrive. This brings us to the fundamental problem of having to wait for the back-end team to provide something useful. As depicted in the following figure, we see that both teams start in parallel, but at a certain time one team is kept waiting for the other to catch up.



12-1. Diagram showing how front-end tasks depend on the back-end team finishing the API

Besides this, having no kind of fixed structure makes each change a potentially dangerous one. So the focus of this article is to present a way that front-end teams can become independent of the back end and at the same time provide a useful interface which reduces the risk of structural changes.

A Ticketing System without a Real Back End

In order to achieve that independence, it's imperative to start thinking upfront about your project. What entities are you going to use? What communication endpoints result therefore?

This can be done by creating a small table highlighting the necessary REST endpoints and describing their purpose. Remember the reason we're doing that upfront is for both parties to agree upon a common structure for communication. That doesn't mean it has to be perfectly done, but it should help you get started with the most important steps. As time passes, just update your

interface accordingly with the new routes needed.

The actual process of creating a back-endless environment is to capture all HTTP requests and instead of letting them go out into the wild, and reply with a fake response containing the information we'd like to have. This article will demonstrate the approach by describing a simple ticketing system. It uses the endpoints shown in the following table.



Using POST

Note that the example utilizes the `POST` verb for both the update and create route. Another option would be to leverage `PUT` for the update process. Keep in mind, though, that PUT should be idempotent, meaning every consecutive call has to produce the same result. Feel free to choose whatever suites your needs.

| Method | Route | Request body | Description |
|--------|-------------|---------------|--|
| GET | /ticket | None | Request all tickets |
| GET | /ticket/:id | None | Request a single ticket via the provided :id parameter |
| POST | /ticket | Ticket entity | Create a new or update an existing ticket |
| DELETE | /ticket/:id | None | Delete a ticket, identified by the :id parameter |

Table 1: Consumed endpoints of the ticketing system

The Ticket entity is a simple TypeScript class containing some basic ticket information:

```
export class Ticket {  
  public _id: string;  
  public title: string;  
  public assignedTo: string;  
  public description: string;  
  public percentageComplete: number;  
  
  constructor(id: string, title: string, assignedTo: string,  
    description: string, percentageComplete: number) {  
    this._id = id;  
    this.title = title;  
    this.assignedTo = assignedTo;  
    this.description = description;  
    this.percentageComplete = percentageComplete;  
  }  
}
```

`ticket.entity.ts` describing the ticket entity



Example Code

You may find the complete code as well as a preview for this example on [Plunker](#).

The Angular Project Setup

Enough theory, let's get our hands dirty with some coding. The project structure shown here is built upon the one proposed in the [Angular Getting Started guide](#). As such, we won't waste too much time explaining every part of it. For this article, you can just open up the above-mentioned Plunker to follow the code parts explained below.

As most single page applications start with an `index.html` file, let's take a look at that first. The first section imports the necessary polyfills. Followed by that we can see another reference to `system.config.js` which, amongst other things, configures third-party dependencies and Angular's application files. The

Reactive Extensions (Rx) aren't actually a true dependency but simplify the work with Angular's observables, which are the replacement for the previously used Promises. I highly recommend [this article by Cory Rylan](#) to learn more about this topic.

Note that manual script referencing is not the recommended way to create production-ready apps. You should use a package manager like [npm](#) or [jspm](#). The later one works hand in hand with SystemJS, described in section two. SystemJS is a module loader previously based on the ECMAScript 2015 draft and now part of [WHATWG's Loader specification](#). As such, it enables the use of the `import x from 'module'` syntax. In order to use it properly we need to configure it inside the previously mentioned file `system.config.js` and then import the application's main entry point `app`, which points to the file `app/boot.ts`.

This article won't deep dive into details of the `system.config.js` as those are just an example, based on the Angular Quickstart example.

Finally, we create the app by using a custom tag named `my-app`. Those are called Components and are somewhat comparable to AngularJS 1.x directives.

```
<!DOCTYPE html>
<html>

  <head>
    <title>ng2 Ticketing System</title>

    <!-- 1. Load libraries -->
    <!-- Polyfill(s) for older browsers -->
    <script src="https://unpkg.com/core-js/client/shim.min.js"></script>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.10.1/lodash
    ↳.min.js"></script>

    <script src="https://unpkg.com/zone.js@0.6.25?main=browser"></script>
    <script src="https://unpkg.com/reflect-metadata@0.1.8"></script>
```

```

<script src="https://unpkg.com/systemjs@0.19.39/dist/system.src.js"></script>

<!-- 2. Configure SystemJS -->
<script src="system.config.js"></script>
<script>
  System.import('app')
    .then(null, console.error.bind(console));
</script>

<meta charset="utf-8"/>
<link href="vendor/bootstrap/css/bootstrap.min.css" rel="stylesheet"/>
<link rel="stylesheet" href="styles.css"/>

</head>

<!-- 3. Display the application -->
<body>
  <my -app>Loading ...</my>
</body>
</html>

```

The file `boot.ts` is used to bootstrap Angular into the `my-app` component. Together with all application-specific code, it's located inside the folder `app`. Inside `boot.ts` we're going to perform the first steps necessary in order to leverage a mocked back end, which will act as a substitute for the real back end.

We start by creating a root module, to house our application. Its `provider` section is used to tell Angular's DI (dependency injection) system which actual instance of a class we'd like to use and what dependencies it requires. `BaseRequestOptions` provides general http helpers and `MockBackend` registers an instance of a mock implementation, which we're going to use to create our fake replies. If we look at the third provider configuration, creating a custom instance of the `Http` service, we can see that the requested dependencies (`deps`) are passed on to the `useFactory` method. Those are then used to create a new instance of `Http`.

The `imports` property is then used to declare additional module dependencies,

followed by the `declarations`, registering all available components of the root module. This module-wide registration enables each component to know what's available, without having to explicitly state directive requests as in previous versions of Angular 2. The last property, `bootstrap`, is used to state which component should be the entry point.

Finally, the method `bootstrapModule` is used to kickstart the app.

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { MockBackend } from '@angular/http/testing';
import { Http, BaseRequestOptions } from '@angular/http';
import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';
import { TicketComponent } from './ticket.component';

@NgModule({
  providers: [
    BaseRequestOptions,
    MockBackend,
    {
      provide: Http,
      deps: [MockBackend, BaseRequestOptions],
      useFactory: (backend, options) => { return new Http(backend, options); }
    }
  ],
  imports: [BrowserModule, FormsModule],
  declarations: [ AppComponent, TicketComponent ],
  bootstrap: [AppComponent]
})
export class AppModule { }

const platform = platformBrowserDynamic();
platform.bootstrapModule(AppModule);
```

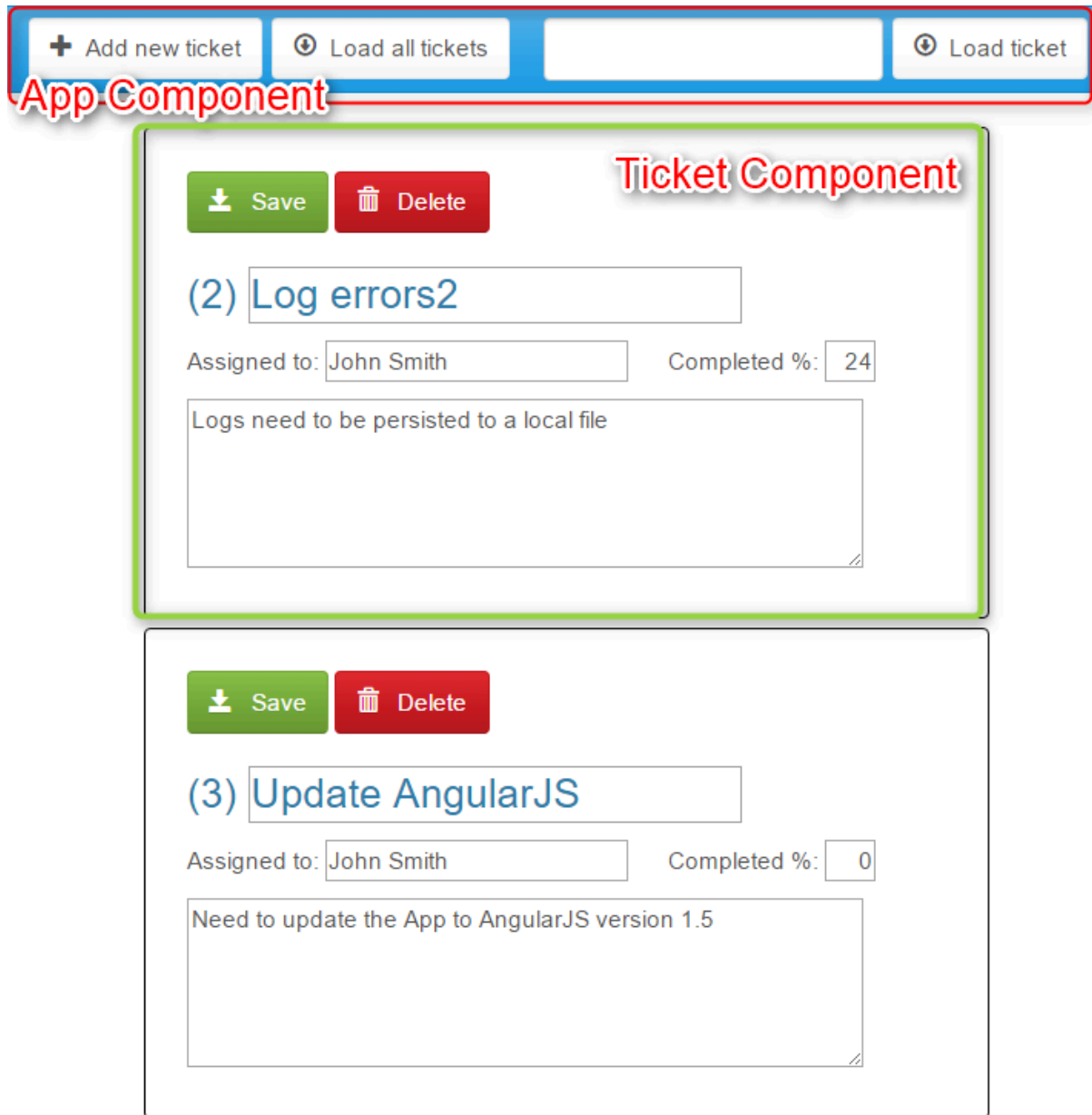


Ahem, Excuse Me ...

The class `MockBackend` is originally meant to be used in unit testing scenarios, in order to mock real server calls and therefore keep unit test runs quick and isolated. You can read more about this in the [official Http Documentation](#).

Working with Components

It's now time to take a look at the finished application to identify the components we're going to work with. As with every Angular 2 application, there's a so-called `AppComponent`, which acts as the main entry point into the application. It also can be used as a container, showing the general navigation and hosting sub-components. Speaking of these, we can see the `TicketComponent` being used repeatedly to display multiple ticket entities.



12-2. Screenshot of the ticket system, highlighting the separate components

The app component is configured to be used with the selector `my-app`, loading the template `index.html` located in the `templates` subfolder. Finally, `providers` tells Angular's DI that we'd like to obtain an instance of the `TicketService`.

```
@Component({
  selector: 'my-app',
  templateUrl: 'app/templates/index.html',
  providers: [TicketService]
})
export class AppComponent {
```

Next we define a `db` class property, which will hold a set of fake Tickets.

```
// Fake Tickets DB
private db: Ticket[] = [
  new Ticket(
    '1', 'Missing Exception', 'John Smith',
    'Method XYZ should throw exception in case ABC', 0),
  new Ticket(
    '2', 'Log errors', 'John Smith',
    'Logs need to be persisted to a local file', 24),
  new Ticket(
    '3', 'Update AngularJS', 'John Smith',
    'Need to update the App to AngularJS version 1.5', 0),
  new Ticket(
    '4', 'Border is missing', 'Jane Doe',
    'The element div.demo has no border defined', 100),
  new Ticket(
    '5', 'Introduce responsive grid', 'Jane Doe',
    'Implement responsive grid for better displays on mobile devices', 17)
];
```

The constructor now receives the injected `TicketService` as well as the fake back end. In here, we now subscribe to the `connections` stream. For each outgoing request we're now going to check its `request.method` and `request.url` in order to find out what type of endpoint is requested. If the proper route is matched, we reply using the `mockRespond` method, with a new `Response` containing the expected result as body which is initialized with the class `ResponseOptions`.

```
constructor(private service: TicketService, private backend: MockBackend) {
```

```

this.backend.connections.subscribe( c => {

  let singleTicketMatcher = /\api\ticket\/([0-9]+)/i;

  // return all tickets
  // GET: /ticket
  if (c.request.url === "http://localhost:8080/api/ticket" && c.request.method ===
    'GET') {
    let res = new Response( new ResponseOptions({
      body: JSON.stringify(this.db)
    }));

    c.mockRespond(res);
  }
}

```

When requesting a single ticket, we use the `singleTicketMatcher` defined above in order to perform a regex search on the `request.url`. After that, we search for the given ID and reply with the corresponding ticket entity.

```

// return ticket matching the given id
// GET: /ticket/:id
else if (c.request.url.match(singleTicketMatcher) && c.request.method === 'GET') {
  let matches = this.db.filter( (t) => {
    return t._id === c.request.url.match(singleTicketMatcher)[1]
  });

  c.mockRespond(new Response( new ResponseOptions({
    body: JSON.stringify(matches[0])
  })));
}

```

In case of updates and the creation of new tickets, we get the ticket entity delivered via the request body instead of a query parameter or URL pattern. Besides that, the work is pretty simple. We first check whether the ticket already exists and update it, otherwise we create a new one and send it back with the response. We do this in order to inform the requester about the new Ticket ID.

```

// Add or update a ticket
// POST: /ticket
else if (c.request.url === 'http://localhost:8080/api/ticket' &&
↳c.request.method === 1) {
    let newTicket: Ticket = JSON.parse(c.request._body);

    let existingTicket = this.db.filter( (ticket: Ticket) => { return ticket._id ==
↳ newTicket._id});
    if (existingTicket && existingTicket.length === 1) {
        Object.assign(existingTicket[0], newTicket);

        c.mockRespond(new Response( new ResponseOptions({
            body: JSON.stringify(existingTicket[0])
        })));
    } else {
        newTicket._id = parseInt(_.max(this.db, function(t) {
            return t._id;
        })._id || 0, 10) + 1 + '';

        this.db.push(newTicket);

        c.mockRespond(new Response( new ResponseOptions({
            body: JSON.stringify(newTicket)
        })));
    }
}

// Delete a ticket
// DELETE: /ticket/:id
else if (c.request.url.match(singleTicketMatcher) && c.request.method === 3) {
    let ticketId = c.request.url.match(singleTicketMatcher)[1];
    let pos = _.indexOf(_.pluck(this.db, '_id'), ticketId);

    this.db.splice(pos, 1);

    c.mockRespond(new Response( new ResponseOptions({
        body: JSON.stringify({})
    })));
}

});
}

```

Last but not least, the page life cycle hook `ngOnInit` will trigger the loading of all tickets when the component is fully rendered.

```
public ngOnInit() {
  this.service.loadAllTickets();
}
}
```

In a real production app, you'd separate the mock setup into a separate service and inject it as a dependency into the AppComponent. Or even better, you'd create a whole new module housing your fake server and add it to your app's root module. This is omitted here in order to keep the demo simpler.

Looking at the `TicketComponent` we can see that nothing too interesting happens, besides the Component decorator. We define `ticket` as the selector and again point to a separate template file. Now, in contrast to the `AppComponent`, we expect a ticket tag to be created with an attribute named `title` as well and getting the to be rendered entity.

The constructor then finally gets the `TicketService` injected and assigns it to a class property `service`.

```
import {
  Component,
  Input
} from '@angular/core';

import {Ticket} from './ticket.entity';
import {TicketService} from './ticket.service';

@Component({
  moduleId: module.id,
  selector: 'ticket',
  templateUrl: 'templates/ticket.html',
  //providers: [TicketService] < -- this would override the parent DI instance
})
```

```
export class TicketComponent {
  @Input('ticket') ticket: Ticket;

  constructor(private service: TicketService) { }
}
```

The Ticket Service

The last thing missing is the `TicketService`, used to abstract the Ajax calls away from the components. As we can see, it expects the `http` service to be injected. Now, remembering the initial `boot.ts` file, we know that the instance provided will be the one with the mocked back end. The actual request stays the same by leveraging the `HTTP` services request methods like `post` or `get`, mapping the result — which in this case will be the fake reply — and proceeding with the custom application logic.

```
import {Ticket} from './ticket.entity';
import {Injectable} from '@angular/core';
import {Http, Headers} from '@angular/http';
import 'rxjs/add/operator/map';

@Injectable()
export class TicketService {
  tickets: Ticket[] = [];

  constructor(private http: Http) {

  }

  addNewTicket() {
    var headers = new Headers();
    headers.append('Content-Type', 'application/json');

    var newTicket = new Ticket("0", 'New Ticket', 'Nobody',
      ↪ 'Enter ticket description here', 0);
    this.http
```



```

    .post('http://localhost:8080/api/ticket', JSON.stringify(newTicket), headers)
    .map(res => res.json())
    .subscribe(
      data => this.tickets.push(data),
      err => this.logError(err),
      () => console.log('Updated Ticket')
    );
  }

  saveTicket(ticket: Ticket) {
    ...
  }

  deleteTicket(ticket: Ticket) {
    ...
  }

  loadAllTickets() {
    ...
  }

  loadTicketById(id) {
    ...
  }

  logError(err) {
    console.error('There was an error: ' + err);
  }
}

```

Conclusion

Summing up, we saw how Angular's dependency injection can help us to replace the default `XHRBackend` of the `HTTP` service with a mocked back end. Inside the `AppComponent`, we then created our fake database, intercepted every outgoing request, and replied with a custom fake response. The benefits we've gained are now complete independence from the back-end team and, at the same time, a defined interface. Now, once the production back-end is in place, all we need to

do is to remove the dependency injection override and faked back end, and we're good to go.

React vs Angular: An In- depth Comparison

Pavels Jelisejevs

Chapter

13

Should I choose Angular or React? Today's bipolar landscape of JavaScript frameworks has left many developers struggling to pick a side in this debate. Whether you're a newcomer trying to figure out where to start, a freelancer picking a framework for your next project, or an enterprise-grade architect planning a strategic vision for your company, you're likely to benefit from having an educated view on this topic.

To save you some time, let me tell you something up front: this article won't give a clear answer on which framework is better. But neither will hundreds of other articles with similar titles. I can't tell you that, because the answer depends on a wide range of factors which make a particular technology more or less suitable for your environment and use case.

Since we can't answer the question directly, we'll attempt something else. We'll compare Angular (2+, not the old AngularJS) and React, to demonstrate how you can approach the problem of comparing any two frameworks in a structured manner on your own and tailor it to your environment. You know, the old "teach a man to fish" approach. That way, when both are replaced by a BetterFramework.js in a year's time, you'll be able to re-create the same train of thought once more.

Where to Start?

Before you pick any tool, you need to answer two simple questions: "Is this a good tool per se?" and "Will it work well for my use case?" Neither of them mean anything on their own, so you always need to keep both of them in mind. All right, the questions might not be that simple, so we'll try to break them down into smaller ones.

Questions on the tool itself:

- How mature is it and who's behind it?
- What kind of features does it have?
- What architecture, development paradigms, and patterns does it employ?

- What is the ecosystem around it?

Questions for self-reflection:

- Will I and my colleagues be able to learn this tool with ease?
- Does it fit well with my project?
- What is the developer experience like?

Using this set of questions you can start your assessment of any tool and we'll base our comparison of React and Angular on them as well.

There's another thing we need to take into account. Strictly speaking, it's not exactly fair to compare Angular to React, since Angular is a full-blown, feature-rich framework, while React is just a UI component library. To even the odds, we'll talk about React in conjunction with some of the libraries often used with it.

Maturity

An important part of being a skilled developer is being able to keep the balance between established, time-proven approaches and evaluating new bleeding-edge tech. As a general rule, you should be careful when adopting tools that haven't yet matured due to certain risks:

- The tool may be buggy and unstable.
- It might be unexpectedly abandoned by the vendor.
- There might not be a large knowledge base or community available in case you need help.

Both React and Angular come from good families, so it seems that we can be confident in this regard.

React

React is developed and maintained by Facebook and used in their own products,

including Instagram and WhatsApp. It has been around for roughly three and a half years now, so it's not exactly new. It's also one of the most popular projects on GitHub, with about 74,000 stars at the time of writing. Sounds good to me.

Angular

Angular (version 2 and above) has been around less then React, but if you count in the history of its predecessor, AngularJS, the picture evens out. It's maintained by Google and used in AdWords and Google Fiber. Since AdWords is one of the key projects in Google, it is clear they have made a big bet on it and is unlikely to disappear anytime soon.

Features

Like I mentioned earlier, Angular has more features out of the box than React. This can be both a good and a bad thing, depending on how you look at it.

Both frameworks share some key features in common: components, data binding, and platform-agnostic rendering.

Angular

Angular provides a lot of the features required for a modern web application out of the box. Some of the standard features are:

- Dependency injection
- Templates, based on an extended version of HTML
- Routing, provided by @angular/router
- Ajax requests by @angular/http
- @angular/forms for building forms
- Component CSS encapsulation
- XSS protection
- Utilities for unit-testing components.

Having all of these features available out of the box is highly convenient when you don't want to spend time picking the libraries yourself. However, it also means that you're stuck with some of them, even if you don't need them. And replacing them will usually require additional effort. For instance, we believe that for small projects having a DI system creates more overhead than benefit, considering it can be effectively replaced by imports.

React

With React, you're starting off with a more minimalistic approach. If we're looking at just React, here's what we have:

- No dependency injection
- Instead of classic templates it has JSX, an XML-like language built on top of JavaScript
- XSS protection
- Utilities for unit-testing components.

Not much. And this can be a good thing. It means that you have the freedom to choose whatever additional libraries to add based on your needs. The bad thing is that you actually have to make those choices yourself. Some of the popular libraries that are often used together with React are:

- React-router for routing
- Fetch (or axios) for HTTP requests
- A wide variety of techniques for CSS encapsulation
- Enzyme for additional unit-testing utilities.

We've found the freedom of choosing your own libraries liberating. This gives us the ability to tailor our stack to particular requirements of each project, and we didn't find the cost of learning new libraries that high.

Languages, Paradigms, and Patterns

Taking a step back from the features of each framework, let's see what kind higher-level concepts are popular with both frameworks.

React

There are several important things that come to mind when thinking about React: JSX, Flow, and Redux.

JSX

JSX is a controversial topic for many developers: some enjoy it, and others think that it's a huge step back. Instead of following a classical approach of separating markup and logic, React decided to combine them within components using an XML-like language that allows you to write markup directly in your JavaScript code.

While the merits of mixing markup with JavaScript might be debatable, it has an indisputable benefit: static analysis. If you make an error in your JSX markup, the compiler will emit an error instead of continuing in silence. This helps by instantly catching typos and other silly errors.

Flow

Flow is a type-checking tool for JavaScript also developed by Facebook. It can parse code and check for common type errors such as implicit casting or null dereferencing.

Unlike TypeScript, which has a similar purpose, it does not require you to migrate to a new language and annotate your code for type checking to work. In Flow, type annotations are optional and can be used to provide additional hints to the analyzer. This makes Flow a good option if you would like to use static code analysis, but would like to avoid having to rewrite your existing code.

■ Further reading: [Writing Better JavaScript with Flow](#)

Redux

Redux is a library that helps manage state changes in a clear manner. It was inspired by Flux, but with some simplifications. The key idea of Redux is that the whole state of the application is represented by a single object, which is mutated by functions called reducers. Reducers themselves are pure functions and are implemented separately from the components. This enables better separation of concerns and testability.

If you're working on a simple project, then introducing Redux might be an over complication, but for medium- and large-scale projects, it's a solid choice. The library has become so popular that there are projects implementing it in Angular as well.

All three features can greatly improve your developer experience: JSX and Flow allow you to quickly spot places with potential errors, and Redux will help achieve a clear structure for your project.

Angular

Angular has a few interesting things up its sleeve as well, namely TypeScript and RxJS.

TypeScript

TypeScript is a new language built on top of JavaScript and developed by Microsoft. It's a superset of JavaScript ES2015 and includes features from newer versions of the language. You can use it instead of Babel to write state of the art JavaScript. It also features an extremely powerful typing system that can statically analyze your code by using a combination of annotations and type inference.

There's also a more subtle benefit. TypeScript has been heavily influenced by Java and .NET, so if your developers have a background in one of these languages, they are likely to find TypeScript easier to learn than plain JavaScript (notice how we switched from the tool to your personal environment). Although Angular has been the first major framework to actively adopt TypeScript, it's also possible to use it together with React.

■ **Further reading:** [An Introduction to TypeScript: Static Typing for the Web](#)

RxJS

RxJS is a reactive programming library that allows for more flexible handling of asynchronous operations and events. It's a combination of the Observer and Iterator patterns blended together with functional programming. RxJS allows you to treat anything as a continuous stream of values and perform various operations on it such as mapping, filtering, splitting or merging.

The library has been adopted by Angular in their HTTP module as well for some internal use. When you perform an HTTP request, it returns an Observable instead of the usual Promise. Although this library is extremely powerful, it's also quite complex. To master it, you'll need to know your way around different types of Observables, Subjects, as well as around a [hundred methods and operators](#). Yikes, that seems to be a bit excessive just to make HTTP requests!

RxJS is useful in cases when you work a lot with continuous data streams such as web sockets, however, it seems overly complex for anything else. Anyway, when working with Angular you'll need to learn it at least on a basic level.

■ **Further reading:** [Introduction to Functional Reactive Programming with RxJS](#)

We've found TypeScript to be a great tool for improving the maintainability of our projects, especially those with a large code base or complex domain/business logic. Code written in TypeScript is more descriptive and easier to follow. Since TypeScript has been adopted by Angular, we hope to see even more projects

using it. RxJS, on the other hand, seems only to be beneficial in certain cases and should be adopted with care. Otherwise, it can bring unwanted complexity to your project.

Ecosystem

The great thing about open source frameworks is the number of tools created around them. Sometimes, these tools are even more helpful than the framework itself. Let's have a look at some of the most popular tools and libraries associated with each framework.

Angular

Angular CLI

A popular trend with modern frameworks is having a CLI tool that helps you bootstrap your project without having to configure the build yourself. Angular has [Angular CLI](#) for that. It allows you to generate and run a project with just a couple of commands. All of the scripts responsible for building the application, starting a development server and running tests are hidden away from you in `node_modules`. You can also use it to generate new code during development. This makes setting up new projects a breeze.

■ **Further reading:** [The Ultimate Angular CLI Reference](#)

Ionic 2

[Ionic 2](#) is a new version of the popular framework for developing hybrid mobile applications. It provides a Cordova container that is nicely integrated with Angular 2, and a pretty material component library. Using it, you can easily set up and build a mobile application. If you prefer a hybrid app over a native one, this is a good choice.

Material design components

If you're a fan of material design, you'll be happy to hear that there's a Material component library available for Angular. Currently, it's still at an early stage and slightly raw but it has received lots of contributions recently, so we might hope for things to improve soon.

Angular universal

Angular universal is a seed project that can be used for creating projects with support for server-side rendering.

@ngrx/store

@ngrx/store is a state management library for Angular inspired by Redux, being based on state mutated by pure reducers. Its integration with RxJS allows you to utilize the push change detection strategy for better performance.

■ **Further reading:** Managing State in Angular 2 Apps with ngrx/store

There are plenty of other libraries and tools available in the Awesome Angular list.

React

Create React App

Create React App is a CLI utility for React to quickly set up new projects. Similar to Angular CLI it allows you to generate a new project, start a development server and create a bundle. It uses Jest, a relatively new test runner from Facebook, for unit testing, which has some nice features of its own. It also supports flexible application profiling using environment variables, backend

proxies for local development, Flow, and other features. Check out this [brief introduction to Create React App](#) for more information.

React Native

React Native is a platform developed by Facebook for creating native mobile applications using React. Unlike Ionic, which produces a hybrid application, React Native produces a truly native UI. It provides a set of standard React components which are bound to their native counterparts. It also allows you to create your own components and bind them to native code written in Objective-C, Java or Swift.

Material UI

There's a [material design component](#) library available for React as well. Compared to Angular's version, this one is more mature and has a wider range of components available.

Next.js

Next.js is a framework for the server-side rendering of React applications. It provides a flexible way to completely or partially render your application on the server, return the result to the client and continue in the browser. It tries to make the complex task of creating universal applications as simple as possible so the set up is designed to be as simple as possible with a minimal amount of new primitives and requirements for the structure of your project.

MobX

MobX is an alternative library for managing the state of an application. Instead of keeping the state in a single immutable store, like Redux does, it encourages you to store only the minimal required state and derive the rest from it. It provides a set of decorators to define observables and observers and introduce reactive logic to your state.

■ **Further reading:** [How to Manage Your JavaScript Application State with MobX](#)

Storybook

[Storybook](#) is a component development environment for React. It allows you to quickly set up a separate application to showcase your components. On top of that, it provides numerous add-ons to document, develop, test and design your components. We've found it to be extremely useful to be able to develop components independently from the rest of the application. You can [learn more about Storybook](#) from a previous article.

There are plenty of other libraries and tools available in [the Awesome React list](#).

Adoption, Learning Curve and Development

Experience

An important criterion for choosing a new technology is how easy it is to learn. Of course, the answer depends on a wide range of factors such as your previous experience and a general familiarity with the related concepts and patterns. However, we can still try to assess the number of new things you'll need to learn to get started with a given framework. Now, if we assume that you already know ES6+, build tools and all of that, let's see what else you'll need to understand.

React

With React, the first thing you'll encounter is JSX. It does seem awkward to write for some developers. However, it doesn't add that much complexity --- just expressions, which are actually JavaScript, and a special HTML-like syntax. You'll also need to learn how to write components, use props for configuration and manage internal state. You don't need to learn any new logical structures or

loops since all of this is plain JavaScript.

The [official tutorial](#) is an excellent place to start learning React. Once you're done with that, [get familiar with the router](#). The react router v4 might be slightly complex and unconventional, but nothing to worry about. Using Redux will require a paradigm shift to learn how to accomplish already familiar tasks in a manner suggested by the library. The free [Getting Started with Redux](#) video course can quickly introduce you to the core concepts. Depending on the size and the complexity of your project you'll need to find and learn some additional libraries and this might be the tricky part, but after that everything should be smooth sailing.

We were genuinely surprised at how easy it was to get started using React. Even people with a backend development background and limited experience in frontend development were able to catch up quickly. The error messages you might encounter along the way are usually clear and provide explanations on how to resolve the underlying problem. The hardest part may be finding the right libraries for all of the required capabilities, but structuring and developing an application is remarkably simple.

Angular

Learning Angular will introduce you to more new concepts than React. First of all, you'll need to get comfortable with TypeScript. For developers with experience in statically typed languages such as Java or .NET this might be easier to understand than JavaScript, but for pure JavaScript developers, this might require some effort.

The framework itself is rich in topics to learn, starting from basic ones such as modules, dependency injection, decorators, components, services, pipes, templates, and directives, to more advanced topics such as change detection, zones, AoT compilation, and Rx.js. These are all covered in the [documentation](#). Rx.js is a heavy topic on its own and is described in much detail on the [official website](#). While relatively easy to use on a basic level it gets more complicated

when moving on to advanced topics.

All in all, we noticed that the entry barrier for Angular is higher than for React. The sheer number of new concepts is confusing to newcomers. And even after you've started, the experience might be a bit rough since you need to keep in mind things like Rx.js subscription management, change detection performance and bananas in a box (yes, this is an actual advice from the documentation). We often encountered error messages that are too cryptic to understand, so we had to google them and pray for an exact match.

It might seem that we favor React here, and we definitely do. We've had experience onboarding new developers to both Angular and React projects of comparable size and complexity and somehow with React it always went smoother. But, like I said earlier, this depends on a broad range of factors and might work differently for you.

Putting it Into Context

You might have already noted that each framework has its own set of capabilities, both with their good and bad sides. But this analysis has been done outside of any particular context and thus doesn't provide an answer on which framework should you choose. To decide on that, you'll need to review it from a perspective of your project. This is something you'll need to do on your own.

To get started, try answering these questions about your project and when you do, match the answers against what you've learned about the two frameworks. This list might not be complete, but should be enough to get you started:

- 1 How big is the project?
- 2 How long is it going to be maintained for?
- 3 Is all of the functionality clearly defined in advance or are you expected to be flexible?

- 4 If all of the features are already defined, what capabilities do you need?
- 5 Are the domain model and business logic complex?
- 6 What platforms are you targeting? Web, mobile, desktop?
- 7 Do you need server-side rendering? Is SEO important?
- 8 Will you be handling a lot of real-time event streams?
- 9 How big is your team?
- 10 How experienced are your developers and what is their background?
- 11 Are there any ready-made component libraries that you would like to use?

If you're starting a big project and you would like to minimize the risk of making a bad choice, consider creating a proof-of-concept product first. Pick some of the key features of the projects and try to implement them in a simplistic manner using one of the frameworks. PoCs usually don't take a lot of time to build, but they'll give you some valuable personal experience on working with the framework and allow you to validate the key technical requirements. If you're satisfied with the results, you can continue with full-blown development. If not, failing fast will save you a lot of headaches in the long run.

One Framework to Rule Them All?

Once you've picked a framework for one project, you'll get tempted to use the exact same tech stack for your upcoming projects. Don't. Even though it's a good idea to keep your tech stack consistent, don't blindly use the same approach every time. Before starting each project, take a moment to answer the same questions once more. Maybe for the next project, the answers will be different or the landscape will change. Also, if you have the luxury of doing a small project with a non-familiar tech stack, go for it. Such experiments will provide you with invaluable experience. Keep your mind open and learn from your mistakes. At some point, a certain technology will just feel natural and *right*.