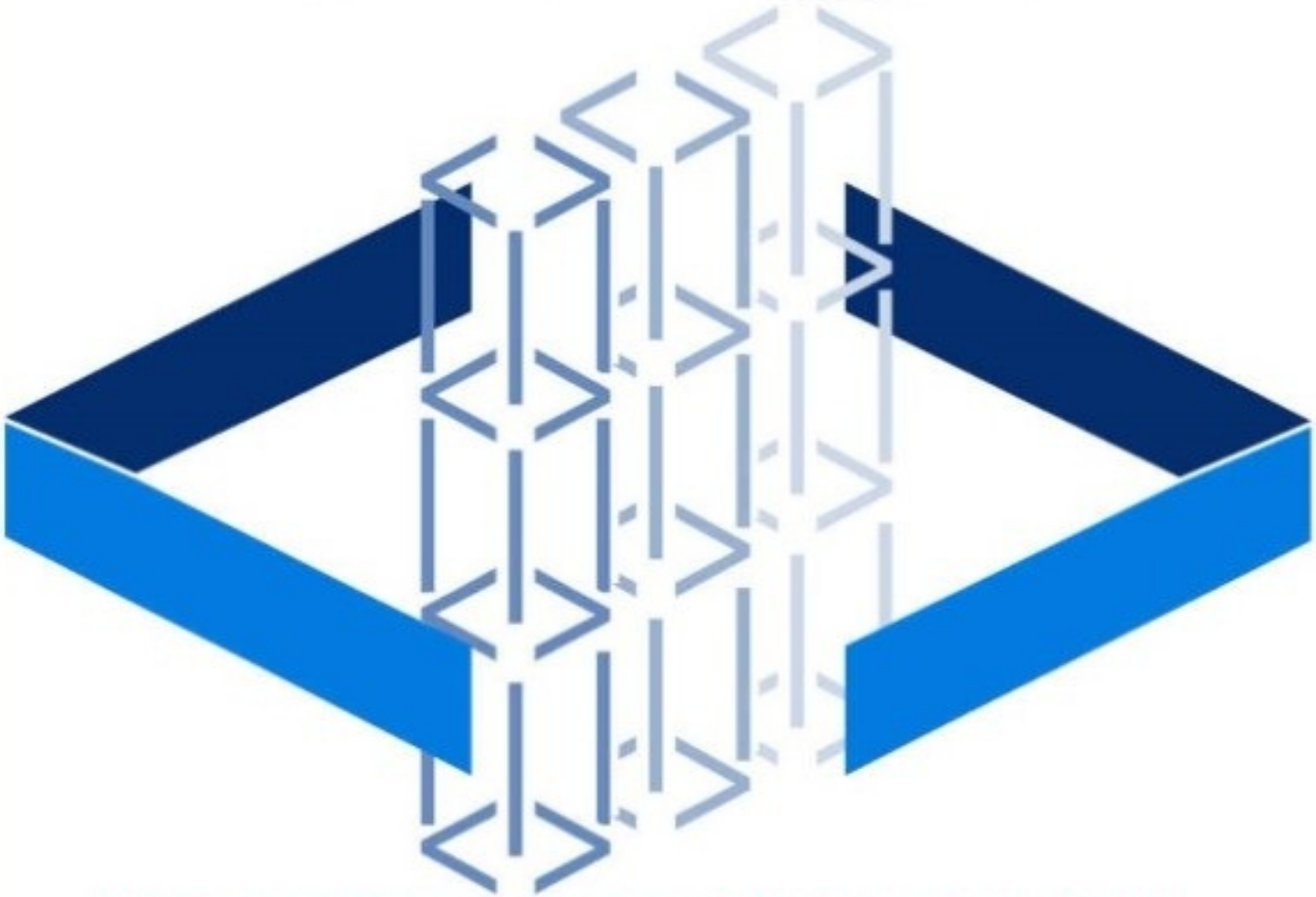
Bleeding Edge Press

Developing an Angular 2 Edge



Filip Lauc, Suguru Inatomi, Wojciech Kwiatek,
Mary Gualtieri, Ran Wahle & Michael Haberman

Developing an Angular 2 Edge

By Filip Lauc, Suguru Inatomi, Wojciech Kwiatek, Mary Gualtieri, Ran Wahle, and Michael Haberman

Developing an Angular 2 Edge

ISBN 9781939902375
Published by: Bleeding Edge Press, Santa Rosa, CA 95404
Title: Developing an Angular 2 Edge
Authors: By Filip Lauc, Suguru Inatomi, Wojciech Kwiatek, Mary Gualtieri, Ran Wahle, and Michael Haberman
Acquisitions Editor: Christina Rudloff
Editor: Troy Mott
Cover Design: Martin Murtonen
Website: bleedingedgepress.com

Preface

What is Angular 2?

Angular 2 is a JavaScript framework intended for development across a variety of platforms including web, mobile, and desktop applications. The framework's development is led by some of the industry's biggest names, including Miško Hevery and Igor Minar. Although developed and maintained primarily by Google, it also boasts an incredible community following and support.

At the moment of writing this book, Angular 2 is in the state of release candidate 1, with the first production release on the way.

Should I use Angular 2?

Angular 2 uses the latest application development technologies and patterns to maximize speed and performance of your applications. With the production release on the way, there has never been a better time to start using the framework. If you have an interest in web application development or even native mobile and desktop development, Angular 2 can be a great framework for you.

Even though it may not be as “beginner friendly” as some JavaScript frameworks, thanks to the great team of developers behind Angular 2, you can rest assured in the quality of the coding practices you learn while using Angular 2.

What do I need to know prior to reading?

This book introduces the Angular 2 framework. It's intended for developers who have prior experience with web development. As such, you will need to have some experience with the JavaScript programming language, as well as HTML and CSS, to take full advantage of the materials provided in this book.

What will you learn from this book?

This book is focused on demonstrating the key features of the Angular 2 framework. Although there are many great resources out there already, this book serves as a great starting point for newcomers as well as an examination of some more advanced features.

By the end of the book you will have the knowledge required for developing a robust web application. You will also have an insight into application testing, a task too often overlooked in the development process.

Code samples

We will write numerous code samples throughout the book. We will also refer to a chat and scheduling application developed for the purposes of the book. The entire application is openly available at https://github.com/flauc/angular2-edge-app.

The writing process

This book was written as a focused virtual book sprint over the course of a month or two. This process helps create fresh and current content, whereas conventional books often lag behind the coverage of cutting edge trends and technology.

About the authors

Filip Lauc is a Full Stack JavaScript developer currently working at Gauss Development in Osijek, Croatia. Among other things he is passionate about writing code for the open source community and working with new technologies. He is active on GitHub.

Wojciech Kwiatek is a JavaScript developer, trainer and mentor. Connected with DevMeetings organization. Loves cutting edge technologies, working with both Angular 2 and React right now. Recently excited about reactive and functional programming style.

Suguru Inatomi is an engineer at Topgate in Tokyo, Japan. He is an Angular Contributor and an Animeholic. You can reach him on Twitter.

Mary Gualtieri is an Associate E-Commerce Web Developer of Ritter's Communications, a direct print marketing company in South Florida. As the sole developer on staff, she oversees the development of all e-commerce projects. She is passionate about the tech community and enabling women to achieve their goals through technology.

Ran Wahle is a Full Stack JavaScript developer at Tikal Knowledge , a professional services company in Israel. He has many years of web development, starting with ASP.NET, through many client libraries, Angular (1.x and 2.0) and nodeJs. Ran also tends to speak in various web sessions, an active on GitHub.

Michael Haberman is a consultant for small startups and a major company, and loves the challenges he faces in his daily work. He focuses on Angular, React, and Ionic on the client side and Node, Web API, and ASP.NET MVC on the server side. You can reach him on Twitter.

Chapter 1. Angular 2 Introduction

Angular 2 was highly anticipated by front-end developers. Two years elapsed from the initial announcement by Google, before the beta was released. For a front-end developer, that seems like ages. Indeed, Angular 2 turned out to be a brand new framework, however, it is sharing some of the concepts with its predecessor. Let’s take a brief look at AngularJS and JavaScript, before focusing on what Angular 2 offers.

Going back in time to the early stages of web applications, even if an app contained a lot of dynamic content, the general approach was to generate static HTML files on the server and send them back to the user. Every user action ended with a new, regenerated page. It worked like a charm until people wanted more dynamics added to the webpages. Sending a whole new HTML file was not so bad when most of the content differed, it was overkill when doing something like showing user text for the wrong input in a form.

JavaScript was first introduced to add some dynamic content to static files and mainly to allow modifying forms on the fly. It was a radical change. In the meantime, we have seen a lot of people working with the still most widely used library: jQuery. It made DOM modifications a breeze in comparison to the native API. The community pushed it more and more and started to use jQuery not only to show/hide elements, but also to create rich applications that had only one HTML file server from the server, and then everything was fetched dynamically using AJAX. The server role began a transformation from being a main source of application, to being a layer between the application and the database.

AngularJS

Looking back, there was a possibility to create very advanced web applications using the pure JavaScript API, but it was really difficult to maintain the initial codebase, and to test all of it. And then in 2010 AngularJS was introduced as a JavaScript MVW Framework. It has tree big advantages that made people love it:

- Code production was incomparably fast
- Every piece of the application was easily testable
- Google was behind the project

There are other things that make Angular so good for programmers. The first is two-way data binding. Angular allowed you to see data changed in JavaScript to be reflected automatically on the UI. It made things a lot easier to develop at the beginning, because no more coding was required except for linking a proper controller to the part of HTML. The second benefit that Angular brings to programmers are directives. They are the starting point for all of the components we see now in the modern front-end. Directives allowed the code to be much more reusable and separated than ever before. AngularJS forced a Dependency Injection, and it helped in mocking dependencies. Its appearance in the framework made Angular a huge step forward in the case of testing front-end applications.

All of these advantages led more and more companies to rewrite their apps from their own solutions built on top of other libraries to AngularJS.

Angular 2

AngularJS is a great way to kickstart an app or MVP. With a rising popularity and more and more features coming to the core, the Angular team decided to rewrite the original framework, introducing Angular 2. Some say Angular 2 and AngularJS share only one thing: the name. There is a migration path (called `ng-upgrade`) from AngularJS to Angular 2. However, Angular 2 is still a brand new framework sharing only some concepts of its predecessor.

The whole concept of application structure has changed in Angular 2. Previously it was the MVC framework that allowed you to create applications in the pattern of rather tightly coupled entities like controllers, views, services, etc.

Now, the concept of directives has been pushed further to be much closer to the Web Components' standard and React's way of structuring the application. It is all about components in Angular 2. It means the whole application is now a component, which contains another set of components (which can be routable). It ends up with a tree-like structure.

The purpose of the Angular 2 application architecture is to create components that don't depend on each other, which are as loosely coupled as possible.

The important thing is to introduce two ways of creating the components:

- Smart components: They know about application state, and they can communicate with services to fetch or modify data.
- Dumb components: They should only have inputs and outputs. They are ready to be placed anywhere in the system (or even outside it) when providing proper values to the input, and they should not know about application state existence.

Performance

Having such a tree of components makes a big difference in performance. The purpose with AngularJS wasn't to create the most efficient framework, but instead the easiest one to write in. As performance became more of a problem, Angular 2 was introduced to solve the issue. AngularJS had a *digest cycle*, which allowed changes to trigger updates up and down. Angular 2, on the other hand, has a directional graph of components that is always being checked once (due to one traversal path from the root to the leaves). According to the Angular core team members, these changes made Angular 2 applications work 3-10x faster than the same apps created with the latest AngularJS.

Ecosystem

Angular is now more than ever considered a framework. The tool that the Angular team gave us is a complete solution. This is the opposite of React (which is just a library for rendering a component), but here we are able to create whole applications without using any third-party solutions. The framework provides us with the following blocks to use:

- Component - The main and the most important one. Angular 2 is about components and modularization. These are the main blocks to create apps with. It contains logic and the view.
- Directive - It is very similar to the component, but does not contain a template (view). It just makes additional operations to the existing DOM elements.
- Pipe - Gives you the ability to process variables in templates in a specific manner.
- Service - Injectable class used to share logic between components. Usually (and preferably) it is a layer between the component and the API.
- Forms - A complex solution for handling forms in the app. This time you can choose between two styles: template driven and model driven.
- HTTP - A service built on top of the observables to handle the connection between the front end and the API.
- Router - Now routing directly to the components.
- Testing - A whole bunch of tools for both unit and end-to-end testing, ready to handle scenarios with all of the blocks above.

Modern JavaScript is going further beyond just the web browsers. Now you hear about server-side rendering and native mobile apps created with JavaScript. In Angular 2 there are plenty of other projects connected to the core, which are worth a mention and deserve their own story:

- Angular Universal - server-side rendering.
- Angular CLI - command line interface for kickstarting a project.
- NativeScript 2 - native mobile apps.
- Ionic 2 - mobile apps using web-view.
- Codelyzer - linting specific for Angular 2.
- ngrx - reactive programming.

It all makes Angular not only a framework, but a whole platform.

Is Angular 2 the right choice for my project?

As you can probably see, there is much more to learn in Angular 2 than in the previous version. It also makes projects more complex, so the first steps are harder. It is the opposite feeling than we had with AngularJS. It is generally not easy (or even impossible) to jump into the project without knowing about its concepts and toolbox. Although this can change in the future, we recommend only starting Angular 2 projects if you are an experienced JavaScript developer.

We mentioned how Angular 2 is a first citizen framework. This is incredibly powerful, but has its own drawbacks. The problem is mainly *zone.js*, which is interfering with most of the asynchronous operations that are taking place in the browser. It can break your existing code if you're trying to merge it with another framework or library. Due to this inconvenience, you should be very careful when trying to upgrade your existing non-Angular app step-by-step (but it is possible). On the other hand, it's highly recommended you upgrade your AngularJS app (*ng-upgrade* will be helpful), and we cover this topic later in the book.

Now you should be convinced that there are two easy paths to start with Angular 2:

- upgrading an existing AngularJS 1.5 app (based on components)
- starting an app from the scratch

Another good question is: it is good for my team? As you can probably see, we're going deeper and deeper trying to introduce even more tools and concepts. They make a small barrier at the beginning of a project, but they are beneficial in the long run. It also makes Angular 2 a solution to use with a big team. Most of the conventions that are there help with the maintenance of a complex application. It also makes Angular 2 a better tool for long-term projects.

Chapter 2. Choosing your scripting language

One of the most difficult things for developers when creating new applications is choosing the most effective way to write JavaScript. There are many different scripting languages to choose from, such as TypeScript, EMCAScript, JavaScript, or Dart, just to name a few.

TypeScript and EMCAScript, however, seem to be the standout scripting languages for Angular 2 applications. It is important to choose the scripting language that is going to make you the most powerful, and to fully accomplish your application’s purpose.

What is TypeScript?

TypeScript is an open source programming language. The specific purpose of TypeScript is to aid in the development of JavaScript applications for the client-side or server-side (such as Node.js) execution. It is designed for the development of large applications. Since TypeScript is a superset of JavaScript, any JavaScript programs are valid TypeScript programs.

TypeScript supports definition files that contain type information of existing JavaScript libraries. This, in turn, allows other programs to use the values defined in in the files as if they were static TypeScript forces.

What is great about TypeScript is that it starts and ends with JavaScript. TypeScript uses the same semantics and syntax of JavaScript. It makes it easier for JavaScript developers to use existing JavaScript code and incorporate JavaScript Libraries. TypeScript compiles to clean and simple JavaScript code that is able to run in any browser that supports ECMAScript 3 or newer.

Developers use types to enable the use of highly productive development tools. This allows for a practice of static checking and refactoring code when developing JavaScript applications. Of course, types are optional. However, type reasoning allows for a few type annotations that can make a significant difference to the static verification of your code. Types actually let you define these reasons between components. This help you gain insights into the behavior of existing JavaScript libraries. Essentially, TypeScript is a strong tool for larger scaled applications.

TypeScript supports all of the latest and evolving features of JavaScript. Most notably, TypeScript supports ECMAScript 2015, and all future versions, to help build strong components. We see these features during the development period of high-quality applications. Eventually, these features are compiled into simple JavaScript that targets ECMAScript environments.

History of TypeScript

We first saw TypeScript in October 2012 after being developed by Microsoft. It was criticized for the lack of support for an Integrated Development Environment (IDE) aside from Microsoft Visual Studio, which is not available on Linux or OS X. This criticism was taken into consideration and changed in 2013, making it more readily available for integrated development environments, such as Eclipse.

In July 2014, Microsoft's development team announced a new TypeScript compiler. The new compiler claimed to be five times faster than previous versions.

The idea behind TypeScript was to develop a language to compensate for the shortcomings of JavaScript for large-scale applications at Microsoft and their customers. Microsoft saw the challenges with dealing with complex JavaScript code and a demand for a custom tool to simplify the development of components. This led developers to create TypeScript, which was a solution to not break any compatibility with JavaScript, and to have cross-platform support.

TypeScript is based off the proposal of ECMAScript's standard proposal. ECMAScript promised future support for class-based programming. From this proposal, a JavaScript compiler was created with a set of syntactical language extensions that transformed the extensions into regular JavaScript. A key aspect that differentiates ECMAScript standard proposal from the TypeScript proposal is that TypeScript added optional static typing to enable static language analysis. This helps facilitate tooling and IDE support.

Features of TypeScript

TypeScript has many desirable features that the developers took into consideration when developing Angular 2. TypeScript is very appealing to developers because it offers an optional type system. This means that developers can choose when and where to use types. This is specifically useful on the server-side. Other notable features of TypeScript include type annotations, compile-time type checking, type inference, type erasure, interfaces, enumerated types, mixins, generics, namespaces, and tuple types.

Type Annotationa

Type annotations define the inputs and outputs of a function. This may include a function's return type, the number of arguments, and the types of arguments or errors it may pass. Type annotations are intended to be light ways to record the contract of the function or variable.

An example of type annotation:

```
function greeter(person: string) {
            return “Hello, “ + person;
}

var name = “Jane Doe”

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

“:string” is part of the annotation that TypeScript offers.

Type Inference

The main purpose for type inference is to provide type information when there is no type annotation. For instance, with `let x = 5`, we can infer that x is to be `number`. We see this type of inference when initializing variables, setting parameter default values, and determining function return types.

Type Erasure

During the compilation period, TypeScript will remove type annotations, interfaces, type aliases, and other type system constructs during compilation.

Input:

```
var x: Something;
```

Output:

```
var x
```

If you take a look at this example, we have an input of `var x: Something;`. But the output is `var x;`. What we experience was during the run-time, so no information was present to say that some variable `x` was declared as being of type `Something`.

Interfaces

Interfaces have several different roles in TypeScript, such as describing an object, ensuring Class instance shape, and the static shape of a Class or Constructor Object.

JavaScript functions can take a settings object. TypeScript interfaces allow for optional properties to aid in using these objects correctly. TypeScript interfaces can also be used to represent what the type of an indexing operation is.

Interfaces describe the shape of an instance of a class. However, we can use them to describe the static shape of the class.

```
interface Person {
            firstName: string;
            lastName: string
}
function greeter(person: Person) {
            return “Hello, “ + person.firstName + “ “ + person.lastName;
}

var user = { firstName: "Jane", lastName: "User" };


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

Enumerated Types

Enums are a way to organize a collection of related values. TypeScript added this feature because JavaScript does not have this feature. If we look at the following example, we can see that when an enum is written in JavaScript, a completely different output of JavaScript is generated.

```
enum State {
           False,
           True,
         Unknown
}
```

```
var State
           (function(State) {
                    State[Sstate[“False”]=0] = “False”;
                    State[State[“True”] = 1 = “True”;
                    State[State[“Unknown”] = 2] = “Unknown”;
})(Tristate || (Tristate = {}));
```

Generics

A generic is one of the primary tools used in creating reusable components. Generics help aid in being able to create a component that can work over a variety of types instead of just one. If we did not have generics, we would have to give the identity function a certain type or describe the identity function using the `any` type.

```
function name(arg: any); any {
          return arg;
}
```

`Any` is generic and will accept all types, but we lose the information about what the type was when the function returns.

The alternative is figuring out a way to capture the type of the argument in a way that we can denote what is being returned. We can do this by using a type variable, which is a variable that works on types rather than values.

```
function name<T>(arg: T): T{
          return arg;
}
```

Mixins

From the traditional object oriented hierarchies, another way to approach building up classes from reusable components, is to build them by combining simple partial classes. Every mixin should have a specific purpose. They resemble tiny classes in TypeScript.

The mixin function is what does all the heavy lifting because the behaviors from the mixin classes are increased into a class to represent the combined behaviors. This function is necessary in your program because it is the only way to access your mixin classes.

Tuple Types

Since JavaScript does not support tuple classes, developers will tend to use an array. However, TypeScript sought to remedy this by supporting tuple classes. Tuple types allow for TypeScript to begin typing new patterns that can feature array destructuring. Tuple types also can accurately describe an array with mixed types.

```
var nameNum: [string, number];
   nameNum = [ ‘Mary’, 1234];
var [name, number] = nameNum;
```

What is ES6?

ES6 or ECMAScript 6 is the sixth version of the ECMAScript language specification. It is based on JavaScript. JavaScript is now tracking ECMAScript, and ES6 was finalized in June 2015. This edition has added some notable new syntax for writing complex applications, but also defines them semantically in the same expression as ECMAScript 5. Other notable features are iterators and for/of loops, generators, and many more.

History of ECMAScript

The ECMAScript specification is a standardized specification of a scripting language developed by Brendan Eich, who worked for Netscape and developed JavaScript. Since JavaScript was a widespread hit among developers, Microsoft ended up developing a compatible language of JavaScript and called it Jscript. Eventually, Netscape delivered JavaScript to ECMA International for standardization. The first edition of the specification was released in June 1997, with many editions being released after that.

Features of ES6

Let’s walk through some of these ES6 features.

Classes

When ES6 was released, the developers really simplified the classes with a single declarative form. This makes class patterns convenient, easier to use, and encourage interpolation.

```
class ExampleMe extends FOUR.Mesh {
    constructor(tires, cars) {
        super (tires, cars);

this.idHouse = ExampleMe.defaultHouse();
this.box = [];
this.boxMatrices = [];
}
update( me) {
    super.update();
}
static  defaultHouse() {
return new FOUR.House7();
}
}
```

Enhanced Object Literals

Object literals are to support setting the prototype at construction. They are also used for defining methods, making super calls and figuring property names with expression. The benefit of enhanced object literals are that they bring class declarations closer to object literals. This, in turn, lets object-based design benefit from these handy features.

```
var example = {
__proto__: theProtoObj,
hander,

toString() {
    return “d “ + super.toString();
},
    [  ‘prop_’ + (() => 44)() ]: 42
};
```

Arrows

Arrow function shorthand was added to support statement block bodies and expression in bodies that return the expression. It is a similar idea to CoffeeScript.

```
var odd = even.map(v => v +1);
```

Template Strings

Template strings were added to provide a stronger and cherry-on-top syntax for constructing strings. Tags can be added to allow for the strong construction to be customized or avoid injection attacks.

```
‘This is my example of literal string creation.'
```

Destructuring

Destructuring allows for types of binding used pattern matching. This supports matching arrays and objects. Destructuring will produce `undefined` values when those values are not found.

```
var [a,  , b] = [1, 2, 3];

var [a = 1] =[];
a === 1;
```

Generators

Generators were designed to simplify iterator-authoring using `function*` and `yield`. You can declare a function as `function*`, and it will return as a Generator instance. Generators are a secondary type of iterators that include additional `next` and `throw`. This allows the values to drift back into the generators. So, `yield` is an expression form that will return a value.

```
var example = {
    [Symbol.iterator]: function*() {
        var before = 0, cur = 1;
for (;;) {
    var pseudo = before;
    before = cur
            cur += pseudo;
            yield cur;
        }
    }
}

for (var n of example) {
    if (n >100 )
   break;
    console.log(n);
}
```

Modules

The ES6 modules main goal was to create a format that users could be happy with. The modules are to include declarative syntax for importing and exporting and programmatic loader API to configure how modules are loaded.

Export:

```
export function add( a, b) {
    return a + b;
}

export var pi = 3.14;
```

Import:

```
import {add, pi} from “lib/math”;
alert(“2pi = “ + sum(pi, pi));
```

Symbols

Symbols are used to allow access for control for the object state. Symbols allow for properties to be keyed by a `string` or a `symbol`. Symbols are considered types. You can use an optional `description` parameter for debugging. Symbols are unique, but not private, because they are exposed through reflection features.

```
var Class = (function() {

  var key = Symbol("key");

  function Class(privateData) {
    this[key] = privateData;
  }

  Class.prototype = {
    stuff: function() {
      ... this[key] ...
    }
  };

  return Class;
})();

var a = new Class("hello world!")
a["key"] === undefined
```

For...Of loops

ES6 introduces a loop that iterates over iterable objects (such as arrays, map, string, etc.). This is a handy tool, because you can create custom iteration hooks with a statement to be executed for the value of each property. For..of loops are specific to collections, instead of objects. The loop will iterate over the elements of any collection that has a [Symbol.iterator] property. `let` can used as a constant as long as the variable is not modified inside the block.

```
let example = {
  [Symbol.iterator]() {
    let before = 0, current = 1;
    return {
      next() {
        [before, current] = [current, before + current];
        return { done: false, value: current }
      }
    }
  }
}

for (var n of example) {
  if (n > 100)
    break;
  console.log(n);
}
```

Map + Set + WeakMap + WeakSet

For common algorithms, Map + Set + WeakMap + WeakSet was developed.

```
var set = new Set();
set.add("hey").add("bye").add("hey");
set.size === 2;
set.has("hey") === true;

var map = new Map();
map.set("hey", 2);
map.set(set, 4);
map.get(set) == 4;

var weakMap = new WeakMap();
weakMap.set(set, { extra: 2 });
weakMap.size === undefined

var weakSet = new WeakSet();
weakSet.add({ data: 2 });
```

Proxies

In ES6, Proxies allow for the creation of objects with a range of behaviors to be able to host objects. Proxies can be used for interception, object virtualization, logging/profiling, and more. You need to remember that proxies cannot be transpiled.

```
var a = {};
var b = {
  get: function (receiver, name) {
    return `Hello, ${name}!`;
  }
};

var p = new Proxy(a, b);
p.world === "Hello, world!";
```

Promises

Promises are a library for asynchronous programming. Promises are a class representation of a value that can possibly be made available in the future.

```
function timeout(duration = 0) {
    return new Promise((resolve, reject) => {
        setTimeout(resolve, duration);
    })
}

var p = timeout(100).then(() => {
    return timeout(200);
}).then(() => {
    throw new Error("hmm");
}).catch(err => {
    return Promise.all([timeout(10), timeout(20)]);
})
```

What are shims?

Shims are a small library that can intercept API calls and change the argument that passes or redirects the operation. Developers usually see shims when the behavior of an API changes, which in turn, causes compatibility issues for older applications. This can mess up the functionality of the application. When this happens, shims are used for running the program on different software than what they were developed for.

What are polyfills?

Polyfill’s are browser fallbacks that are made in JavaScript. Polyfill’s allow functionality that work in most browsers. Polyfill’s use additional code to provide facilities that are not built into a web browser.They also provide a more uniform API landscape.

Should You Use TypeScript or ECMAScript?

TypeScript shouldn’t be used in every situation. As a developer, you should determine if TypeScript is the appropriate plan of attack for your Web Application. Some things you should take into account for when determining if TypeScript is appropriate are the code size, compatibility with ES6, or if you are using Angular 2.0.

Some Web Applications can have a huge source. Errors can happen all the time because we are human and not perfect. But, we can be proactive in preventing these errors. TypeScript has a safety mechanism built into it to prevent code from breaking when someone goes in and tries to change code. The TypeScript transpiler will reveal where your bugs are and helps identify those obvious mistakes because it is so readable.

ES6 isn’t right for every application out there. Since JavaScript is defined by a committee, ES6 is considered the standard right now. However, the downside of ES6 is browsers do not implement the features of ES6 natively and have to be compiled down to ES5.

Summary

When building an Angular 2 application, you must choose the best scripting language that will suit the purpose of that application. ECMAScript 6 has been around a number of years and has a well established reputation among the Angular developers community. It is a powerful scripting tool that can help aid in making substantial Angular 2 web applications. However, TypeScript specifically supports ECMAScript and all the features it offers.

Our sample application is written in TypeScript throughout this book.

Chapter 3. Introducing Our App

Angular 2 is very popular right now, and there are many great resources for getting started with the framework (we will be listing to many of them as we go along). We will not, however, be lingering on the basics, but rather taking a more hands on approach in this book, which we see as adding value.

Throughout this book we will be building a dashboard application with a few features. While building the application we will be covering all of the basic parts of Angular 2, as well as more complex functionalities. Keep in mind that Angular 2 was only just released, and we will be demonstrating development strategies we think best suit the framework.

The entire application is openly available at https://github.com/flauc/angular2-edge-app.

One of the first things to consider when starting a new application is how to structure it. This might seem like a small thing, but when you make these decisions at the beginning of the project, you end up saving a lot of time later on as you create it.

Folder Structure

We found that there are two good folder structures for Angular 2 web applications, depending upon their size.

If you are building a small application with fewer separate files, consider using the following structure:

- app [folder]
  - components [folder]
    - something [folder]
      - something.component.ts
      - something.html
  - services [folder]
  - pipes [folder]
  - directives [folder]
  - app.component.ts
  - boot.ts
- assets [folder]
- index.html
- system.config.js
- tsconfig.json

The app folder is the application's main folder. It contains all of the code related to Angular 2. The components, services, pipes and directives folders contain the various parts of an Angular 2 application. It is typical for only the components folder to have sub-folders. This is done so that everything related to a certain component has it's own folder.

The assets folder is intended for all of the styles, images, external JavaScript, and so on.

This works great with small applications, because the structure is at most two levels deep, thus making every file easily available. However, with larger applications, this structure becomes messy and hard to maintain.

When building larger applications, you shouldn't place files only based on type (component, service, etc.), but rather into folders representing the part of the application containing that file.

Consider a web application with multiple pages. Here, for example, you might divide the

folders based on these pages, with a common folder that contains all of the code that is shared over multiple pages. This is what you end up with:

- app [folder]
    - common [folder]
        - components [folder]
        - services [folder]
        - pipes [folder]
        - directives [folder]
        - interfaces [folder]
        - config [folder]
            - app.values.ts
    - pages [folder]
        - Home [folder]
            - home.component.ts
            - home.html
        - Login [folder]
            - contact.component.ts
            - login.component.ts
    - app.component.ts
    - boot.ts
- assets [folder]
- index.html
- system.config.js
- tsconfig.json

This is the folder structure used in most of the applications we build. It makes files easy to locate, because the structure mimics the way that the application appears to the end user.

If the application had more levels of depth, such as a user and an admin area with it's own sub pages, you could still mimic it in the folder structure:

- pages [folder]
    - admin [folder]
        - dashboard [folder]

        - settings [folder]
    - user [folder]
        - dashboard [folder]
        - settings [folder]

The application we will build for this book isn't that large. You might argue that we could just get away with using the first approach, but given how most applications are usually larger and more complex, we will use the latter approach instead.

Application Setup

Setting up an Angular 2 browser application requires a bit more work than you might be used to with AngularJS or a similar framework. You will most likely be using Typescript, which will require compiling, and then you need to set up the module loader of your choice (we'll be using system.js). Angular does a great job of explaining the basic setup in it's "5 min quick start," and we highly recommend you go through that, but here we'll jump right in to coding.

You can build the basic folder structure and application setup yourself, or you can clone our github repository.

To get the project in to the right state for this part of the book, run the following commands in your console:

note: Read through the readme to see if you have all of the npm packages installed accordingly.

```
git clone https://github.com/flauc/angular2-edge-app.git
git checkout start-up
npm run build
```

Now that you have all of the right files and everything is injected and compiled properly you can start working on the application.

note: You can safely ignore all of the code outside of the public folder. That's the server we'll be communicating with in later stages of our application.

Hello World

Following the standard programming tradition, let's first start with building a "Hello World!"

location: `public/app/app.component.ts`

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

@Component({
    selector: 'app',
    template: `<p>Hello World!</p>`,
})
export class AppComponent {
    constructor() {}

}
```

This is the `AppComponent`, which is the entry point for the application. The `AppComponent` is of course a component, and components are the building blocks of every Angular 2 application. They are classes decorated with the `@Component` metadata that control parts of the DOM through its user-facing template. Angular 2 evaluates and places the components template in to the DOM wherever it encounters the matching selector element, and we've placed the app element inside of our index.html file.

note: We will take a closer look in to components in Chapter 4.

location: public/index.html

```
<!--head-->
<body>
    <app></app>
</body>
```

At this point you can call `npm start` to start the application. Now navigate to `http://localhost:5000` and you should see the "Hello World!" paragraph.

note: You can change the port on which the application is running on in the `server/config/config.ts` folder.

Template Syntax

Let’s go ahead and add some template binding to our “Hello World!” application.

location: `public/app/app.component.ts`

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

@Component({
    selector: 'app',
    template: `
        <p>{{hw}}</p>
        <button [class]="buttonClass" (click)="sayHi()">Say Hi!</button>
    `

})
export class AppComponent {
    constructor() {}

    public hw = 'Hello World!';
    public buttonClass = 'special';

    sayHello() {
        console.log('Hi!');
    }
}
```

We’ve added a lot of code here, so let’s go through it step-by-step.

Interpolation

When looking at the template you first come across the double-curly braces syntax `{{ }}`. You might recognize it from AngularJs. The syntax is called **interpolation** and it's a form of **one-way data binding** where data flows from the data source to the template. The content inside of `{{ }}` is a **template expression**. During view initialization, the template expression gets evaluated and then converted to a string.

With some exceptions, most valid JavaScript expressions are also valid template expressions. Here are a few examples:

```
<p>{{'Hello' + ' World!'}}</p>                  <!-- Hello World -->
<p>{{'Hello World!'.length*2}}                  <!-- 24 -->

<!-- We can also call properties and methods from the component -->
<p>{{hw}}</p>
<p>{{sayHi()}}</p>
```

Template expressions are powerful, but they can also be dangerous at times. As a rule, you should never write anything inside of them that alters the component. Angular guards against this by not allowing assignment operators (`=`, `+=`, `*=`...), for example. But you can still call a method that changes a property, and Angular has no way of knowing. This is dangerous, because a template expression can get evaluated multiple times during the component's lifecycle (you'll be hearing more about this in Chapter 4), causing for some unexpected behavior.

Square bracket syntax

We have used the square bracket syntax `[]` in our “Hello World!” application to bind to the button elements class property like this: `[class]="buttonClass"`. Like interpolation, it’s also a form of one-way data binding where data flows from the data source to the template, but this syntax allows us to bind to properties and attributes of HTML elements. Also as you will see in Chapter 4. it allows us to bind to properties of Angular 2 components and directives.

Event Binding

Similarly to how we use the square bracket syntax for binding to properties and attributes, you can use the parentheses syntax `()` for binding to events. You can bind to any HTML attribute event using this syntax as well as custom component and directive events. When binding to HTML event attributes, omit the “on” from the beginning of the word. For example, if you want to bind to the HTML’s `onclick=" "` event, use the following `(click)=" "`.

Event binding is a form of one-way data binding where the data flows from the template to the data source.

Two-way Data Binding

The `[(ngModel)]` syntax is a form of two-way data binding in Angular 2. It's primarily used when writing forms allowing us to both display a data property and update that property when the user makes changes. You use it by declaring a property in our component and assigning it to a form element with `[(ngModel)]`.

```
@Component({
    template: `<input type="text" name="name" [(ngModel)]="name" />`
})
export class SomeComponent {
    public name: string;
}
```

This syntax only works for the following HTML native elements:

- input
- textbox
- select

note: It could work with a custom component if you write a suitable value accessor.

You can also listen to changes on the model by binding to the `(ngModelChange)` event.

```
<input [ngModel]="name" name="name" (ngModelChange)="doSomething()" />
```

We will explore this syntax in a lot more detail in Chapter 9, Forms.

Built-in Directives

Unlike in the previous installment of Angular, in Angular 2 you only get a handful of built in directives out of the box. This is primarily because of how powerful the binding syntax is. In Angular 2 you only need to write custom directives if you want to create very specific functionality. In most common cases using any of the aforementioned bindings will do the trick. You do, however, still get some built-in directives.

To see them in action, write a new component for the application. Let's call it the `UserBlockComponent`. It should display a list of registered users and their status (online, offline).

location: `public/app/common/components/user-block/user-block.component.ts`

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

@Component({
    selector: 'edge-userBlock',
    template: `
        <button (click)="isVisible = !isVisible">Open Users</button>
        <ul class="users-block" *ngIf="isVisible">
            <li *ngFor="let user of users; let e = even; trackBy:trackById">
                <span [ngClass]="{online: user.status === 'online', offline: user.status === 'offline'}"></s
                <span>{{user.username}}</span>
                <span [ngStyle]="styling(e)"> Even: {{e}}</span>
            </li>
        </ul>`
})
export class UserBlockComponent {
    public users = [
        {id: 1, username: 'filip.lauc93@gmail.com', status: 'online'},
        {id: 2, username: 'laco0416@gmail.com', status: 'offline'},
        {id: 3, username: 'mgualtieri7@gmail.com', status: 'online'},
        {id: 4, username: 'ran.wahle@gmail.com', status: 'online'},
        {id: 5, username: 'wojtek.kwiatek@gmail.com', status: 'offline'}
    ];

    public isVisible: boolean = false;

    styling(even) {
        return {
            'background': even ? '#8ff76f' : '#6fccf7',
            'color': even ? '#333' : '#fff',
            'font-style': even ? 'italic' : 'normal'
        }
    }

    // Improves performance of *ngFor
    trackById(index, user) {
        return user.id
    }
}
```

note: At this point you could import the `UserBlockComponent` in to our `AppModule` (or in to any other component) using this syntax:

```
@NgModule({
    declarations: [
        AppComponent,
        UserBlockComponent
    ]
})
```

And then use it in the `AppComponent` like this:

```
import {UserBlockComponent} from 'common/components/user-block/user-block.component'
@Component({
    selector: 'app',
    template: `
        <edge-userBlock></edge-userBlock>
        <p>{{hw}}</p>
        <button [class]="buttonClass" (click)="sayHi()">Say Hi!</button>
    `
})
```

NgIf

The first built-in component we used is `*ngIf`.

note: The `*` that appears before the directive name is syntactic sugar that makes it easier to read and write directives that modify HTML layout with the help of templates. You can read more about it on angular.io.

It's quite straightforward if the expression evaluates to falsy the element is **removed** from the DOM, but if the expression is truthy, the element is inserted in to the DOM. Keep in mind that the element and all its content aren't hidden, but are completely removed when the expression is false.

note: The Mozilla developer site is a great resource for javascript. You can read more about truthy and falsy values there.

NgFor

`*ngFor` is a bit more of a complex directive. This is the basic syntax: `*ngFor="let user of users"`. The directive copies the HTML block for each item of the array. Each item is accessible inside of the HTML block using the input variable we defined (`let user`). In our component the user's property is an array of objects, which each has an object with a username and a status. Therefore, you can access the values of the current iteration like this: `user.username` or `user.status`.

As you can see from `UserBlockComponent` the `*ngFor` syntax is a bit more nuanced then `*ngIf`. `*ngFor` provides several exported values, which you can append to template variables and use in your template. Here are the available exported values:

- `index` holds the integer value of the current loop iteration
- `first` a boolean value indicating whether the item is the first one in the iteration
- `last` is true if the item is the last of the iteration
- `even` a boolean value that is true if the item has an even index
- `odd` a boolean value that is true if the item has an odd index

We bind one of the values to a template variable using this syntax: `let i = index`. We can of course bind to multiple values if required: `let i = index; let e = even`...

Another feature we have used in our component is `trackBy`. This is done so that Angular 2 doesn't have to re-build the entire list when something changes. You tell Angular 2 that two users with the same `id` are in fact the same user. In our component we don't really manipulate the users array in any way, but this is still a good practice to hold on to for later use.

NgSwitch

We haven't really made use of `*ngSwitch` in our component, but it's another built in directive that fits in the same category like `*ngIf` and `*ngFor`, because it also manipulates the template. Here is a simple example:

```
<div class="wrapper" [ngSwitch]="user.favoriteAnimal">
    <div *ngSwitchCase="'dog'">
        <i class="icon">dog</i>
    </div>

    <div *ngSwitchCase="'cat'">
        <i class="icon">cat</i>
    </div>

    <div *ngSwitchCase="'crocodile'">
        <i class="icon">crocodile</i>
    </div>

    <div *ngSwitchDefault>
        <i class="icon">default</i>
    </div>
</div>
```

`*ngSwitch` works much in the same way a switch loop works. The HTML block that satisfies the expression gets injected in to the DOM, or if none of the values match then it defaults to the `*ngSwitchDefault` block. For example, if `user.favoriteAnimal === 'dog'`, the first block would get injected in to the DOM.

note: We have only mentioned one syntax for writing `*ngIf`, `*ngFor` and `*ngSwitch` directives, but all three have an alternate syntax you might make use of. This is the `<template>` syntax, and the `*` syntax is just a shorthand for it, that is used for easier reading and writing of the code. You can read more about the template syntax here.

NgClass and NgStyle

You can use `[class.active]="isActive()"` or `[style.width.px]="widthToSet()"` to append a single class to an element or change a style. However, when you need to dynamically add/remove/change multiple classes or styles, this syntax becomes messy. This is why Angular 2 has the built in directives: NgClass and NgStyle.

We have made use of the NgClass directive in the component like this:

```
<span [ngClass]="{online: user.status === 'online', offline: user.status === 'offline'}"></span>
```

NgClass behaves slightly different based on the type of value it receives. It accepts any of the following values:

- `string` NgClass adds all the received classed to the element. The different classes need to be separated by spaces

- `array` All of the array elements get added as classes to the element
- `object` The object key corresponds to the class and the value needs to resolve to a boolean that determines whether the class should be added or removed (this is the syntax we used)

NgStyle accepts an object where the keys correspond to the CSS style and the value is in the right format for that style:

```
<span [ngStyle]="{'background': 'red', 'color': 'black', 'font-size': '20px', }">I got styled</span>
```

Introduction To Services

In the `UserBlockComponent` that we built earlier, we used a simple sample array of users. Even though this is just a simple application, you should still stick to best practices. In Angular 2 services should **handle data access and server communication** and your components should focus on **displaying the data**.

Maintaining separation of concerns helps scale and preserve good structure later on in your application development. It also allows you to write proper unit tests, as you will see in Chapter 7, Testing.

Let's start our introduction by writing a service for the `UserBlockComponent`.

location: `public/app/common/services/user.service.ts`

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

@Injectable()
export class UserService {

    private users = [
        {id: 1, username: 'filip.lauc93@gmail.com', status: 'online'},
        {id: 2, username: 'laco0416@gmail.com', status: 'offline'},
        {id: 3, username: 'mgualtieri7@gmail.com', status: 'online'},
        {id: 4, username: 'ran.wahle@gmail.com', status: 'online'},
        {id: 5, username: 'wojtek.kwiatek@gmail.com', status: 'offline'}
    ]

    get() {
        return this.users;
    }
}
```

In Angular 2 a service is just a class that an application requires. The `@Injectable` decorator emits metadata about the service that Angular 2 needs, so it can resolve other dependencies. Even though our service doesn't have any dependencies at the moment, it's considered best practice to add it anyway.

note: We didn't do much here. We just moved the "hard coded" users array in to a service and provided a method for getting the users. In a "real world" application you would need to fetch the data with HTTP or Web Sockets and the get method might return an HTTP response. We will do this in Chapter 6, Services, but for now let's stick to the essentials.

Now that we created our service, here is how we use it in our `UserBlockComponent`:

1. First You need to register a **provider** of the `UserService` in the `AppModule`:

```
@NgModule({
    declarations: [
        AppComponent,
        UserBlockComponent
    ],
    providers: [UserService]
```

```
})
```

2. Import the `UserService` class just like any other exported class in typescript. `import {UserService} from '../services/user.service'`
3. Define your service property. This is done in your class constructor. `constructor(private _userService: UserService) {}` This line simultaneously defines the injection site of the `UserService`.

This is what the component looks like now:

location: `public/app/common/components/user-block.component.ts`

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

// Import the service
import {UserService} from '../services/user.service'

@Component({
    selector: 'edge-userBlock',
    // The template...
})
export class UserBlockComponent implements OnInit {
    constructor(
        // Define the service
        private _userService: UserService
    ) {}

    // All our methods and properties...
}
```

Now that you can use the `UserService` in your component, let's use it to load the array of users.

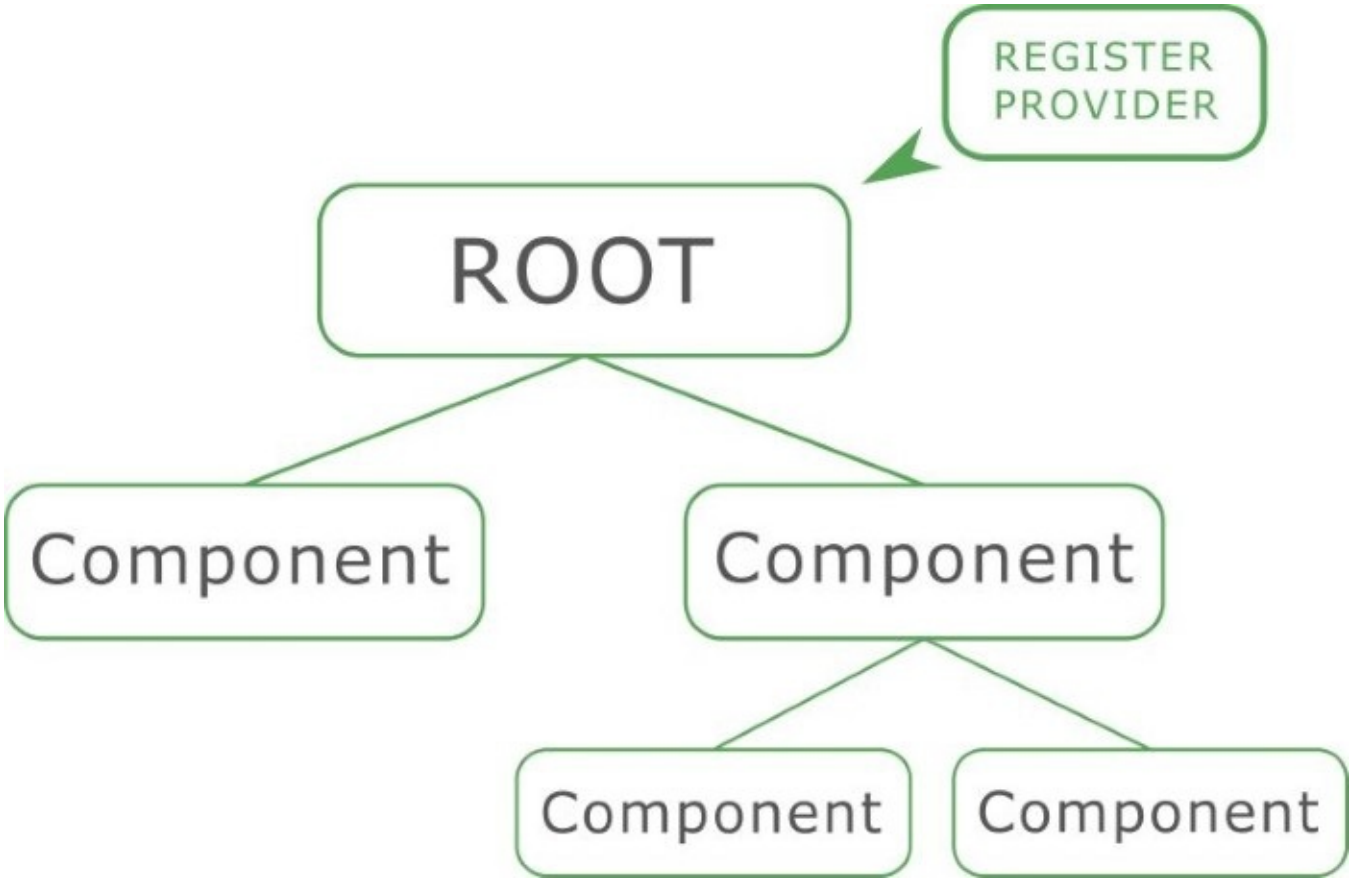
```
export class UserBlockComponent implements OnInit {
    // ...constructor

    public users;

    ngOnInit(): void {
        this.users = this._userService.get();
    }
}
```

It's a best practice to do this kind of initialization request inside of the `ngOnInit` **lifecycle hook**. You will learn all about the **component lifecycle** in Chapter 4, Components, but for now it's important to note that it's always better to do logic like fetching data from services in `ngOnInit` rather than in the components constructor. Although the end result is pretty much the same, testing constructors is a lot harder, and you should limit what constructors do to a simple instance configuration.

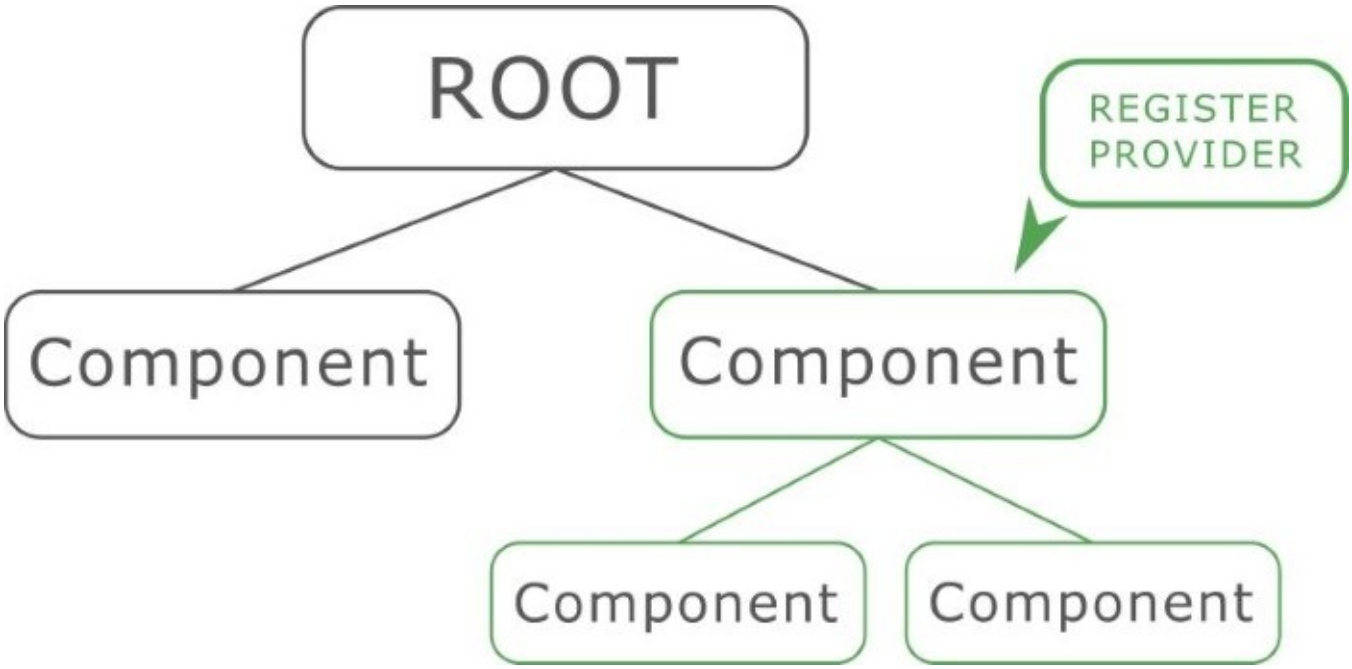
Dependency Injection

Registering the provider is the most interesting part we did, and deserves a closer look. Using `providers` in the `AppModule`, register a provider for the service. The provider tells the **injector** how to create the service. In Angular 2 the injector is responsible for maintaining a container of service instances and creating new ones if required by a component if they aren't already in the container.

You can register the service provider in any module as well as at the root in `AppModule`. If you register at the root, then the same instance of the service is available across the entire application.



The same instance of the service is available to all of the declarations of the module (all of the components, directives and pipes) where you have registered the provider for the service. This means that if you register your service provider lower down the application tree, a new instance of the service will be created for that module. It will also be available to all of the components of that module and so on all the way down the tree. This is what we have done in `UserBlockService`.



As you can see, Angular 2 provides a lot of functionality with its Dependence Injection framework. We might often register the service provider at the root level so that we have a singleton service across the entire application, or we might register it in a component and have different instances of the service.

These are both perfectly valid approaches in Angular 2, and depend only on the logic that the application requires.

Basic Routing

Client side routing has become a central part of web development, and Angular 2 like its predecessor, comes with a great routing library. We will explore the nuances of routing in Chapter 8, Routing, but for now let's create the basic outline of the application that will later provide login and signup functionality among other features.

Before we can start creating routes for the application, there is a bit of configuring to do.

location: `public/index.html'

```
<html>
    <head>
        <base href="/">
        <!-- Other config -->
```

We need the `<base href="/">` element tag to make routing work. You can read more about it here.

Now we create a router configuration file:

note: We are creating the basic outline for an application that will later provide login and signup functionality.

location: `public/app/router.config.ts`

```
import {Routes, RouterModule} from '@angular/router';
import {HomeComponent} from './pages/home/home.component';
import {LoginComponent} from './pages/login/login.component';
import {SignUpComponent} from './pages/sign-up/sign-up.component';

const routesConfig: Routes = [
  { path: '', component: HomeComponent },
  { path: 'login', component: LoginComponent },
  { path: 'signup', component: SignupComponent },
  { path: '**', redirectTo: '' }
];

export const routing = RouterModule.forRoot(routesConfig);
```

RouterConfig

The `Routes` that we have created is an array of objects called **Router Definitions**. We provide a path (with out the leading '/') and a component in each Router Definition. When the URL changes to match the path segment the appropriate component is created and its view gets instantiated inside of the `<router-outlet></router-outlet>` element which we will create in out `AppComponent`. The router declaration with the `'**'` path redirects to the HomeComponent if no other path was matched.

RouterModule.forRoot()

The `RouterModule.forRoot()` function tells Angular 2 how to configure our routes.

Now we need to import our `routes` in our `AppModule`.

location: `public/app/app.ts`

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

Now we can start creating routs in our `AppComponent`:

location:_ `public/app/app.component.ts`

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

@Component({
    selector: 'app',
    template: `
        <nav>
            <ul>
                <li><a routerLink="">Home</a></li>
                <li><a routerLink="/login">Log In</a></li>
                <li><a routerLink="/signup">Sign Up</a></li>
            </ul>
        </nav>
        <router-outlet></router-outlet>
    `,
})
export class AppComponent {}
```

note: We went ahead and removed all of the code from the “Hello World” example.

We add the `<router-outlet></router-outlet>` as mentioned previously.

RouterLink

We use the `RouterLink` directive to link to a defined route. The `RouterLink` just like the `RouterOutlet` is part of the `RouterModule`. This is how we use it: `routerLink="/"` or `[routerLink]="['/']"` the template expression needs to return an array of link parameters. For now, you only need to pass in the path as the first element of the array, but as you will see later we can also pass in an id and query parameters if required.

note: We can also use the `routerLinkActive` directive if we want to add a specific class to the active route.

Before you can give your routing a test run, you need to create all of the components used. We will fill out these components as we go along through the book. For now, let's just define and import them in `AppComponent`.

location: `public/app/pages/signup/signup.component.ts`

```
@Component({
    selector: 'edge-signup',
    template: `<h1>Signup Page</h1>`,
})

export class SignupComponent { }
```

location: `public/app/pages/login/login.component.ts`

```
@Component({
    selector: 'edge-login',
    template: `<h1>Login Page</h1>`,
})

export class LoginComponent { }
```

location: `public/app/pages/home/home.component.ts`

```
@Component({
    selector: 'edge-home',
    template: `<h1>Home Page</h1>`,
})

export class HomeComponent { }
```

We will also need to add the created components to the `AppModule`:

location: `public/app/app.module.ts`

```
@NgModule({
    imports: [
        BrowserModule,
        routing
    ],
    providers: [
        UserService
    ],
    declarations: [
```

```
        AppComponent,
        UserBlockComponent,

        // Pages
        HomeComponent,
        LoginComponent,
        SignupComponent
    ],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Now if you run the following console commands: `npm start`

You will see basic routing in action.

note: If you are having trouble getting something to work, you can always refer to our repository on github.

Angular CLI

Angular CLI is an alternate route into creating your application. The command line interface is used to scaffold your application. It is an easy and quick way to begin building your application.

You can reference Angular CLI repository for a reference on Angular CLI.

Installing Angular CLI

In order to install Angular CLI, you must first have NPM and Node installed locally on your machine. Then we can run the command `npm install -g angular-cli`. This insures that Angular CLI is installed globally.

Let’s walk through a simple application set up using Angular 2 CLI.

Creating your Angular 2 application using CLI

First, we need to run the command below to create a new Angular 2 application. This begins the scaffolding process.

```
ng new AppName –prefix Name
```

Build & Serve

Next, we need to build and serve our application. We use the commands:

`ng build` and `ng serve`.

`ng build` will create a “dis/” folder that contains our complied application. `ng serve` will run location but will not open any browser.

Creating a model & service

Now, lets create a model and service for our application.

//create model

```
ng generate class share/name model
```

//create service

```
ng generate service shared/name
```

Router

Finally, we need to create a router for our Angular 2 application.

```
ng generate route name
```

Final thoughts on Angular CLI

There are other scaffolding command that you can run, in order to generate the code that you need.

Angular 2 CLI is a preference choice. As a developer, it is up to you to make the decision if you want to use Angular 2 CLI. It can be a great feature to use for someone who is pressed for time.

Chapter 4. Modules

Angular 2 has the ability to contain components, services, and pipes in a module. The module can depend on other imported modules.

What is a module?

An Angular module is a container of components, directives, pipes, and injectable classes. By declaring a module we can have its entire components compile ahead of time and therefore save time when loading an application.

One of the things that happens through the compilation is the creation of the <component>.ngFactory class, which is the product of the component's template compilation.

Module declaration

To declare the module you may create a class and decorate it with the @NgModule decorator. This decorator accepts an object that has the following properties:

1. Providers: an array of providers, which are an injectable objects that can be used internally in this module.
2. Declarations: An array of components / directives / pipes that belongs to this module.
3. Exports: A list of components / directives / pipes that can be used in any module imports this module.
4. entryComponents: A list of components that should be compiled as well when this module is defined.
5. bootstrap: A list of components that needs to be bootstrapped when this module is bootstrapped.
6. schemas: A list of elements that are not Angular components or directives that needs to be used by our module and therefore needs to be declared in the schema.
7. id: An opaque id used to register this module in NgModuleFactory.

In case you have come from Angular 1.x and you are familiar with the angular.module method, please note the difference. Angular 2.0 has all module components declared in the module declaration, while in Angular 1.x you probably registered each module component (i.e controller, service, factory, filter, etc.) using the angular.module method.

Why Modules again?

For those of you who followed Angular 2.0 from its earlier beta (or even alpha) stages, you must have noticed that modules was introduced to Angular 2 on its fifth release candidate. Prior to that RC, our Angular application was built with one bootstrapped component that consumed all other components, directives, providers, and pipes. This type of application structure forced Angular to compile each part on its runtime and by the browser. This is a JIT (Just In Time) compilation.

What does compilation means?

To better understand what this module structure provides, you must first understand what is the “compilation” of Angular. Since you are a programmer you may think of this term as “translating a programming language code to machine code,” but in Angular this is, of course, not the case.

Compilation in Angular 2.0 is basically everything between our application code and the running application. Here it refers to the component’s templates parsing and instantiations. When a template is loaded, Angular does several things with it. It creates a JavaScript context code auto-generated for the template while traversing its DOM for elements and attributes. Looking at your Angular application, you may find it under <Your component>.ngFactory.js. This code is generated by your browser using the Angular compiler.

A good video explanation about the Angular compilation can be found here: https://www.youtube.com/watch?v=kW9cJsvcsGo

Compiling in build time

Angular modules enable us to create those files on our build time and save the browsers with the load of compiling each template. You need to install npm packages to your application:

1. @angular/compiler-cli
2. @angualr/platform-server

The first package comes with the “ngc” command line, which you may run later to build your code, while the second one is for using Angular with server side (Node.js).

After installing the two packages (using npm install) you may create a file similar to the tsconfig.json in the quickstart, you may call it tsconfig-aot.json (you may use that name in the command parameter you will send to ngc).

Here are the changes:

1. The compilerOption.module has the “es2015” value instead of “commonJs.” This is to enable the Angular compiler, reducing the downloaded code by removing unused references. This feature is enabled by the AoT compiler’s Tree-Shacking step (see more below), which comes after the compilation itself.
2. A whole section called “angularCompilerOptions” needs to be added with two members:

a. *genDir* - The directory where the generated files are put. b. *skipMetadataEmit* (boolean), when true - prevent the compiler from making metadata files.

Here’s a sample for that file:

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "es2015",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": true,
    "suppressImplicitAnyIndexErrors": true,
    "types": []
  },

  "files": [
    "app/app.module.ts",
    "app/main-aot.ts",
    "./typings/index.d.ts"
  ],

  "angularCompilerOptions": {
    "genDir": "aot",
    "skipMetadataEmit" : true
```

```
  }
}
```

After doing so, you may simply run the following command:

```
node_modules/.bin/ngc -p tsconfig-aot.json
```

If you’re using Windows, you may want to surround the path with quotes.

Now, instead of your AppModule, the compiler has created an AppModuleFactory, which contains your module declaration after compilation. You should replace your main module with the newly factory created. Therefore, on your main.ts file, don’t import your module, but your ModuleFactory from your AoT path. The bootstrapping code below illustrates that:

```
import { platformBrowser }    from '@angular/platform-browser';
import { AppModuleNgFactory } from '../aot/app/app.module.ngfactory';
platformBrowser().bootstrapModuleFactory(AppModuleNgFactory);
```

What the compiler will do is create your compiled application in the path you have defined for it in the angularCompilerOptions.genDir property. You may look there at the <component>.ngFactory.ts files

TreeShaking

We mentioned earlier the Tree-Shaking part in the compilation process. At the end of this process you may want to have a minified JavaScript file that contains your transpiled code along with the compiled code. The AngularJs official tutorial demonstrates the rollup utility, which is a Tree-Shaking utility.

Use JiT for development and AoT for production

One of the best practices regarding AoT is to leave JiT for development, but to go with AoT on production. You might want to do so without changing the code. One problem you might face is when using a templateUrl on your component decorator, while the JiT template URL is flexible, the AoT require templateUrl to relate to the component's location.Stick to relating the *templateUrl* to the component class file.

One other practice you may stick to is a complete separation of Aot-related files from their corresponding JiT. For example, you have tsconfig.json and tsconfig-aot.json. Keep index.html, which consumes the JavaScript build artifact in a different folder. Use two files for your bootstrapping code (main.ts and main-aot.ts). Your goal is to have your AoT compiled up, which looks and behave the same as your JiT compiled app, only faster, without changing your code.

Summary

Angular uses modules to enable its compilation to come not only on the runtime (JiT) compilation, but to have your application served from the server compiled already (AoT). This can be done, first of all, because Angular works in Modules and not in small pieces, because it used to be before AoT was introduced (rc-5). It is very important that AoT-related files be kept separately from the files they override, so you are able to have JiT on your development environment, but AoT on your production side.

Chapter 5. Components

Components are Angular 2 building blocks that contain both logic and UI behavior. Components contain both HTML templates and classes. You can say that component classes are like Angular 1.x controllers.

You can compare components to an Angular 1.x directive, however, a definition of a component is one autonomic module that contains both UI and UI-oriented logic. If you are familiar with Angular 1.x, you may compare a controller to a small module that contains both an HTML template and a controller, all while the exported class acts as the controller.

Components in Angular 2.0 increase the HTML element name vocabulary with the app's components, in the same way that HTML5 web components are supported in Angular 2.0.

While building a component we will provide it with both HTML template and UI related logic. The UI related logic is on a class that we export. The class is decorated with a component decorator function that receives an object with the HTML template as one of its members.

Component decorator

A component contains an exported class decorated with a component decorator. A component decorator is a function that gets a ViewMetaData typed object as a parameter.

When creating a component, we have to decorate our class with the component decorator. The component decorator is a JavaScript function that receives a component description object as a parameter. This object is of type ViewMetaData and contains the following properties:

1. **selector** : The selector for the component. It can be a new HTML element name, attribute, class, etc. In most cases, the selector is set for a new HTML element name.
2. **template** : an HTML markup to be rendered when the component is set.
3. **templateUrl**: A URL points at an HTML file that contains the template.
4. **directives**: An array of types of other components or directives used by this component.
5. **pipes**: An array of pipes used by the component as pipes (See pipes later on in this chapter).
6. **Styles**: An array of strings where each string is a style rule.
7. **styleUrls**: A URL to an external CSS file.
8. **encapsulation**: The encapsulation type of the view.
9. **providers**: An array of types of injectable services used by the components, so whenever the component is instanciated, an instance of each provider is instanciated as well.

A component is structured as follows:

```
import {Component} from '@angular/core'
/**
Another imports
**/
@Component({/*ViewMetaData object */})

export class MyComponentClass {
}
```

The component template The component template may be written inside the component decorator in the ViewMetaData's template member, or written in a separate file while the URL is given by the templateUrl member of the ViewMetaData object.

Please note that the template URL is relative to the application root and not to the component.

ViewEncapsulation ViewEncapsulation is an enumeration used to determine how styles will be encapsulated in the component. It has three entries:

1. Native: Use the browser’s native encapsulation mechanism. This means using Shadow DOM for the component and create a Shadow root on the component’s host element.
2. Emulated: Emulates the native encapsulation. The emulation is done by adding a surrogate ID to the host element and preprocess the style rules given on the ViewMetaData object of the component. This is the default option and it will be used unless stated otherwise.
3. None: No style is encapsulated for the component.

Component Usage:

Using a component is enabled by importing the component's class, by adding the component's selector in the suitable place in the template using the component, and by adding the component's class to the array of directives used by the host components. You may add some attribute to the same hosting element to send data to the component. We will later demonstrate it in the Input/Output section in this chapter.

Another use is with routing. Angular 2.0 routing is based on a path and a corresponding component. For that, as discussed in previous chapter, you may add a route object to the route array sent to the @Route decorator, see the home route below:

```
{path: '/home', compnent: HomeComponent}
```

In that case you won't need any use of the component selector, its template will be injected to the router-outlet element on the host component template.

Using injectable services

This is a component that wishes to use injectable service needs to receive it in one of its constructor's parameters. Having the *private* keyword before each injectable parameter makes a class member named the same as the parameter name and assigns the parameter to it.

Providers: Having the constructor parameters doesn't make sure that in instance will be sent to the constructor. To make sure in instance is created, we need to put the injectable service type in the providers array of the *ViewMetaData*.

Note: Unlike Angular 1.x services, the Injectable services are not necessarily singletons. Whenever a type of service is in a component's providers array, an instance of it is created upon the creation of the component's instance (i.e., it is not a singleton). It will be preferred that injectable services you wish to use as a single instance will be put in one provider's array in the application's main component.

The code example below demonstrates how to declare an injectable service and how to use it.

First, let's write an injectable service:

```
@Injectable()
export class MyService {
/*Service code goes here*/
}
```

As you can see above, this is simply an exported class decorated with the *Injectable* decorator.

Second, let's use it in a component:

First, we have to import the class, because we will always do this when we want our code to use code from another file:

```
import {MyService} from 'path/to/myService';
```

Second, we need to instantiate it. The Angular 2.0 framework will do that for us after we will put our service in a component's providers array. If the service is put in the provider's array in another component (for example, our main component), the service has to be imported there as well.

```
@Component({ /*some ViewMetaData members */
   providers: [ MyService],
   /*some other ViewMetaData members */ });
```

And, last but not least, we have to inject the service in our component's class constructor.

```
export class MyComponentClass {
 constructor(private _myService:MyService){
 }
}
```

By doing so we create a class member named *_myService* with the type of *MyService*, and the parameter sent to the constructor is assigned to it.

Please note that it creates a class member without you declaring it, No additional code is needed here.

Pipes

When binding data on our HTML template, we might want to present the data in a different form rather than just the data itself (meaning that Angular will use the `_toString` method on the presented object).

Therefore, we can run the data through a *pipe*.The syntax for doing so is to add a pipe sign (`|`) after the binding expression and use a pipe.

We might add some more argument to the pipe using the colon mark (`:`) before each argument we add.

We will need, of course, to add the pipe's type to our *pipes* array member of our component's *ViewMetaData* after importing it.

Built in pipes

Here are some built in pipes, which all may be imported from '@angular/common';

1. *UpperCasePipe* (name: ‘uppercase') returns an upper-case string
2. *LowerCasePipe* (name: ‘lowercase') returns a lowe-case string
3. *DatePipe* (name: ‘date') receives *pattern* as argument and returns a formatted date according to the given *pattern*, or to the browser’s default locale if no pattern has been given

Writing custom pipes

A pipe is also a class, however, this class implements the *PipeTransform* interface that has a method named *transform*, which is the one who does the formatting work.

To demonstrate, here is a simple pipe that reverses its given source:

```
import {Pipe, PipeTransform} from '@angular/core';

@Pipe({name: 'reverse'})
export class ReversePipe implements PipeTransform {

    transform(source:any) {
        if (!source || !source.length){
            return source;
        }
        let result = '';
        for (let i = 0; i < source.length; i++)
       {
          result+= s[s.length - i - 1];
       }
       return result;
    }
}
```

To use the pipe in our component we will import it, put its type in our pipes array on our *ViewMetaData* parameter sent to the *Component* decorator, and put its name wherever we need in our template.

```
/*Import the child component */

import {ReversePipe} from 'path/to/reversepipe';

@Component({selector: 'main-component',
template: `<h1>{{componentName}}</h1>
         <!-- and now in reverse -->
         <h2>{{componentName | reverse}}</h2>
         `


 pipes: [ReversePipe]
 })
 export class MainComponent{
     private componentName: string;

     constructor(){
      this.componentName = 'Main app component';
     }
 }
```

Directives

In case you wish to use another components or directives in your component, all types of components/directives you wish to use has to be written in the *directive* array member of the *ViewMetaData* object. After doing that, all you need to do is simply put the directive.

For that, you need to do the following:

1. Import the other component / directive type.
2. Put the type you've imported in the *directives* array member of the *ViewMetadata* object.
3. Use the directive / component as stated in its *selector*.

In the following example, let's create a simple component and use it on another one.

This is the child component we'll use:

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

@Component({selector: 'child-component',
 template: `<h2>This is an child component</h2>`
 })
 export class Childcomponent{
 }
```

Now, let's use it in another component:

```
/*Import the child component */
import {Childomponent} from 'path/to/childComponent';

@Component({selector: 'main-component',
template: `<h1>Main app Component</h1>
         <!-- using the child component here -->
         <inner-component></inner-component>
         `
         /*Putting the child component type in our directives*/
 directives: [Childomponent]
 })
 export class MainComponent{
 }
```

Note that forgetting to put the type in your directives array member won't produce any error messages on your console, but won't render your component.

Inter-component communication

You can pass data to and from a component you're using. In order for our inner component to receive data to one of its class members, this class member is to be decorated by the *Input* decorator.

To demonstrate it, let's alter our inner component a little bit.

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

@Component({selector: 'child-component',
 template: `<h2>{{componentName}}</h2>`
 })
 export class ChildComponent{
 @Input() componentName:string;
 }
```

We have added a property named *componentName* to the inner component preceded with the *Input* decorator, imported also from the @angular/core.

To show it, we've bounded it to the inner component template.

Now, all we need to do, is bind the *componentName* to a member in the main component.

```
/*Import the child component */
import {ChildComponent} from 'path/to/childComponent';

@Component({selector: 'main-component',
template: `<h1>Main app Component</h1>
         <!-- using the inner component here -->
         <child-component [componentName]="childComponentName"></child-component>
         `
         /*Putting the child component type in our directives*/
 directives: [ChildComponent]
 })
 export class MainComponent{
   childComponentName: string;

   constructor(){
    this.childComponentName = "child component";
   }
 }
```

Using Events

After discussing the method of the parent component way to send data to its child components, we need to pass data in the opposite direction. Because the child component doesn't know which component is going to contain it, it cannot bind to any member of its containing component. Therefore, the only way a child component can communicate with its parent is through *Events*.

For that matter we will use another decorator named *Output* and a class named *EventEmitter* for launching the event.

The *EventEmitter* is a generic type where the type parameter is the type of the argument sent with the event.

EventEmitter class

We've previously discussed how to use this class, so now we will elaborate a little more about that class.

This class extends the *Subject* class from rx.js, which extends the *Observable* class of the same module. You may read more about that in Chapter 10, Change Detection.

Whenever you use the event on your HTML template you actually subscribe on the event, and your handler is added to its *observers* array. All observers are called when using the event's *emit()* method.

Built in Events

As discussed on the *Data Binding* section, the event is bound by round brackets. There are many built in events, however, on our component we are going to create a custom event.

We will firs add a textbox to the child component and bind it to another property.

Second, we will create an EventEmitter typed member that will invoke an event whenever the newly created property is changed.

Third, we will bind the *change* event of the input element to a method on the child component in that method, where the event we've just created will be launched.

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

@Component({selector: 'child-component',
 template: `<h2>{{componentName}}</h2>
       <input type="text" [(ngModel)]="comopnentValue" (change)="componentValueChanged()"/>
     `

 })
```

```
export class ChildComponent{
@Input() componentName:string;
@Output() onComponentValueChange:EventEmitter<any> = new EventEmitter<any>();

componentValue: string;

componentValueChanged(){
    this.onComponentValueChange(this.componentValue);
}

}
```

After doing that, on our main component all we have to do is to bind to the newly created custom event. We will add another string member to our main component class, named *childComponentValue* and bind it to a span's inner text on our main component, so the changes will be shown on the screen.

```
/*Import the child component */
import {ChildComponent} from 'path/to/childComponent';

@Component({selector: 'main-component',
template: `<h1>Main app Component</h1>
        <!-- using the inner component here -->
        <child-component [componentName]="childComponentName " (onComponentValueChange)="childComponentVal
        </child-component>
        <!-- showing child component value on the main component-->
        <span>{{childComponentValue}}</span>
        `
        /*Putting the child component type in our directives*/
 directives: [Childomponent]
 })
 export class MainComponent{
   childComponentName: string;
   childComponentValue: string;

   constructor(){
    this.childComponentName = "child component";
   }

   childComponentValueChange(componentValue){
     this.childComponentValue = componentValue;

   }
 }
```

Now we have created a custom event that will be fired from the child component to its parent whenever the componentValue is changed on the child component.

Component lifecycle

A component has its own lifecycle events so we can hook them and right the code that will run whenever these events occur. Angular manages the lifecycle itself so we can have our custom code.

For hooking an event our component class should implement the corresponding interface. Angular core has a corresponding interface for each event. That interface has a method with the *ng* prefix and the event name (for example *ngOnInit* , *ngOnDestroy* etc.).

Here's a list of lifecycle hooks

1. *OnInit* occurs after the component is being initialized.
2. *OnDestroy* occurs **before** a component is destroyed.
3. *OnChanges* occurs on *every* change that occurs in the component data. This hook called when an input value is changed.
4. *DoCheck* extends the *Onchanges* handler, and when *DoCkeck* is implemented, it overrides the default change detection algorithm.
5. *AfterContentInit* occurs after content init.

In order to hook one of these lifecycle events you should implement its corresponding interface (interface names are listed above). Each interface has a method that will handle the lifecycle event.

Summary

Components hold the UI structure and behavior of our application. They can be built on the *HTML5 web component* wherever the browser supports them. A component can contain another component and bind data to it by its @Input decorated members. The inner component can send messages to its container component by using custom events. Cutom events are written with `EventEmitter<type>` type members decorated with the `@output` decorator.

Chapter 6. Directives

In this chapter we will be taking a look at directives and the role they play in an Angular 2 application. There are three types of directives in Angular 2. We have already covered one of them in the previous chapter. Components are in fact considered directives. To quote the Angular team:

> A Component is really a directive with a template.

They are the most common type of directives. Now we will examine the remaining two types of directives:

- Attribute directives
- Structural directives

In common applications we don’t write a lot of attribute or structural directives. This is primarily because of how verbose Angular 2’s template binding is, and the few built in directives we get really cover a vast array of tasks that we encounter.

Attribute Directives

Attribute directives are directives that can change the appearance or behavior of an element. We have encountered two built in attribute directives: `NgStyle` and `NgClass`. A common scenario for using an attribute directive is when we want to apply styling to an element on a particular event.

note: In some cases we can use [css pseudo-classes] such as `:hover` for example. We would only write this kind of an attribute directive if our particular behavior can't be achieved with only CSS, for example `onclick`, `ondblclick` and so on.

Let's write an attribute directive that will add or remove a border to the **host** representing it being selected on double click.

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

@Directive({
    selector: '[edge-select]'
})
export class SelectDirective {
    constructor(private _el: ElementRef, private _renderer: Renderer) {
        _renderer.setElementStyle(_el.nativeElement, 'border', '2px solid #00897b');
    }
}
```

As always, first we import everything required. In this case its the `Directive` decorator, which we apply to our class, and the `ElementRef` service through which we can directly access the DOM element using its `nativeElement` property, as well as the `Renderer` which we use to manipulate the style of the element.

note: The `Renderer` has many other useful methods. You can find all of them here.

In the `@Directive` decorator we specify the selector property in square brackets. This determines the way our directive is attached to an HTML element.

Before we can use our directive we need to add it to the declarations array in our `AppModule` just like we do with components then we can now attach our directive to any element in a component like this:

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

@Component({
    selector: 'cool-directives',
    template: `
        <span edge-select>I have a green border!</span>
    `
})

export class CoolDirectives { }
```

Inside the `SelectDirective` constructor, we set the border to be `2px solid #00897b`. This means that

the border will be applied to the **host** element (the `<span></span>` element in this case) in our component as soon as it is rendered.

Now let's add the double click functionality:

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

@Directive({
    selector: '[edge-select]'
})
export class SelectDirective {
    constructor(private _el: ElementRef, private _renderer: Renderer) {}

    private selected: boolean = false;

    @HostListener('dblclick') onDoubleClick() {
        if (!this.selected) {
            this._renderer.setElementStyle(this._el.nativeElement, 'border', '2px solid #00897b');
            this.selected = true;
        }

        else {
            this._renderer.setElementStyle(this._el.nativeElement, 'border', 'null');
            this.selected = false;
        }
    }
}
```

We add the `HostListener` decorator to the `onDoubleClick()` method providing the event we want to bind to.

```
@HostListener('dblclick')
```

note: We call the selector `edge-select` in favor of just `select` because we don't want our directive to overlap with a native HTML attribute or some external directive we might import.

The `selected` private property keeps track of whether the element already has the border or not.

We can improve upon our directive by adding a bindable input property that will allow us to specify different border colors for different instances of the directive. First we add `Input` to our other imports:

```
import {Directive, ElementRef, Renderer, HostListener, Input} from '@angular/core'
```

Then we use it in our class like this:

```
@Input('edge-select') borderColor: string = '#00897b';

@HostListener('dblclick') onDoubleClick() {
    if (!this.selected) {
        this._renderer.setElementStyle(this._el.nativeElement, 'border', `2px solid ${this.borderColor}`);
        this.selected = true;
    }

    else {
        this._renderer.setElementStyle(this._el.nativeElement, 'border', 'null');
        this.selected = false;
```

```
    }
}
```

note: We aren't limited to only one input property when working with attribute directives. We can have as many as our directive requires.

Now that we have an input property, we need to change the implementation of the directive in our component:

```
import {Component} from '@angular/core'
import {SelectDirective} from './select.directive';

@Component({
    selector: 'cool-directives',
    template: `
        <!-- border-color: #00897b -->
        <span [edge-select]>I have a green border!</span>
        <div [edge-select]="'red'">My border is red!</div>
    `
})

export class CoolDirectives { }
```

We replace `edge-select` with `[edge-select]` and we can also pass in a value that would replace the `borderColor` properties value.

note: We could have created a component with the same functionality using Angular 2's template bindings. However, the advantage of building an attribute directive is that we can now apply it to any HTML element.

Replicating NgClass

`NgClass` is a great example of an attribute directive. We have covered it in Chapter 5's Template Syntax sub-chapter. You can find the source code for `NgClass` Here. Now, let's create our own implementation of the directive:

```
import {Directive, OnInit, ElementRef, Input, Renderer} from '@angular/core'

@Directive({
    selector: '[edge-classes]'
})
export class ClassesDirective implements OnInit {
    constructor(
        private _el: ElementRef,
        private _renderer: Renderer
    ) {};

    @Input('edge-classes') classArray: string[];

    ngOnInit(): void {
        this.classArray.forEach(a => this._renderer.setElementClass(this._el.nativeElement, a, true))
    }
}
```

As you can see, our implementation is nowhere near as complex as the original. We can only receive a string array of classes to add to the host element and we don't really handle cleanup. Nonetheless, it serves as a good usage example for attribute directives.

In conclusion, attribute directives are great when you need to add styling and functionality to a variety of different elements. The directive you create can be attached to a `<div></div>` or `<span></span>` or any other HTML element. On the other hand, if the functionality you plan on implementing always has the same template, it might be better to create a component. It's more verbose, and you can pretty much achieve the same result using Angular 2's template binding, inputs and outputs.

Structural Directives

Whereas **attribute directives** change the look and behavior of elements, **structural directives** change the DOM layout by adding and removing elements. They allow us to dynamically change the structure of the DOM based on our application's state.

For example, the `NgIf` structural directive adds or removes an element from the DOM based on whether its expression evaluates to `true`.

We have worked with three built in structural directives: `NgIf`, `NgFor` and `NgSwitch`. You recognize a structural directive by the `*` prefix, although there is an alternate `<template></template>` syntax. Angular 2 expands the `*` in to the `<template></template>` syntax (you can read all about it here) so there is most often no benefit of writing one over the other, and the `*` syntax is faster to use, so we will stick with this method.

Now let's create a simple structural directive:

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

@Directive({ selector: '[edgeTimes]' })
export class TimesDirective {
    constructor(
        private _viewContainer: ViewContainerRef,
        private _templateRef: TemplateRef<Object>
    ) {}

    @Input() set edgeTimes(times: number) {
        this._viewContainer.clear();
        for (let i = 0; i < times; i++) this._viewContainer.createEmbeddedView(this._templateRef);
    }
}
```

This directive takes a number as input and then copies the template that number of times. We declare a structural directive in much the same way we would an attribute directive. The difference is that in structural directives we import the `TemplateRef` and the `ViewContainerRef`.

We use the `TemplateRef` to get access to the template that the directive governs and the `ViewContainerRef` provides us with various template-rendering methods. We used the `createEmbeddedView()` method, which creates an instance of the provided `templateRef`.

`edgeTimes` gets called when there is an attempt to set the property, which happens on initiation of the directive and then again every time the input changes.

note: You can read more about setter here.

Before we create an instance of the template we first clean the entire container with the `clear()` method of the `ViewContainerRef`. If we don't do this whenever the input value changes, we would create that many instances of the template without removing the old ones.

note: If you are coding along don't forget to add every directive in to the `AppModule`.

This is how we use the directive in a component:

```
import {Component} from '@angular/core'
import {TimesDirective} from './times.directive';

@Component({
    selector: 'cool-directives',
    template: `
        <div *edgeTimes="times">
            <span>This gets copied {{times}}</span>
        </div>
    `
})

export class CoolDirectives {
    public times: number = 5;
}
```

The `TimesDirective` is pretty much the simplest structural directive you would write and you could achieve this behavior without building a directive. However, it serves well as an introduction, so now let's write a structural directive that's a bit more complex.

We will write a directive that loops through the provided object array similar to what `NgFor` does, but we will add them to the DOM one by one in 2-second intervals.

```
import {
    Directive,
    Input,
    IterableDiffers,
    IterableDiffer,
    ChangeDetectorRef,
    EmbeddedViewRef,
    DoCheck,
    TemplateRef,
    ViewContainerRef
} from '@angular/core';

@Directive({ selector: '[edgeOneByOne]' })
export class OneByOneDirective implements DoCheck {
    constructor(
        private _viewContainer: ViewContainerRef,
        private _templateRef: TemplateRef<Object>,
        private _iterableDiffers: IterableDiffers,
        private _cdr: ChangeDetectorRef
    ) {}

    private items: any;
    private _differ: IterableDiffer;

    @Input() set edgeOneByOneOf(value: any) {
        this.items = value;
        if (value && !this._differ) this._differ = this._iterableDiffers.find(value).create(this._cdr);
    }

    ngDoCheck() {

        let changesArray: any = [];

        if (this._differ) {
            let changes = this._differ.diff(this.items);
            if (changes) changes.forEachAddedItem((change) => changesArray.push(change))
        }
```

```
        let index = 0;

        let timer = setInterval(() => {
            if (index < changesArray.length) {
                this.createItem(changesArray[index]);
                index++;
            } else {
                clearInterval(timer);
            }

        }, 2000)
    }

    createItem(change) {
        let view = <EmbeddedViewRef<OneByOneSingle>>this._viewContainer.createEmbeddedView(this._templateRe
        view.context.$implicit = change.item;
    }
}

export class OneByOneSingle {
    constructor(public $implicit: any) {}
}
```

As you can see, structural directives can get a bit complex. Let's start by going through all of the new imports.

- IterableDiffers - A repository of different iterable diffing strategies. We use the `find()` method in our setter to determine what the type of the differ that was provided is.
- IterableDiffer - A single iteratableDiffer its used for tracking changes over time to an iterable.
- ChangeDetectorRef - Our directive's change detector. The `IteratableDiffers` `create()` method requires it as an argument.
- EmbeddedViewRef - Represents an Angular 2 View.

Just defining the imports doesn't really tell us that much. So let's go through how we used them.

The setter is named `edgeOneByOneOf`, and this precise wording is required. It allows us to use our directive like this:

```
<div *edgeOneByOne="let i of fu">
    <span>{{i}}</span>
</div>
```

In the setter we set the received value to the `items` property, and we create the differ if it isn't created already.

Now we get to the `ngDoCheck()` method. Implementing this method tells Angular 2 that we are going to use our own change detection.

```
let changes = this._differ.diff(this.items);
```

The `changes` variable will hold all of the changes to the original object. Now we add each

item added to the object to our local `changesArray`.

note: If you take a look at the `NgFor` implementation here, you will notice that we only do `forEachAddedItem` checks and don't really handle removing or moving. This is fine for our purposes, but this means that we don't handle changes to the input value after the directive was initiated.

```
let index = 0;

let timer = setInterval(() => {
    if (index < changesArray.length) {
        this.createItem(changesArray[index]);
        index++;
    } else {
        clearInterval(timer);
    }

}, 2000)
```

Here we add changes one by one in two second intervals by calling the `createItem()` method.

```
createItem(change) {
    let view = <EmbeddedViewRef<OneByOneSingle>>this._viewContainer.createEmbeddedView(this._templateRef, n
    view.context.$implicit = change.item;
}
```

The method creates the view and binds the value of the collection item using a special local variable called `$implicit`.

We can now use the directive in a component like this:

```
import {Component} from '@angular/core'
import {OneByOneDirective} from "./oneByOne.directive";

@Component({
    selector: 'app',
    template: `
        <div *edgeOneByOne="let i of fu">
            <span>{{i}}</span>
        </div>

    `
})
export class SomeComponent {
    public fu = ['fu', 'bar', 'something']
}
```

Summary

In conclusion, even though in most applications we will only write a few attribute directives, and even less structural directives, they are still a great tool Angular 2 provides us with. They allow us to manipulate the way data is displayed in the DOM, and provide us with accessibility methods for handling changes in our data.

Chapter 7. Services

We had a brief introduction to services in Chapter 3. We explained that in Angular 2, application services should handle **sending and receiving data**, allowing components to focus on controlling the view.

In Chapter 3 we also created the `UserService` to handle fetching a list of users for the `UserBlockComponent`. However, the `UserService` will only return a "hard coded" list of users, not really communicating with the server to get the data.

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

@Injectable()
export class UserService {

    private users = [
        {id: 1, username: 'filip.lauc93@gmail.com', status: 'online'},
        {id: 2, username: 'laco0416@gmail.com', status: 'offline'},
        {id: 3, username: 'mgualtieri7@gmail.com', status: 'online'},
        {id: 4, username: 'ran.wahle@gmail.com', status: 'online'},
        {id: 5, username: 'wojtek.kwiatek@gmail.com', status: 'offline'}
    ]

    get() {
        return this.users;
    }
}
```

In this chapter we will expand upon our knowledge of services while writing a service that communicates with the server through Angular 2's HTTP client service.

We will also handle storing and retrieving data from the browser's `localStorage` with a service.

In addition, we will learn how to import external libraries in Angular 2 applications and create a service that uses `Socket.io`.

HTTP and Observables

The HTTP client service heavily uses RxJs in its implementation. `RxJs` is a broad topic, so for now we will just stick to the implementation, and explore it in Chapter 11.

To use observables, you need to import the `RxJS` library. Even though the library is fairly large and you won't use all of it, to keep us focused on writing our service we import the entire library in our `AppComponent`.

location: `public/app/app.component.ts`

```
import 'rxjs/Rx'
// ...
```

note: To minimize load time in a production application, you should only import the parts of the library that you need. You can find an example of this in our sample application here

A common practice that we like to follow, is to take a more general approach and create a single service that will handle all of the HTTP requests. Instead of the `UserService` that we created earlier, we will write a service called the `ApiService` that will be responsible for all HTTP communication in our application. This is done because HTTP requests generally don't vary that much. This also allows us to create general rules for our HTTP calls, such as appending an authorization header, for example.

Before we can use Angular 2's HTTP client service, we need to import the `HttpModule` in the applications `AppModule`.

location: `public/app/app.module.ts`

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

@NgModule({
    imports: [
        BrowserModule,
        HttpModule,
        // ...
    ],
    // ...
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Now we can safely create the `ApiService`:

__location: `public/app/common/services/api.service.ts`

```
import {Injectable} from '@angular/core'
import {Http, Headers, RequestMethod, Request} from '@angular/http'
import {Observable} from 'rxjs/Observable';
```

```
/*
    This service handles all HTTP requests to the server. Handling all
    requests through one service provides a simple way to append headers to all requests
    with out extending angular 2's Http class
 */
@Injectable()
export class ApiService {
    constructor (private _http: Http) {}

    send(url: string, type: "Get" | "Post" | "Put" | "Update" | "Delete", item?: any) {

        // Define the options for the request
        let options = {
            method: RequestMethod[type],
            url: url,
            body: '',
            headers: new Headers({
                'Content-Type': 'application/json'
            })
        };

        // If the passed item is a string use it
        // Otherwise json stringify it
        if (item && type !== 'Get') options.body = typeof item === 'string' ? item : JSON.stringify(item);

        return this._http.request(new Request(options))
            .map(res => res.json().data)
            .catch(err => Observable.throw(err));
    }
}
```

note: You can find the documentation as well as a list of API URL's for our server here. The server only handles a few API requests, because the rest of the communication with the client happens through web sockets.

We import everything we need from `@angular/http` and we attach the `@Injectable()` decorator to our service. This allows us to inject the service in to other components and services.

The `send` method is the most important part here. It accepts a url, the type of the request and an optional item if the request isn't a get request. The type specifies the HTTP request we want to make.

```
return this._http.request(new Request(options))
    .map(res => res.json())
    .catch(this.logError);
```

The `send` method returns the observable `_http.request` with our defined options.

Before we can replace the `UserService` in our `UserBlockComponent`, we need to register the provider for our `ApiService`. The `ApiService` will handle all of the HTTP requests our application will need (even thought it only handles getting users for now), we don't want to have to register the provider in the metadata of every component where we want to consume it. We also don't need multiple instances of the service. Registering the provider at the root level will make the service a singleton across the entire application.

We add the `api.service` to the list of providers in our `AppModule`:

location: `public/app/app.module.ts`

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {HttpModule} from '@angular/http';
import {ApiService} from 'common/services/api.service';

@NgModule({
    imports: [
        BrowserModule,
        HttpModule,
        // ...
    ],
    providers: [
        // ...
        ApiService
    ]
    // ...
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Once we have everything properly registered we can go ahead and replace the `UserService`:

location: `public/app/common/components/user-block.component.ts`

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

// Import the ApiService in place of the 'UserService'
import {ApiService} from '../services/api.service'

@Component({
    selector: 'edge-userBlock',
    // The template...
})
export class UserBlockComponent implements OnInit {
    constructor(
        // Define the service
        private _api: ApiService
    ) {}

    public users;

    ngOnInit() {
        // Consuming the observable returned by the send method
        this._api.send('http://localhost:5000/api/users', 'Get').subscribe(
            res => this.users = res.data,
            err => console.log(err)
        )
    }
}
```

Just like we did with the `UserService`, we use our `ApiService` in the `ngOnInit` lifecycle hook. The `send` method returns the `_http.request`, which is an observable instead of calling `then()` and `catch()` like we would with a promise. We use the `subscribe()` method.

Communicating with local storage and event emitters in services

One of the reasons why we built the `ApiService` to handle all HTTP requests for the application is to easily apply rules to all of the request. For instance, services often have protected routes that require us to provide a valid token.

A common practice is to get a valid token for the API using the intended API call and then store the token in to the clients `localStorage`. Then we retrieve the token in our service and send it in the header when sending requests to protected routes.

First, let's create the service that will handle accessing and storing data in to `localStorage`. We will call it `UserStoreService`, because in our case, we only store user information and the token in to `localStorage`.

location: `public/app/common/services/user-store.service.ts`

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

@Injectable()
export class UserStoreService {
    constructor() {}

    public emitter: Subject<any> = new BehaviorSubject(null);
    private _appName = 'edge-user';

    get() {
        let temp = localStorage.getItem(this._appName);
        if (temp) return JSON.parse(temp);
        else return null;
    }

    set(user?) {
        // If a user object was received stringify it and save it
        // in local storage
        // otherwise remove the user from localStorage
        if (user) localStorage.setItem(this._appName, JSON.stringify(user));
        else localStorage.removeItem(this._appName);
        this.emitter.next(user);
    }

}
```

This is a pretty straightforward service. We call native `localStorage` methods to get and set the 'edge-user' in our `get()` and `set()` methods.

A small addition is the `Subject` and `BehaviorSubject` from `RxJs` that we use to broadcast changes of the value in `localStorage`.

note: You can read more about `Subject` here and the `BehaviorSubject` here.

For example, this is what we would do in a component to listen to changes:

```
import {UserStoreService, OnInit} from './common/services/userStore.service'

@Component({
    selector: 'some-component',
    template: '<p>{{user}}</p>'
})

export class SomeComponent implements OnInit {
    constructor(
        private _userStore: UserStoreService
    ) {}

    public user: any;
    private _userChangeListener: any;

    ngOnInit(): void {

        // Subscribe to the emitter so we can
        // update the user in our template when the value in localStorage changes
        this._userChangeListener = this._userStore.emitter.subscribe(item => this.user = item)
    }

    // Its a best practice to clean up the listener
    ngOnDestroy() {
        this._userChangeListener.unsubscribe();
    }
}
```

Just like with the `ApiService` before, we can use the `UserStoreService` we need to add it to our list of providers in the `AppModule`.

location: `public/app/app.module.ts`

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {HttpModule} from '@angular/http';
import {ApiService} from 'common/services/api.service';
import {UserStoreService} from 'common/services/user-store.service';

@NgModule({
    imports: [
        BrowserModule,
        HttpModule,
        // ...
    ],
    providers: [
        // ...
        ApiService,
        UserStoreService
    ]
    // ...
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Now we can use it in the `ApiService` and create the rule to apply the token from `localStorage` to desired requests:

__location: `public/app/common/services/api.service.ts`

```
import {Injectable} from '@angular/core'
import {Http, Headers, RequestMethod, Request} from '@angular/http'
import {Observable} from 'rxjs/Observable';
import {UserStoreService} from 'common/services/user-store.service';
```

```
/*
    This service handles all HTTP requests to the server. Handling all
    requests through one service provides a simple way to append headers to all requests
    with out extending angular 2's Http class
 */
@Injectable()
export class ApiService {
    constructor (private _http: Http) {}

    // If we want the token applied to our request we pass true for the authHeader param
    send(url: string, type: "Get" | "Post" | "Put" | "Update" | "Delete", item?: any, authHeader?: boolear

        // Define the options for the request
        let options = {
            method: RequestMethod[type],
            url: url,
            body: '',
            headers: new Headers({
                'Content-Type': 'application/json'
            })
        };

        // If the passed item is a string use it
        // Otherwise json stringify it
        if (item && type !== 'Get') options.body = typeof item === 'string' ? item : JSON.stringify(item);

        // We add the token header from the UserStoreService
         if (authHeader) options.headers.append('Authorization', 'Bearer ' + this._userStoreService.getUser(

        return this._http.request(new Request(options))
            .map(res => res.json().data)
            .catch(err => Observable.throw(err));
    }
}
```

We added the logic for appending the header when required so now we can call protected routes by adding `aouthHeader` param.

note: If you tried the `htpp:/localhost:5000/api/users` request before, you noticed that it works even though the authorization header wasn't applied. This is because we left the route unprotected for the purposes of the sample application.

Importing external libraries

If you have looked at the documentation for our server here and the source code of the sample application here, you've noticed that we only provide a few routes in the API. This is because once the user is logged in and gets all of the data to load the initial view, all of the server - client communication switches to web sockets.

We won't be looking into web sockets, because it is outside the scope of the book, but we will demonstrate how to import external libraries. To do this we will create a service that uses socket.io. This is a part of the service our sample application uses for socket communication:

location: `public/app/common/services/socket-control.service.ts`

```
import {Injectable} from '@angular/core'
import {Router} from '@angular/router-deprecated'
import {UserStoreService} from './userStore.service'
import {socketValues} from '../config/app.values'
import {DataService} from './data.service'


@Injectable()
export class SocketControlService {
    constructor(
        private _router: Router,
        private _userStore: UserStoreService,
        private _data: DataService
    ) {
        // Open connection
        this.socket = io.connect(this.sv.url);
    };

    public socket;

    // ...

}
```

note: You can find the code for the entire service here.

We use this line to import socket.io: `import * as io from 'socket.io'`. We can do this because there are `.d.ts` files available for socket.io through typings or types. To install them, we run the following command in our console:

```
npm install --save @types/socket.io
```

note: You can always find out if definition files are available for a library by running `typings search [library name]`. The number of available library definition files is expanding constantly so its always worth a check.

In our `system.config.js` file we tell systemjs where to find socket.io like this:

```
    var map = {
        'socket.io': 'node_modules/socket.io-client/socket.io.js'
        // ...
```

```
};
```

Alternatively we can import the library in the old-fashioned way using the `script` element in the `index.html`.

```
<script src="../node_modules/socket.io-client/socket.io.js"></script>
```

If we use this approach we need to declare io in files where we use it, to prevent typescript and our editor from throwing errors.

```
declare const io: any;
```

Summary

Angular 2 services are an essential part in maintaining data, as well as how data is handled and accessed through server communication. While components may handle the view, we also need to be able to store this data in our local storage or IndexDb as well as perform other CRUD operations on it.

Communication to the server is achieved through services that utilize Angular 2's HTTP client, but we also use services for a variety of other tasks in our application like component communication and as interfaces for native methods.

All this makes services an important part of every Angular 2 application, learning when to use them and how to use them properly is a big step in learning Angular 2.

Chapter 8. Pipes

Pipe is a transformer to display data. You can use pipes for string formatting, data converting or other things. Pipes let you separate *view-specific logic* from components or services.

In Angular 2, you can use pipes with **pipe operator (|)** in the component template. Let's look at the following simple example:

```
@Component({
  selector: 'my-cmp',
  template: `<h1>{{ title | uppercase }}</h1>`,
})
export class MyComponent {
  title = 'Pipes';
}
```

The component will be rendered as `<h1>PIPES</h1>`. In this case, `title` property is transformed by the `uppercase` pipe. All pipes take an input value from the left-side of the operator (`|`), and some pipes take additional parameters from the right-side of the pipe's name as `pipe:param1:param2` like the following code:

```
@Component({
  selector: 'my-cmp',
  template: `{{ text | replace:'World':'Angular' }}`, // -> Hello Angular
})
export class MyComponent {
  text = 'Hello World';
}
```

A reason of the name, *pipe*, is that supports chained transforming like data streams. You can use multiple pipes at the same input.

```
@Component({
  selector: 'my-cmp',
  template: `{{ text | replace:'World':'Angular' | lowercase }}`, // -> hello angular
})
export class MyComponent {
  text = 'Hello World';
}
```

Built-in Pipes

By default, many useful built-in pipes are provided and you can use them without explicit importing. Ok, let’s figure them out by its kind.

String -> String

First, let's figure out about pipes transforming string to string.

`UpperCasePipe` / `LowerCasePipe`

`UpperCasePipe` transforms a string to an upper-case string. It's named `uppercase`.

```
@Component({
  selector: 'my-cmp',
  template: `{{ 'Angular 2' | uppercase }}` // -> 'ANGULAR 2'
})
export class MyComponent {
}
```

Similarly, `LowerCasePipe` makes a string lower-cased. It's named `lowercase`.

```
@Component({
  selector: 'my-cmp',
  template: `{{ 'Angular 2' | lowercase }}` // -> 'angular 2'
})
export class MyComponent {
}
```

Number -> String

Some pipes can convert a number value to a string and format it.

`DecimalPipe`

`DecimalPipe` formats a number as a decimal string. It is named `number` and it has an optinal parameter, `digitInfo`. `digitInfo` is a string that has three segments like the following:

```
{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}
```

- `minIntegerDigits` is the minimum number of integer digits to use. It defaults to 1.
- `minFractionDigits` is the minimum number of digits after fraction. It defaults to 0.
- `maxFractionDigits` is the maximum number of digits after fraction. It defaults to 3.

In other words, default `digitInfo` is `1.0-3`.

See the following example:

```
@Component({
  selector: 'my-cmp',
  template: `
    {{ 12.3456 | number }},
    {{ 12.3456 | number:'3.0-2' }},
    {{ 12 | number:'3.2' }}
  `
  // -> 12.346, 012.35, 012.00
})
export class MyComponent {
}
```

`PercentPipe`

`PercentPipe` is similar to `DecimalPipe`. The difference is that `PercentPipe` shows a number in percentage.

```
@Component({
  selector: 'my-cmp',
  template: `
    {{ 0.1234 | percent }},
    {{ 0.1234 | percent:'3.0-2' }},
    {{ 0.12 | percent:'3.2' }}
  `
  // -> 12.34%, 012.34%, 012.00%
})
export class MyComponent {
}
```

`CurrencyPipe`

`CurrencyPipe` is used for displaying a number as a price. Basically it's the same to `DecimalPipe`, but `CurrencyPipe` takes two additional parameters: `currencyCode` and `symbolDisplay`.

`currencyCode` is the ISO 4217 currency code, such as "USD" for the US dollar and "EUR" for

the euro.

`symbolDisplay` is an optional parameter to use the currency symbol (e.g. $) or the currency code (e.g. USD) in the output. It defaults to `false`.

```
@Component({
  selector: 'my-cmp',
  template: `
    {{ 12.3456 | currency:'USD' }},
    {{ 12.3456 | currency:'USD':true }},
    {{ 12.3456 | currency:'USD':true:'.0-2' }}
  `
  // -> USD12.346, $12.346, $12.35
})
export class MyComponent {
}
```

Object -> String

The following pipes can transform an object to a string.

`JsonPipe`

`JsonPipe` converts an object to a string by `JSON.stringify`. This pipe is useful for development.

```
@Component({
  selector: 'my-cmp',
  template: `
    {{ data | json }}
  `
  // -> { "name": "foo", "value": "bar" }
})
export class MyComponent {
  data = { name: 'foo', value: 'bar' };
}
```

`DatePipe`

`DatePipe` transforms an input date value to formatted text. This pipe accepted some types as an input. The first type is a `Date` type. Look at the following code:

```
@Component({
  selector: 'my-cmp',
  template: `
    {{ targetDate | date:'yMMMd' }}
  `
  // -> Sep 3, 2015
})
export class MyComponent {
  targetDate = new Date(2015, 8, 3);
}
```

`DatePipe` has an optional parameter, `format`, which is used for formatting the date. The following symbols can be used in the parameter:

- `G`/`GGGG`: era symbols corresponding to `AD`/`Anno Domini`
- `y`/`yy`: year symbols corresponding to `2015`/`15`
- `M`/`MM`/`MMM`/`MMMM`: month symbols corresponding to `9`/`09`/`Sep`/`September`
- `d`/`dd`: day symbols corresponding to `3`/`03`
- `EEE`/`EEEE`: weekday symbols corresponding to `Sun`/`Sunday`
- `h`/`hh`: 12-hour symbols corresponding to `01`/`01 PM`
- `H`/`HH`: 24-hour symbols corresponding to `13`/`13`
- `m`/`mm`: minute symbols corresponding to `5`/`05`
- `s`/`ss`: second symbols corresponding to `5`/`05`
- `z`/`Z`: minute symbols corresponding to `Pacific Standard Time`/`GMT-8:00`

In addition, there are some predefined formats:

- `medium`: equivalent to `yMMMdhms`
- `short`: equivalent to `yMdhm`
- `fullDate`: equivalent to `yMMMMEEEEd`
- `longDate`: equivalent to `yMMMMd`
- `mediumDate`: equivalent to `yMMMd`
- `shortDate`: equivalent to `yMd`
- `mediumTime`: equivalent to `hms`
- `shortTime`: equivalent to `hm`

An important thing is that **the formatting depends on the browser-locale**. This pipe uses the Internationalization API. Therefore, it isn't usable in some browsers.

Misc

In addition to that, there are some useful pipes. Let's look at those.

`SlicePipe`

`SlicePipe` creates a new list or string from an input one. It's named `slice` and takes two parameters: `start` and `end`.

```
@Component({
  selector: 'my-cmp',
  template: `
    {{ 'abcdef' | slice:2 }},
    {{ 'abcdef' | slice:2:4 }}
  `
  // -> cdef, cd
})
export class MyComponent {
}
```

`SlicePipe` can accept types of only `Array` or `string`.

`AsyncPipe`

`AsyncPipe` is a special pipe. Its named `async`, and it can transform an **async** object to a **sync** value. The async object is an `Observable` or a `Promise`. When the async object emits new value, `AsyncPipe` updates views with the value.

Let's see the following example:

```
@Component({
  selector: 'my-cmp',
  template: `
    {{ asyncData | async }}
  `
})
export class MyComponent {
  asyncData = new Promise(resolve => {
    setTimeout(() => resolve('Hello'), 1000);
  });
}
```

`asyncData` is a promise object, which will emit a value after 1000ms. And that view is empty before that time.

Also you can use the pipe with `Observable` and a view will be updated at each time when the new value is emmitted.

```
@Component({
  selector: 'my-cmp',
  template: `
    {{ asyncData | async }}
  `
})
export class MyComponent {
  asyncData = Observable.interval(1000)
```

```
    .map(i => `Count: ${i}`);
}
```

I18nPluralPipe

`I18nPluralPipe` and the following `I18nSelectPipe` are helpful pipes for *internationalization*. `I18nPluralPipe` can map a number input and a string output. For example, the following is a simple counter component:

```
@Component({
  selector: 'my-cmp',
  template: `
    <button (click)="itemCount=itemCount+1">Add item</button>
    {{ itemCount | i18nPlural:amountMapping }}
  `
})
export class MyComponent {
  itemCount = 0;
  amountMapping = {
    '=0': 'No items',
    '=1': 'One item',
    'other': '# items'
  };
}
```

`I18nPluralPipe` takes a mapping object as its parameter to pluralize. If the key is `'=0'`, that rule will be used when the input is `0`. A special key, `'other'`, is used as the default rule and you can interpolate the **actual value** with the `'#'` sign. In the above case, the rules look like the following:

input	output
0	'No items'
1	'One item'
2	'2 items'
x	'x items'

I18nSelectPipe

`I18nSelectPipe` is almost the same as `I18nPluralPipe`. The difference is that pipe maps a string to a string. See the following example:

```
@Component({
  selector: 'my-cmp',
  template: `
    <select [(ngModel)]="gender">
      <option value="" selected>select gender</option>
      <option value="male">male</option>
      <option value="female">female</option>
    </select>
    {{ gender | i18nSelect:inviteMap }}
  `
})
export class MyComponent {
  gender = '';
  inviteMap = {
```

```
    'male': 'Invite her.',
    'female': 'Invite him.',
    'other': 'Invite them.'
  }
}
```

`I18nSelectPipe` takes an object for mapping. The mapping rule is similar to the JavaScript `switch` statement. You can use this pipe to convert a *model-side string* to a *view-side string*.

Custom Pipes

Angular 2 supports creating custom pipes and you can use this easily. Using custom pipes, you can boost speed for developing your awesome application.

Create pipes

You can write your pipes by using the `@Pipe` decorator. For example, let's make a pipe that powers a number input with a given exponent number. First of all, write a simple class that will be the pipe.

```
export class PowerPipe {
}
```

Next, decorate this class with the `@Pipe` decorator and define the pipe's name. In this case, the pipe is named `power`.

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

@Pipe({ name: 'power' })
export class PowerPipe {
}
```

Last, implement the `transform` method. It's defined in the `PipeTransform` interface.

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({ name: 'power' })
export class PowerPipe implements PipeTransform {

  transform(input: number, exp: number): number {
    return Math.pow(input, exp);
  }
}
```

The `transform` method takes several arguments. The first is always an input value. The other arguments are all pipe's parameters. Above, the pipe can be used by `{{ 2 | power:3 }}` and then it displays `8`.

If you want to take multiple parameters like `SlicePipe`, you can add method arguments.

```
// {{ input | somePipe:param1:param2 }}
transform(input, param1, param2) {
}
```

Use custom pipes

You can use built-in pipes by default. To use your custom pipe, you have to notify it to your component. There is `pipes` property in `@Component` decorator's metadata to register additional pipes.

```
@Component({
  selector: 'my-cmp',
  template: `
    {{ 2 | power:3 }}
  `,
  pipes: [ PowerPipe ]
})
export class MyComponent {
}
```

Chapter 9. Routing

The Angular 2 router is strictly connected to the component. The main purpose of it is to have routable components. They are often smart components that are fetching resources from the API.

Configuration

Let’s start with adding a router to the app. The very beginning is to add the general URL to that app which is accessible. This usually ends up with `/` as the entrance point, but you may need to change it such as when you are trying to connect Angular 2 with an existing app.

The entire application is openly available at https://github.com/flauc/angular2-edge-app.

The base href is achieved by adding the element `base` (in `index.html`):

```
<base href="/">
```

This is the only step you have to make in HTML. Next it’s time to configure providers. We move to the place where the app is being bootstrapped and import `provideRouter`:

```
import { bootstrap } from '@angular/platform-browser-dynamic';
import { provideRouter } from '@angular/router';

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

bootstrap(AppComponent, [
  provideRouter(AppRoutes)
]);
```

This way we are able to inject router providers anywhere in the app and they will be singletons.

Router Outlet

Now it's time to add the first routable component. First of all, let's recap the most basic component that `App` could be:

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

@Component({
  selector: 'app',
  template: 'Hello App'
})
class AppComponent {};
```

To add some routes we can start with their definitions:

```
import { Component } from '@angular/core';
import { RouterConfig } from '@angular/router';
import { LoginComponent } from './pages/login/login.component';
import { SignupComponent } from './pages/signup/signup.component';
import { DashboardComponent } from './pages/dashboard/dashboard.component';
import { UserStoreService } from './common/services/userStore.service';
import { HomeComponent } from './pages/home/home.component';

@Component({
  selector: 'app',
  template: 'Hello App'
})
class AppComponent {};

const AppRoutes: RouterConfig = [
  { path: '', component: HomeComponent },
  { path: 'login', component: LoginComponent },
  { path: 'signup', component: SignupComponent },
  { path: 'dashboard', component: DashboardComponent },

  // Catch route
  { path: '**', component: HomeComponent }
]
```

The routes now have to be provided to the module. In our case it'll be the `App` module:

```
@NgModule({
    // ...
    imports: [
        AppRoutes
    ],
    // ...
})
class AppModule { }
```

Now we do have configuration for this particular modules, but routes won't be visible yet. It's happening because of the lack of a place where the router should render its content. So we need `<router-outlet></router-outlet>`:

```
@Component({
  selector: 'app',
  template: `
    <router-outlet></router-outlet>
  `
})
// ...
```

This time, instead of a simple string, we're using `RouterOutlet` to make a placeholder for routable components. If you change the URL to match some path it should render the proper component.

Router Link

It works perfectly but lacks a very important feature - links to other routes. This one is achieved through the `RouterLink` directive. With the appropriate parameters it can route to named routes. Look at the template:

```
import { Component } from '@angular/core';
import { RouterConfig } from '@angular/router';
import { LoginComponent } from './pages/login/login.component';
import { SignupComponent } from './pages/signup/signup.component';
import { DashboardComponent } from './pages/dashboard/dashboard.component';
import { UserStoreService } from './common/services/userStore.service';
import { HomeComponent } from './pages/home/home.component';

@Component({
  selector: 'app',
  template: `
    <nav>
      <ul>
        <li><a routerLink="/home">Home</a></li>
        <li><a routerLink="/dashboard">Dashboard</a></li>
        <li><a routerLink="/login">Log In</a></li>
        <li><a routerLink="/signup">Sign Up</a></li>
      </ul>
    </nav>
    <router-outlet></router-outlet>
  `
})
class AppComponent {};

export const AppRoutes: RouterConfig = [
  { path: '', component: HomeComponent },
  { path: 'login', component: LoginComponent },
  { path: 'signup', component: SignupComponent },
  { path: 'dashboard', component: DashboardComponent },

  // Catch route
  { path: '**', component: HomeComponent }
]
```

The configuration and code up to this point should give a working route to the proper components basing on the routing table provided for the component.

Route Parameters

We're able to route to the static URL path, but now it's time to make it more dynamic. The first thing you meet when trying to solve real problems is to add a dynamic `id` to the URL. It's quite easy in Angular 2. Let's consider following the dashboard component:

```
@Component({
    selector: 'dashboard',
    template: `
      <router-outlet></router-outlet>
      <a [routerLink]="['./', 'room1']">Go to room1</a>
    `
})

export const DashboardRoutes: RouteConfig = [
    { path: '/', component: DashboardMainComponent },
    { path: '/:name', component: RoomComponent },

    // Catch All route
    { path: '**', component: DashboardMainComponent }
]
```

In the routes you'll see:

```
{ path: '/:name', component: RoomComponent },
```

It indicates a path with a parameter. This path can be supplied either by entering with the proper URL, or by adding a router link with a second parameter in the array:

```
<a [routerLink]="['./', 'room1']">Go to room1</a>
```

The router link gets a path as the first item in the array. It can be either absolute or relative. The second one is part of the URL. In our case it passes the name to the component, which is accessible by `/:name`. The first is the map of query string parameters.

Accessing the Router state

On the other hand, in the component we just routed to, it would be nice to get the params there. Routable component is shipped with `ActivatedRoute` injectable. This is the way we can use:

```
constructor(activatedRoute: ActivatedRoute) {
  activatedRoute.params.subscribe(currentParams => {
    console.log('params', currentParams);
  });
}
```

The important part is the `subscribe`. Params is not just the value, *it's a stream of values*. Each time the URL changes, this observable will emit a new set of params reflecting a state. Params are not the only value we can get.

`ActivatedRoute` gives as following observables:

- url
- params
- data
- queryParams
- fragment

Resolve

If you were working with `ui-router` in AngularJS you probably heard of `resolve` feature. It's the answer to a common problem of fetching data when entering the route. Generally you have two options:

- render the component and after the data are available then put it in the proper place
- wait for data to be fetched and when they're ready, then render a fully filled component

The resolve comes to solve the second problem.

Let's start with route definition. What we have to do is to add `resolve` property to the describing object and then give a name to the resource we want to be resolved. In our case we want to wait for `rooms`:

```
export const AppRoutes: RouteConfig = [
  {
    path: 'dashboard',
    component: DashboardComponent,
    resolve: {
      rooms: DashboardResolver
    }
  }
]
```

You can see the `DashboardResolver` variable assigned to it. This one we've not yet created. The *Resolver* is generally a class that implements `resolve` method. In a simple version it can look like this:

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

@Injectable()
export class DashboardResolver implements Resolve<any> {
  constructor(private _apiService: ApiService) {}

  resolve() {
    return this._apiService.getRooms();
  }
}
```

The only missing part is to make this resolver available in the module. It's treated as provider:

```
@NgModule({
  // ...
  providers: [DashboardResolver],
  // ...
})
class DashboardModule {
  //...
}
```

Now we can try to make use of what we have just created. Resolved streams are available at

the `data` property of `ActivatedRoute`. We are introducing one more thing to make it reactive - we take just `rooms` using `pluck` operator and the result (which is observable) is put into the view with `async` pipe:

```
@Component({
  template: `
    <room *ngFor="let room of rooms | async"></room>
  `
})
class DashboardComponent {
  public rooms: Observable<any>;

  constructor(activatedRoute: ActivatedRoute) {
    this.rooms = activatedRoute.data.pluck('rooms');
  }
}
```

Guards

When we're building apps that requires authentication it's cool to have some routes protected. Although front end should not take a responsibility over the data security, we can hide some components and give a feedback to the user when there's no access to given part of the app.

Angular 2 introduces guards, actually 4 of them:

- `CanLoad` - defines whether given part of code can be loaded
- `CanActivate` - defines whether component can be rendered when routed to
- `CanActivateChild` - defines whether component can render its children
- `CanDeactivate` - defines whether a component can be deactivated

The first one is here to make sure the code is not loaded for users that are not allowed to see it. It's connected with lazy loading.

The second and third are main blocks for checking user's permissions. Typically it's used for checking the user session.

The last one it's usually used to ask for confirmation about leaving. Image the popup which asks if you're sure you want to leave the page with unsaved results.

Let's get the `CanActivate` as an example:

```
export const AppRoutes: RouteConfig = [
  {
    path: 'dashboard',
    component: DashboardComponent,
    canActivate: [AuthGuard]
  }
]
```

The guard itself implements particular interface and returns the information whether user can access the component or not:

```
import { Injectable } from '@angular/core';
import { CanActivate, Router}  from '@angular/router';
import { UserStoreService } from '../services/user-store.service';

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(
    private _storageService: UserStoreService,
    private _router: Router
  ) {}

  canActivate() {
    const user = this._storageService.getUser();
    return user && user.token;
  }
}
```

Guards are added to the module as any other provider:

```
@NgModule({
  // ...
  providers: [AuthGuard],
  // ...
})
class DashboardModule {
  //...
}
```

Conclusion

Router for Angular 2 took a long route to be in the current state. Taking the best parts of the *ui-router* and *ngrx/router* and giving considered and complex DSL, seems to be solid and filled with all of the features expected from the modern router. There's a lot more to add about the router than we just covered, as all of its features deserve their own book.

Chapter 10. Forms

Forms are an essential part to all business applications. They are used to login, to submit requests, to place orders, and they have countless other uses. Developers need to pay attention to forms because they modify data, provide for validation, and they display the changes that are reflected on the page.

As a web developer, you must effectively use the correct HTML tags for the form to be displayed to the user. The real challenge is to create an optimal data entry experience that will guide the user efficiently and effectively through the workflow behind the form.

Fortunately, Angular 2 developers took the time to really integrate new tools for developers to create methodical forms through controls, validators, and observers. Controls are what encapsulates the inputs in our forms and gives us the objects to work with. Validators allow the ability to validate inputs to our own discretion. Finally, observers let us watch the form change and respond.

FormControls

Two fundamentals as developers you need to pay attention to in the ng2 form are the `FormControl` and `FormGroup`. Since `FormControls` represent a single input field, it is considered the smallest unit of an Angular form. This is important to remember, because it encapsulates the value the field gave us and the state if the field is valid or not. We use `FormControls` to build up the forms we create and attach metadata and logic to them.

For example, you can create a `FormControl` in TypeScript like this:

```
let exampleControl = new FormControl (“Jane”);
let name = exampleControl.value;

exampleControl.errors
exampleControl.dirty
exampleControl.valid
```

We need to remember that in Angular 2, you have to attach the `FormControl` class to the DOM. Make sure to include the `formControl` attribute. This creates a new `FormControl` object within the context of our form.

```
<input type=”text” [formControl]=”name” />
```

FormGroup

In the previous example, our form had only one field. Angular 2 has developed `FormGroup` to manage multiple `FormControl`s. Developers use `FormsGroups` because it becomes inconvenient to iterate over an array of `FormControls` and check to see if they are valid. `FormsGroups` have a wrapper interface around the collection of `Controls` for better management and organization of forms.

```
let exampleInfo = new FormGroup ({
    firstName: new FormControl("Jane"),
    lastName: new FormControl("Doe")
})
```

You'll notice that when you try to get the value from a `FormGroup`, you'll receive an `object` with key-value pairs. As web developers, this is appreciated because you do not have to iterate each `FormControl` to get the full set of values from the form.

```
exampleInfo = exampleInfo.value;

exampleInfo.errors
exampleInfo.dirty
exampleInfo.valid
```

Your First Form

Lets create a hypothetical form that will store a product name. We will have one input and a submit button. We will also turn our form into a component to make it available throughout the application.

The entire application is openly available at https://github.com/flauc/angular2-edge-app.

FormsModule & ReactiveFormsModule

There are two ways to use the forms library in Angular 2: `FormsModules` and `ReactiveFormsModule`. `FormsModule` allows us to use template directives such as `ngModel` or `NgForm`. Whereas, `ReactiveFormModule` allows for the use of directives `formControl` or `ngFormGroup`. By importing both libraries into our application, you open up the possibilities of using all the available directives.

```
 import {
  FormsModule,
  ReactiveFormsModule
} from '@angular/forms';


@NgModule([
.......
 imports: [
  FormsModule,
  ReactiveFormsModule
 ],
 .....
])
.....
```

The Component

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

@Component({
    selector: ‘product-form’,

})
```

In this component, we have defined the selector as `product-form` in order to call it into our HTML, where it should look like this:

```
<product-form></product-form>
```

The Template

Now let’s add our form template to the component.

```
import { Component } from ‘@angular/core’;


@Component({
    selector: ‘product-form’,
    template: ‘
    <div class=”form-group”>
        <h2 class=”header”> Product Name Form</h2>
        <form #f=”ngForm”
            (ngSubmit)=”onSubmit(f.value)” class=”form”>

    <div class=”field”>
        <label for=”productInput”> Product Name</label?
   <input type=”text”
    id=”productInput”
    placeholder=”Product Name”
    ngControl=”product”>
    </div>

    <button type=”submit” class=”button”>Submit</button>
    </form>
</div>
‘
})

export class ProductForm {
 onSubmit(form: any): void {
  console.log('The product name you submitted:', form);
 }
}
```

By using `FormsModule`, we opened a huge door for many different kinds of selectors to be used. One of the most important selectors, forms requires `ngForm` as an attribute. This is significant because ngForm automatically attaches to any form tag in that view.

By having `#f=”ngForm”` in our template, we are creating a local variable for this view. This is happening because we are creating an alias to find to the #f variable. `ngForm` is a `FormGroup` object because we can use `f` as a `FormGroup` in our view.

The input tags are more tied to CSS frameworks and are not really associated with Angular.

`NgModel` is another significant directive that we need to pay attention to because it specifies a selector of `FormControl`. It is important because we can attach our input tag by adding an attribute to it. We used `FormControl=”product”`. `FormControl` will eventually add a new `FormControl` to the parent `FormGroup`, and will bind a DOM element to the new `FormControl`.

FormBuilder and `form`

When we build our forms using `FormControls` and `FormGroups` with `ngForm` and `ngControl`, it doesn't allow us many options for customizing. However, if you use `FormBuilder`, you have more flexibility to customize your form.

`FormBuilder` is a helper class that helps build forms. Let's build a form with `FormBuilder`.

First, we create our component:

```
export class ProductName {
      form: FormGroup;

    constructor(fb: FormBuilder) {
         this.form = fb.group({
   'name':['ExampleProduct']
 });
}

    onSubmit(value: string): void {
     console.log('The product name you submitted: ', value);
 }
}
```

An instance of `FormBuilder` is created during injection and is assigned an `fb` variable in the constructor. Our variable is called `form`. `form` is typed in a `FormGroup`. Our `FormGroup` is called `fb.group()` and takes an object of key-value pair that specifically target the `Controls`.

Formbuilder in Your Views

In order to use `FormBuilder` in our template, we must call the correct tag in HTML. Since our instance variable is `form`, we will use the HTML tags `<form></form>`. However, we need to use `formGroup` because `formGroup` creates its own existing `FormGroup`, and we do not want to go outside of it.

It should look similar to this:

```
form [formGroup]="form"
(ngSubmit)="onSubmit(form.value)"
```

Lastly, we need to bind our `Control` to input the tag. Since `ngControl` creates a new object and will attach the parent `FormGroup`, we need to use `FormBuilder` to create our own `formControls` to bind with the input tag. In order to do this, we need to bind the existing `formControl` to `formControl`. `FormControl` instructs the directive to look at `form.controls` and to use the existing `product formControl` for this input.

```
<input type="text"
  id="productInput"
  placeholder="Product Name"
  [formControl]="form.controls['product']">
```

Validators

Some users aren't always going to enter the correct information into the input fields. The solution for this is using validations to provide feedback to the user and not have the form be submitted.

Validators are provided by the `Validators` module. The most simple validator is `Validators.required`, which is designed for a required field or the `FormControl` will be considered invalid. We need to remember that in order to use validators, we need to assign the validator to a `FormControl` object and check the status of the validators through the view.

In order to assign a validator to a `FormControl` object, we pass it as a second argument to our `FormControl` constructor.

```
let control = new FormControl('product', Validators.required);
```

However, if you are using `FormBuilder`, you would use this syntax:

```
 constructor(fb: FormBuilder) {
     this.form = fb.group({
         'product': [ ' ',  Validators.required]
 })
}
```

After the validator is written, we need to use the validation in the view by

1. Specifically assigning the `FormControl product` to an instance variable to give you easy access to the view. Or
2. Lookup the `FormControl product` from `form` in the view.

In order to add the `FormControl product` to an instance variable, we set each `FormControl` up as an instance variable in your component definition class.

```
export class FormValidation {
    form: FormGroup;
    product: ExampleControl;

 constructor(fb: FormBuilder) {
 this.form = fb.group({
    'product': [' ', Validators.required]
});

 this.product = this.form.controls['product'];
}

 onSubmit(value: string): void {
  console.log (The product name you submitted: ', value);
}
```

Since the `product` is being validated, we now want to look at our view checking for the forms entire validation, the individual field, and displaying message validation. We also check the coloring of the field if it's invalid, check the validity of the individual field on a certain

requirement, and display the message.

Form Messages

To check the validation of the entire form, you can specifically look at `form.valid`:

```
div *ngIf= “!form.valid”
```

Since `form` is a `FormGroup`, the `FormGroup` is valid when the children `FormControls` are valid.

Field Message

Similarly, we can display a message for a specific field if the field's `FormControl` is invalid.

```
div *ngIF= “!product.valid”
```

Field Coloring

If you add the Semantic UI CSS Framework into your application, you can have it do most of the work for you. The UI CSS Framework adds a `.error` class to `<div class= “field”>`.

```
<div class=”field”
    [class.error]=”!product.valid && sku.touched”>
```

Specific Validation

Forms can be invalid for different reasons, so it is important to be able to show different messages to indicate the reason the form is invalid. You can look up a validation failure by using the `hasError` method.

```
div *ngIf="product.hasError('required')">
```

Remember that `hasError` is defined in the `FormControl` and the `FormGroup`. This means that you can pass another argument to lookup a specific field in the `FormGroup`.

Personalized Validations

Often times, we want to write our own validations. In order to understand how our validators are implemented, its important to understand the class, `Validators.required` from Angular's core.

A validator will take a `FormControl` as its input and return a `StringMap<string, Boolean> where the key is "error code"`. When that value is true, it fails.

Writing a Personalized Validator

In our forms, we can assign specific requirements to our `product`. In our example, let's give our `product` a requirement that it needs to be assigned a `KU_` in front of every product name. The validator would be written similar to this:

```
function productValidator(control: FormControl): { [s: string]: boolean] } {
    if (!control.value.match(/^KU_/))  {
        return { invalidProduct: true};
}
}
```

But now this validator needs to find a home in our `FormControl`. Since we already have a validator in our `product`, we need to use `Validators.compose`.

```
This.form = fb.group({
    'product': [' ', Validators.compose([
 Validators.required. productValidator])]
});
```

When we use `Validators.compose`, it wraps multiple validators and assigns them to the `FormControl`. This lets the `FormControl` be valid when both validations pass.

In our view, we would use our new validator like this:

```
div *ngIf="product.hasError('invalidProduct')"
    class="error message"> Product name must begin with a KU_</div>
```

ngModel, the directive

`ngModel` is a special directive that we use in our forms because it will bind a model to a form. But what makes `ngModel` particularly significant is that it can implement two-way data binding. Two-way data can be rather difficult at times, but it can be necessary in some situations.

If we were to add another input field to our form, we can use `ngModel` to keep the component instance variable in line with the view.

The component class can look similar to this:

```
export class ExampleFormModel {
    form: FormGroup;
    productDescription: string;

constructor(fb: FormBuilder) {
    this.Form = fb.group({
        'productDescription': [' ', Validators.required]
    });
}

onSubmit(value: string): void {
    console.log('You submitted a product description: ' value);
}
}
```

Now, lets put `ngModel` in the input tag:

```
<input type="text"
     id="productDescription"
     placeholder= "Product Description"
     [[FormControl]="form.find('productDescription')"
     [(ngModel)]="productDescription">
`
```

By using brackets and parenthesis around `ngModel`, it indicates two-way binding.

Finally, let's display `productDescription` within our view:

```
<div class="product description">
    The product description is: {{productDescription}}
</div>
```

Summary

Forms are one of the biggest assets you can use for data collection in any type of business web application. It is up to you to decide the best and most effective approach to design and use the form. This is important to remember when implementing a form in an application, and to consider the usability and experience the user will have.

Chapter 11. Change Detection

Change detection is one of the most important features in Angular 2. It covers everything from how is data-binding working to who is checking changes? By understanding these important topics, you can see the logic that is running internally in Angular2. And, you can also understand ways of using performance tuning. By controlling change detection, you can build a dramatically faster application.

What’s Change Detection?

At first, you should review more about **rendering**. Rendering is a process to map *models* to *views*. The *models* can be primitives, objects, arrays or any JavaScript data. And the views can be headers, paragraphs, forms, buttons or any user-interface elements. Generally, those are represented as the *Document Object Model* (DOM).

Well, a simple rendering example is as follows:

```
<h1 id="greeting"></h1>

<script>
document.getElementById("greeting").innerHTML = "Hello World!";
</script>
```

This is a very easy example, because this model will never change, so rendering is needed only once. And it gets so complex when the data changes at runtime. Rendering multiple times to sync models and views, we must think about the following things:

- **What** change happened?
- **Where** did the change happen at?
- **When** did the change happen?
- **How** will the view be updated?

The basic role of change detection is to manage these things above. When a change happens, a manager of the component states, **change detector**, detects the change and notifies it to the render to update the view. Simply, a change detector has two tasks:

- Detect changes from models
- Notify changes to views

Changes and Events

Well, we figured out about the roll of change detection. But then you may wonder; *“what’s a change?”*. A change is the difference between an old model and a new model. In other words, *a change makes a new model*. Let’s look at the following code:

```
@Component({
  template: `
  <span>{{counter}}</span>
  <button (click)="countUp()">Count up</button>
  `
})
class MyComponent {
  counter = 0;

  countUp() {
    this.counter++;
  }
}
```

Beginning rendering, `{{counter}}` in the template will be rendered as `0` by view interpolation. And when the button is clicked, the `counter` property will be changed in the click event handler, and finally counted up so `counter` will be applied to the view; `<span>1</span>`. In this case, **click event** causes a change and the change is a mutation of the property.

Let’s take a look at the next example:

```
class MyComponent {
  counter = 0;

  ngOnInit() {
    setInterval(() => {
      this.counter++;
    }, 1000);
  }
}
```

This component has a timer to count up its counter every second. In this case, it’s a **timer event** causes a change. Finally, let’s look at the following example:

```
class MyComponent {
  data = {};

  constructor(private http: Http) {}

  ngOnInit() {
    this.http.get('/data.json')
      .map(res => res.json())
      .subscribe(data => {
        this.data = data;
      });
  }
}
```

This component will send an HTTP request once it is initialized. When that response will be back, the `data` property of the component will be updated. In this case, it’s a **XHR callback** that causes a change.

In summary, there are three things that can cause a change of the model:

- **Events**: click, blur, submit, ...
- **Timers**: setInterval, setTimeout, ...
- **XHRs**: Ajax(GET, POST, ...)

And these have a thing in common: they are **asynchronous** operations. And now, we can say also that: **all asynchronous operations can cause changes**.

Great, you learned about what causes changes and when it's caused! But you don't know yet who notifies the changes to views. So next, we'll dive into a mechanism that allows Angular to detect changes anytime. It's called **Zone**.

Zones

Zone is one of proposals of the next ECMAScript specification. An Angular team is developing its implementation as zone.js. It says in that source code:

> Zone is a mechanism for intercepting and keeping track of asynchronous work.

A Zone is a global object that is configured with rules about how to intercept and keep track of the asynchronous callbacks. Zone has these responsibilities:

1. Intercept asynchronous task scheduling.
2. Wrap callbacks for error-handling and zone tracking across async operations.
3. Provide a way to attach data to zones.
4. Provide a context specific last frame error handling.
5. Intercept blocking methods.

As a simple example, consider the following code:

```
Zone.current.fork({
  onInvokeTask: (parent, current, target, task) => {
    console.log('Before task');
    parent.invokeTask(target, task);
    console.log('After task');
  }
}).run(() => {
  setTimeout(() => {
    console.log('Task');
  }, 1000);
});
```

The above will log:

```
'Before task'
'Task'
'After task'
```

This is not magic! Zones allow you to hook handlers on any async tasks. In Angular, there is **NgZone**, which is a customized zone for Angular. You can see that at the implementation of `ApplicationRef`:

```
// summarized code
class ApplicationRef {
  constructor(private zone: NgZone) {
    this.zone.onMicrotaskEmpty
      .subscribe(() => {
        this.zone.run(() => {
          this.tick();
        });
      });
  }

  tick() {
    this.changeDetectorRefs
      .forEach((ref) => ref.detectChanges());
  }
```

```
}
```

In the above, when the zone has no tasks, a handler of that event calls `tick()`. It executes change detection for each component of the application.

Summarizing the above:

1. Async operations are scheduled as a task
2. Zones observe the task execution
3. Angular handles events caused by execution of async operations
4. Change detectors run in all components

It’s probably difficult to understand all about zones, but its behaviors in the application is simple. You don’t have to consider about when and where a change happens. Angular is observing those always and detects all changes.

Change Detectors and Components

So far, we outlined the change detection of Angular 2. Here, we'll talk about its performance and how data-binding works in your application.

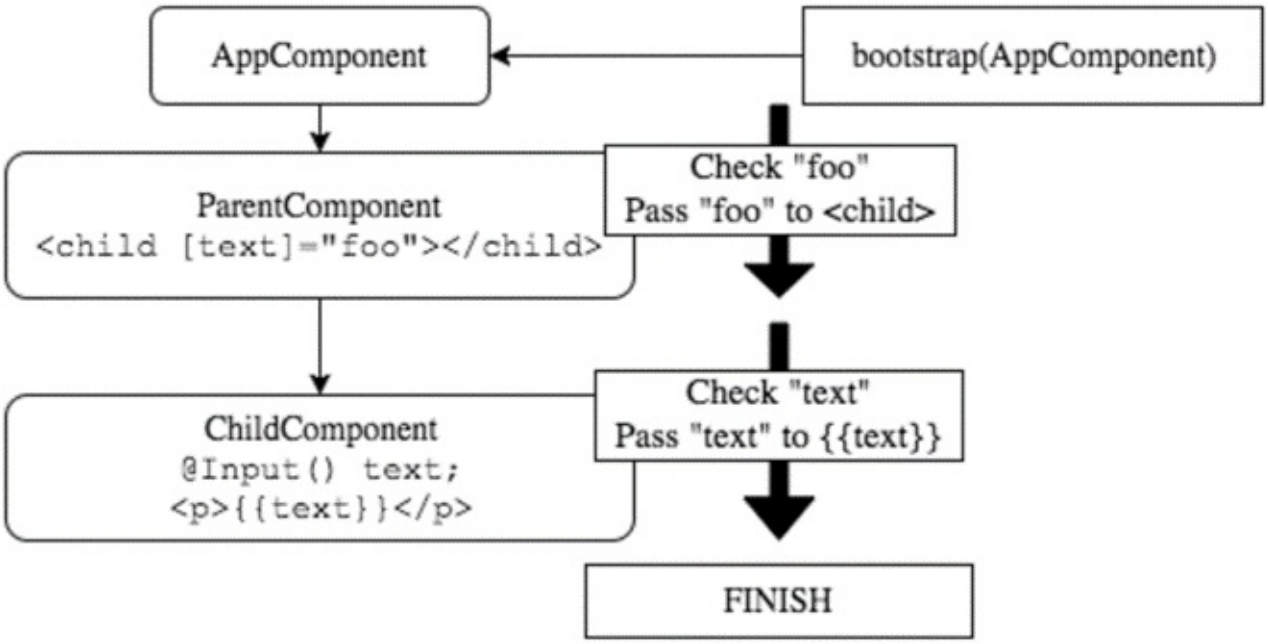
As you know, an application is a tree of components. And now an important thing is that **each component has its own change detector**. It means that **an application is a tree of change detectors**.

By the way, you might wonder. Who does make change detectors? It's good question. They're made by **code generation**. Angular 2 generates them for each component. And those codes are never in polymorphic. There are monomorphic **VM-friendly** codes. This is one of why new Angular is dramatically fast. Basically, each component can execute hundreds of thousands of detections within a couple of milliseconds. You can make fast your application without performance tuning.

Another reason is caused by the change detector tree. In Angular 2, any data flows from top to bottom. Let's see the following components:

```
@Component({
  selector: 'child',
  template: '<p>{{text}}</p>'
})
class ChildComponent {
  @Input() text;
}

@Component({
  selector: 'parent',
  template: '<child [text]="foo"></child>',
  directives: [ChildComponent]
})
class ParentComponent {
  foo = 'bar';
}
```

Change detection always begins at the root component. So in the above example, a detection of `ParentComponent` is earier than `ChildComponent`, and as a result, the `foo` property is passed to the child as `text` input. Then, `ChildComponent` detects a change of `text` property; it changed from `""` into `"bar"`. This is simple but very important. After `ParentComponent` was checked, its change detector doesn't have to run, because it can be updated by only its parent. For single change detection, **every component will be checked only once**.


AppComponent
bootstrap(AppComponent)
ParentComponent
<child [text]="foo"></child>
Check "foo"
Pass "foo" to <child>
ChildComponent
@Input() text;
{{text}}
Check "text"
Pass "text" to {{text}}
FINISH

OnChanges

When any changes of input properties in your component are detected, an event is caused. Then you can get detail those changes as an argument. For example:

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

@Component({...})
class MyComponent implements OnChanges {
  @Input() prop;

  ngOnChanges(changes: {[propName: string]: SimpleChange}) {
    let newValue = changes['prop'].currentValue;
  }
}
```

The **`OnChanges`** interface defines a method, **`ngOnChanges`**. The argument, `changes`, has all of the changes of the component as a key-value map. A value of the map is `SimpleChange`, which has two properties: `previousValue` and `currentValue`. The earlier is an old value before the change, and the later is a new value.

A point you should be careful with is that the hook won't be called by itself. An example is the following:

```
@Component({...})
class MyComponent {
  @Input() prop = null;

  ngOnChanges(changes) {
    // never called after onSomeEvent
  }

  onSomeEvent() {
    this.prop = someExpression();
  }
}
```

`prop` is a property of `MyComponent` and it is modified by itself. In a case like this, `ngOnChanges` is never called. It's called only when its property is changed by its parent. So, you must never modify any input properties in your own component. You should access those as read-only.

Change Detection Tuning

Now, you've finished learning about the basis of change detection! And later, we'll talk about how to tune your change detection. In Angular 2, you can configure change detection behavior for your application and optimize it. As mentioned earlier, change detection is very fast without any tuning. But if you want, you may make its change detection smarter and quite faster.

Change Detection Strategy

In order to customize change detection of your component, there is a utility: `ChangeDetectionStrategy`. Using this, you can change the strategy of change detection for each component.

First, let's imagine a component like this:

```
@Component({
  selector: 'profile-card',
  template: `
  <div>
    <profile-name [name]="profile.name"></profile-name>
    <profile-age [age]="profile.age"></profile-age>
  </div>
  `,
  directives: [ProfileNameComponent, ProfileAgeComponent]
})
class ProfileCardComponent {
  @Input() profile;
}
```

`ProfileCardComponent` is a component, which has `profile` field as an input property. And its template (in other words, **view**) depends on only that property. So, this component and its child components depend on an input property.

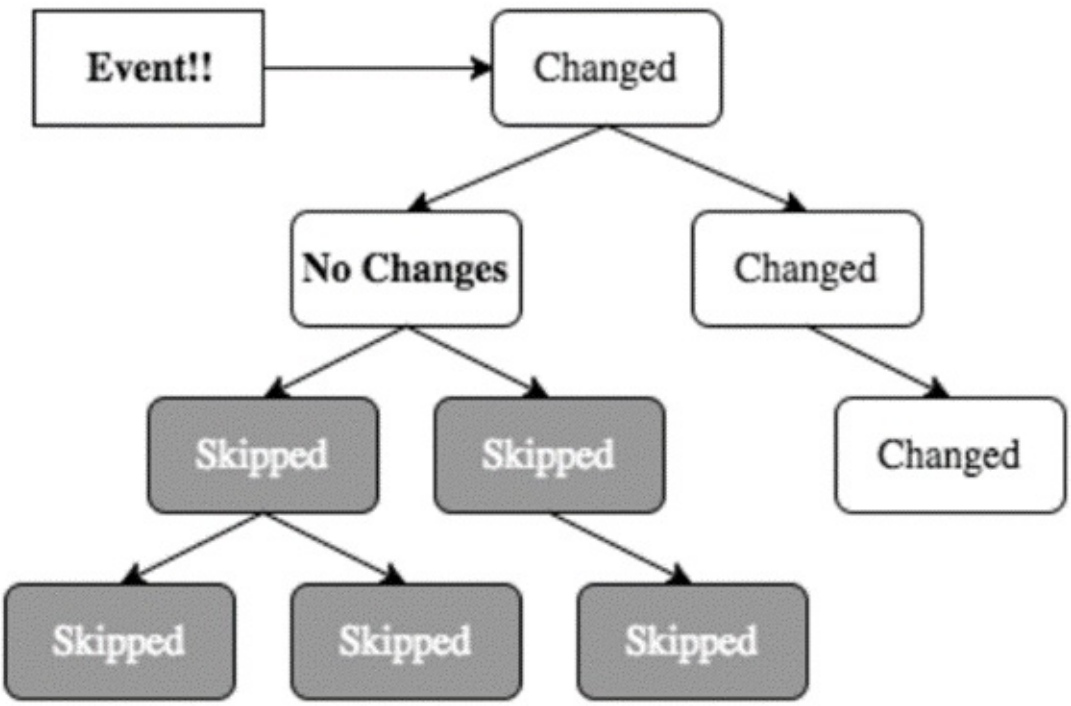
Without a strategy, change detection runs from the root component to every leaf component. But in this case, it's unnecessary to check children of `ProfileCardComponent` if `profile` is not changed. Simply, **you can stop checking at `ProfileCardComponent`**. Well, there is a strategy for the case that uses `OnPush`.

OnPush Strategy

OnPush* is an option of change detection strategy. Let's look at the following code:

```
@Component({
  selector: 'profile-card',
  template: `
  <div>
    <profile-name [name]="profile.name"></profile-name>
    <profile-age [age]="profile.age"></profile-age>
  </div>
  `,
  directives: [ProfileNameComponent, ProfileAgeComponent],
  changeDetection: ChangeDetectionStrategy.OnPush,
})
class ProfileCardComponent {
  @Input() profile;
}
```

There is a new component setting, `changeDetection`, which take a strategy provided as a static field of `ChangeDetectionStrategy`. Using this strategy, the component skips its children's change detection when no changes are caused by its parent. Reducing the number of checks affects a performance of the application directly. As far as possible, you should use the `OnPush` strategy on your components, and for this, you should increase components that depend on only input properties.



Mutable and Immutable

Using the OnPush strategy, it's very important to learn about *mutable* and *immutable*. Basically, change detectors can detect only **reference changes**. Let's see the next bad example:

```
@Component({
  template: `
  <div>
    <profile-card [profile]="profile"></profile-card>
  </div>
  `,
  directives: [ProfileCardComponent],
})
class AppComponent {
  profile;

  ngOnInit() {
    this.profile = { name: 'Brad Green' };
  }

  changeProfile() {
    this.profile.name = 'Igor Minar'
  }
}
```

In this case, the `profile` property of the `AppComponent` is modified but **its reference is never changed**. With *mutable* data like this, the OnPush strategy doesn't work well because the change detector of `AppComponent` cannot know whether `profile` was changed. So, the OnPush strategy needs **immutable data** to detect changes at the component closer to the root.

This is the rewritten code:

```
class Profile {
  constructor(base) {
    this.name = base.name;
    this.age = base.age;
  }

  setName(newName) {
    return new Profile({name: newName, age: this.age});
  }
}

@Component({
  template: `
  <div>
    <profile-card [profile]="profile"></profile-card>
  </div>
  `,
  directives: [ProfileCardComponent],
})
class AppComponent {
  profile: Profile;

  ngOnInit() {
    this.profile = new Profile({ name: 'Brad Green' });
  }

  changeProfile() {
    const newProfile = profile.setName('Igor Minar'); // make new reference
    profile === newProfile; // false
    this.profile = newProfile;
  }
}
```

The reference of `profile` is changed in `changeProfile()`. Because `profile` became immutable, `ProfileCardComponent` can check whether `profile` was changed strictly, and it can control its children’s change detection.

Using Observable

The OnPush strategy is an easy and great way to improve the performance of your application. Well, there is another way to get better performace. It is to use **observable** properties with `ChangeDetectorRef` utility. This way allows you to control all of the change detection manually.

ChangeDetectorRef

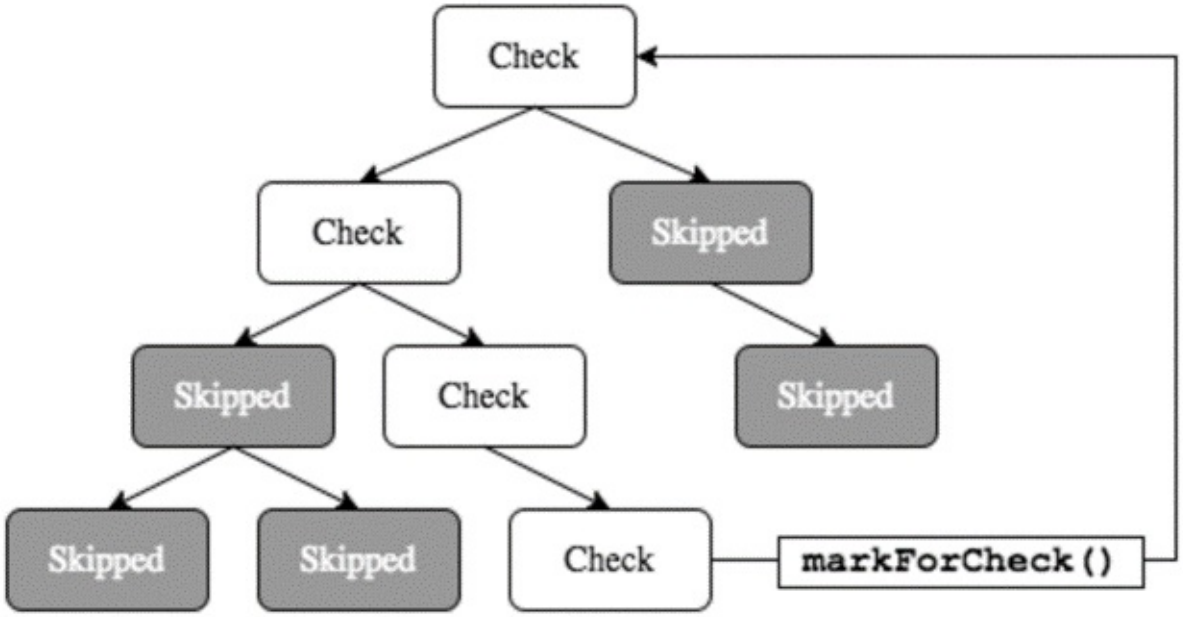
`ChangeDetectorRef` is a reference of the compoment's change detector. This can be implemented by dependency injection at the component:

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

@Component({})
class MyComponent {
  constructor(private cdRef: ChangeDetectorRef) {}
}
```

`ChangeDetectorRef` has some methods. In those, `markForCheck()` is the most useful method. It marks components from the root to the place as **to be checked**. Let's take a look at the following:

```
@Component({
  selector: 'child',
  template: `<p>{{counter}}</p>`,
  changeDetection: ChangeDetectionStrategy.OnPush,
})
class ChildComponent {
  counter = 0;
  constructor(private cdRef: ChangeDetectorRef) {}

  ngOnInit() {
    setInterval(() => {
      this.counter++;
      this.cdRef.markForCheck();
    }, 1000);
  }
}

@Component({
  selector: 'parent',
  template: `<child></child>`,
  changeDetection: ChangeDetectionStrategy.OnPush,
})
class ParentComponent {
}
```

`ChildComponent` is defined as the OnPush component, but it doesn't have any input properties. When will `ChildComponent` be checked? Well, look at its `ngOnInit`! In the interval function, `cdRef.markForCheck()` is called per 1 second. Even without any changes or events, if `markForCheck()` was called, that component is included in the next change detection process.



Next, we talk about `cdRef.detach()` and `cdRef.reattach()`. They allow you to turn on/off change detection at the component. Let's see the following example:

```
@Component({
  template: `
    Detach: <input type="checkbox" (change)="detachCD($event.target.checked)">
    <p>{{counter}}</p>
  `,
})
class ChildComponent {
  counter = 0;
  constructor(private cdRef: ChangeDetectorRef) {}

  ngOnInit() {
    setInterval(() => {
      this.counter++;
    }, 1000);
  }

  detachCD(checked) {
    if (checked) {
      this.cdRef.detach();
    } else {
      this.cdRef.reattach();
    }
  }
}
```

This component has a checkbox to toggle its own change detection. When the checkbox is checked, `cdRef.detach()` will be called, and after that, the component and its children will be never checked. As a result, the view of the sub-tree from the component will be frozen.

And when you uncheck it, `cdRef.reattach()` will be called. After that, the component will join the change detector tree again.

Summary

Change detection of Angular 2 is a great system to manage your application states, and it can provide you with an easy way to improve the performance.

Chapter 12. RxJS

You probably have met RxJS before while playing with Angular 2. The whole idea of change detection and notifying components is based on *Observables*. This nothing new, in fact, it is about to be part of JavaScript in a future release. Despite this, we can use libraries that mimic this behavior. One of them is *RxJS*.

What is Observable?

Let's consider a JS generator for a while.

```
function* returnManyValuesGenerator() {
  yield 1;
  yield 2;
  yield 3;
}

const iterator = returnManyValuesGenerator();
console.log(iterator.next().value); // prints 1
console.log(iterator.next().value); // prints 2
console.log(iterator.next().value); // prints 3
```

But what if we want to add some asynchronous stuff into that, like `setTimeout`? It would end up with this:

```
function* returnManyValuesGenerator() {
  yield 1;
  yield 2;
  yield 3;
}

const iterator = returnManyValuesGenerator();
console.log(iterator.next().value); // prints 1
console.log(iterator.next().value); // prints 2
setTimeout(() => {
  console.log(iterator.next().value); // prints after 1000ms
}, 1000);
```

Imperative programming teaches us that each instruction is implemented one-by-one, line-by-line. The beginning of your JavaScript journey wasn't straightforward in understanding how asynchronous callbacks work. Reactive Extensions push this idea even further. The whole idea behind it is that many instructions can execute in parallel, and their results are later captured in arbitrary order, by so called *observers*. Normally, you'd call a method to get the desired result. This time you will just define how it should behave (as a Observable) and then subscribe an observer to it. This observer is invoked when some event at the observable happens.

In case of the generator's case above you've seen that the asynchronous way is possible, but the problem is that the consumer (iterator) has to be invoked asynchronously. This is useful for some cases, but how do you handle an asynchronous operation defined by the producer?

This concept you've seen in promises:

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(1);
  }, 1000);
});

promise.then((result) => {
  console.log(result); // prints 1 after 1000ms
});
```

The problem with promises is they just return one value and once invoked, cannot be cancelled (yet). Observables merge the best parts of promises and generators. They are also called next-gen promises.

Let’s try it out:

```
import { Observable } from 'rxjs/Rx';

const observable = Observable.create(observer => {
  observer.next(1);
  observer.next(2);
  observer.next(3);
});

observable.subscribe(value => {
  console.log(value); // prints 1 // prints 2 // prints 3
});
```

The output it gives looks the same as the one from generators. What’s the difference? Now the values are pushed by the observable and we just subscribed to it. Now you do not have control over the data being printed. The producer (observable) does. Now we can combine both and achieve some many asynchronously returned values:

```
import { Observable } from 'rxjs/Rx';

const observable = Observable.create(observer => {
  observer.next(1);
  observer.next(2);
  setTimeout(() => {
    observer.next(3); // pushes 3 after 1000ms
  }, 1000);
});

observable.subscribe(value => {
  console.log(value); // prints 1 // prints 2 // prints 3 after 1000ms
});
```

Observables can not only return a value, but also throw an error or finish its job.

```
const observable = Observable.create(observer => {
  observer.next(1);
  setTimeout(() => {
    observer.complete();
  }, 1000);
});

observable.subscribe(function onNext(value) {
  console.log(value); // prints 1 // prints 2 // prints 3 after 1000ms
}, function onError(error) {
  console.error(error);
}, function onComplete() {
  console.log('completed');
});
```

Note, if you throw an error inside `Observable.create` like:

```
observer.error('something went wrong');
```

In this case you’ll not be able to complete Observable.

Hot and cold observables

We're now about to write some code using built in RxJS operators. The one we'll use first will be:

- interval - emits an incremented value in given interval (counts from 0)
- take - takes only a specified number of emitted events

So we initialize new observable and subscribe to it:

```
const observable = Observable.interval(1000).take(5);

observable.subscribe(value => {
  console.log(value);
});
```

You should see results like:

```
0
1
2
3
4
```

Let's add another subscriber, but delayed by 1500ms:

```
observable.subscribe(value => {
  console.log(`a: ${value}`);
});

setTimeout(() => {
  observable.subscribe(value => {
    console.log(`     b: ${value}`);
  });
}, 1500);
```

Now the result seems quite strange like:

```
a: 0
a: 1
     b: 0
a: 2
     b: 1
a: 3
     b: 2
a: 4
     b: 3
     b: 4
```

This means observable is **cold**. What does it mean? Each subscriber sees the same events from the beginning as it would just start. It's independent of any other subscriber. Sometimes we just don't need that. To mimic that logic we have to use some RxJS operator to make observable hot. One such operator is `share`:

```
const observable = Observable.interval(1000).take(5).share();
```

If we leave subscribers as they were, we end up with the following result:

```
a: 0
a: 1
     b: 1
a: 2
     b: 2
a: 3
     b: 3
a: 4
     b: 4
```

Reactive Angular 2

We just went through the general idea behind Reactive Extensions. It's very useful to understand what's going on under the hood. There are a few areas where Angular gives us the possibility to use RxJS, perhaps without even knowing that.

We've created outputs of the components before using RxJS Subject in fact. It means that all of the features of observables apply to it as well. However, handling events using streams is coupled into a more imperative way like calling functions with `$event`.

Event streams flow

Let's start with a very basic example. What we're going to do is to get a stream of click events from the button. To achieve that we need to create a new `Subject` from RxJS. Subject inherits from both Observer and Observable, so for us it will work in a similar manner as we'd create it with pure Observable.

```
import { Component, ChangeDetectionStrategy, Input } from '@angular/core';
import { ROUTER_DIRECTIVES, Routes } from '@angular/router';
import { Subject, Observable } from 'rxjs/Rx';

@Component({
  selector: 'app',
  template: `
    <button (click)="counter$.next(1)">Up Vote</button>
  `,
  directives: [ROUTER_DIRECTIVES, CounterComponent]
})
class AppComponent {
  public counter$: Observable<number> = new Subject<number>();

  constructor() {
    this.counter$.subscribe(console.log.bind(console));
  }
}
```

We've just subscribed to the click events. Note that now in console you see the number one each time the button is clicked. We can now make use of a whole bunch of RxJS operators on a click stream. At the very basics, we can map all values into something different:

```
...
class AppComponent {
  public counter$: Observable<string> = new Subject<number>()
      .map(value => `clicked: ${value}`)

  constructor() {
    this.counter$.subscribe(console.log.bind(console));
  }
}
```

Now instead of 1, there will be a string containing that value. In the case of up votes it would be nice to have a sum of clicks. It's also easy to do with the scan operator:

```
...
class AppComponent {
  public counter$: Observable<string> = new Subject<number>()
    .scan((acc: number, current: number) => acc + current)
    .map(value => `clicked: ${value}`);

  constructor() {
    this.counter$.subscribe(console.log.bind(console));
  }
}
```

`Subject` can be replaced here by `BehaviorSubject`, which has the advantage of taking the first value on init and emitting that.

Although we have great possibilities with streams, Angular components don't even need to

know that the value passed is the next one in the stream. Async pipe is very helpful with that. Consider the full example:

```
import { Component, ChangeDetectionStrategy, Input } from '@angular/core';
import { BehaviorSubject, Observable} from 'rxjs/Rx';

@Component({
  selector: 'counter',
  template: '{{ count }}',
  changeDetection: ChangeDetectionStrategy.OnPush
})
class CounterComponent {
  @Input() public count: number;
}

@Component({
  selector: 'app',
  template: `
    <counter [count]="counter$ | async"></counter>
    <button (click)="counter$.next(1)">Up Vote</button>
  `,
  directives: [ROUTER_DIRECTIVES, CounterComponent]
})
class AppComponent {
  public counter$: Observable<number> = new BehaviorSubject<number>(0)
    .scan((acc: number, current: number) => acc + current);
}
```

HTTP

The real thing when streams can be used (and have to be used) is the HTTP provider. In AngularJS it was done upon the Promise. Now calls like `get` or `post` return observable. It gives more control over its behavior than before. Reactive Extensions have a lot of operators that can be helpful here.

```
import { Injectable } from @angular/core;
import { Http } from @angular/http;

@Injectable()
class DataService {

  constructor(private http: Http) {}

  public search() {
    return this.http
      .get('http://localhost:3000/api')
      .map(res => res.json());
  }

}
```

Take a look at the `map` operator here. You've seen it before. It's an RxJS operator that simply maps value in a stream into another value. It applies here as well. This way, subscribing to the `search` emitted value is in the JSON format.

Forms

Another cool part of Angular that uses RxJS internally are forms. It can be handy to subscribe to the stream created for the form. It came with the new way of handling forms. `FormBuilder` gives us the `valueChanges` stream, which can be directly accessed and then manipulated in our own way like any other stream considered before:

```
import 'rxjs/Rx';
import {
  Component,
  OnInit
} from '@angular/core';
import {
  FORM_DIRECTIVES,
  ControlGroup,
  FormBuilder,
  Control,
  Validators
} from '@angular/common';

@Component({
  selector: 'app',
  template: `
    <form [ngFormModel]="form" (ngSubmit)="onSubmit()">
       <p>
          <label>First Name:</label>
          <input type="text" ngControl="firstName">
       </p>
    </form>
  `,
  directives: [FORM_DIRECTIVES]
})
class AppComponent implements OnInit {
  public form: ControlGroup;

  constructor(fb: FormBuilder) {
    this.form = fb.group({
      firstName: new Control('', Validators.required)
    });
  }

  ngOnInit() {
    this.form.valueChanges
      .filter(value => this.form.valid)
      .subscribe(console.log.bind(console));
  }

  public onSubmit() {
    // handle submit
  }
}
```

Summary

You should now have a better idea about the effectiveness of using RxJS, and in the next chapter we will examine Testing in Angular 2.

Chapter 13. Testing

One of the reasons AngularJS was introduced was a problem of testing front-end apps. It solved that problem quite nicely and version 2 follows that path despite being a totally new piece of code. What makes it a good choice is a separation of concerns. We'll focus mostly on unit testing here. Its responsibility is to ensure that every single function of a system works fine without being even aware of any other part. If the chosen framework (or library) doesn't help with such a separation, and doesn't give ability of mocking dependencies, it is really hard to do it right over time (or even impossible). We won't focus on the test setup, so please take your time to use *angular-cli* or any other starter project that suites you to follow. If you're not familiar with testing, that's OK. We'll start right now from the very beginning. Later we'll move to tests connected directly to our sample app.

First tests

The place to start with is a component. The very basic ‘Hello World’ example looks like this:

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

@Component({
  selector: 'app',
  template: '<span>Hello</span>'
})
export class App {}
```

It’s basically a class with a decorator, which adds some logic Angular makes use of. The important thing is decorators **add** some stuff to the class. It means we can still do: `new App()` and end up with a pure JavaScript object!

Now let’s add some property to the class:

```
@Component({
  selector: 'app',
  template: '<span>{{ hello }}</span>'
})
export class App {
  public hello: string = 'Hello';
}
```

And now let’s move to tests. First, you must place the tests somewhere in a directory structure. One way is to keep unit tests as close to the code as possible, and the other is to keep all tests in separate directory. We’ll follow the first rule as it seems a better for components to have everything connected to single components in one directory (template, logic, styles etc.). However there’s no strictly better solution in fact. Each one has good and bad parts. The choice is yours, but remember, stick to the rules you’ve created. Following chosen convention is actually more important than the choice itself.

If we have a file like `app.component.ts`, which contains the code from the snippet above, then we would create a file named `app.component.spec.ts` or `app.component.test.ts`. Put the following in there:

```
import { App } from './app.component';

describe('App', () => {

  it('should have hello property', () => {
    this.app = new App();
    expect(this.app.hello).not.toBe('Hello');
  });

});
```

> It’s good to write failing tests first and then make it passing – even when no TDD is introduced. It’s just about making sure that a particular test is running and it is not a false positive. In Jasmine you can see a statement like `expect(...).not.toBe(...)`.

That’s pretty much it! We are now able to test Services, Pipes, basic Components and everything that is a JavaScript class.

Let’s move forward and add a method to the component:

```
@Component({
  selector: 'app',
  template: '<span>{{ sayHello() }}</span>'
})
export class App {
  public name: string = 'John';

  sayHello(): string {
    return `Hello ${this.name}`;
  }
}
```

And the actual test looks like this:

```
describe('App', () => {

  beforeEach(() => {
    this.app = new App();
  });

  it('should have name property', () => {
    expect(this.app.name).toBe('John');
  });

  it('should say hello with name property', () => {
    expect(this.app.sayHello()).toBe('Hello John');
  });

});
```

Services

As far as we know, services are nearly pure classes. What makes them different is the ability to inject and be injected. But still it makes them the easiest to start with. Let's start with the very beginning:

```
class TestService {
  public name: string = 'Injected Service';
}
```

We can go ahead and use something you've already learned:

```
describe('TestService', () => {

  beforeEach(() => {
    this.testService = new TestService();
  });

  it('should have name property set', () => {
    expect(this.testService.name).toBe('Injected Service');
  });

});
```

Pure service tests could look just like that. But consider a module providers declaration part: `providers: [TestService]`. What it really does is to provide the service to be ready to be injected through the whole App. It means every time we need that service we can simply use the DI. Things get a little bit more complicated when we want to test such a behavior. This is what is about to happen:

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

Note that all we have additional imports for testing which comes from `@angular/core/testing` not Jasmine. This is because Angular 2 relies heavily on Dependency Injection, and this is not something that Jasmine is aware of. The Angular team made wrappers that add their own logic. The outcome is we can now use `inject()` instead of an anonymous callback function in `it` (or `beforeEach`) that will inject some class.

The test that conforms the bootstrapping with a given provider would look like the following:

```
describe('TestService Injected', () => {

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [TestService]
    });
  });

  it('should have name property set', inject([TestService], (testService: TestService) => {
    expect(testService.name).toBe('Injected Service');
```

```
  }));
});
```

This way we should end up with a tested injected service. We'll see a `TestBed` and `inject` much more over the time. The first one contains of all of the methods needed to mock and create parts of the app for testing while the latter allows to use dependency injection in tests.

One more thing will be required now. We have to set providers for tests to be able to run in the real browser:

```
import { BrowserDynamicTestingModule, platformBrowserDynamicTesting } from '@angular/platform-browser-dynam

TestBed.initTestEnvironment(BrowserDynamicTestingModule, platformBrowserDynamicTesting());
```

It should be done once per whole test environment.

Mocking Providers

So far we haven't solved the problem of Dependency Injection. The last piece is mocking. The whole idea behind the Angular testability is to replace the real provider with a mocked one to ensure that tests are isolated. Mocking is really straightforward:

```
class MockTestService {
  public mockName: string = 'Mocked Service';
}

describe('TestService Mocked', () => {

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [{ provide: TestService, useClass: MockTestService }]
    });
  });

  it('should have name property set', inject([TestService], (testService: TestService) => {
    expect(testService.mockName).toBe('Mocked Service');
  }));

});
```

In the test case code doesn't know what exactly the `TestService` is. It simply doesn't care – it's completely transparent. This way we can test the behavior of the component itself, not its dependencies. Moreover, we can use something from both worlds – the real one and the mocked (if really needed) using simple JavaScript inheritance:

```
class MockTestServiceInherited extends TestService {
  public sayHello(): string {
    return this.name;
  }
}

describe('TestService Mocked Inherited', () => {

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [{ provide: TestService, useClass: MockTestServiceInherited }]
    });
  });

  it('should say hello with name', inject([TestService], (testService: TestService) => {
    expect(testService.sayHello()).toBe('Injected Service');
  }));

});
```

Components

Injecting services themselves to test suites is not what you really need. They can be tested using pure classes (unless you want to inject service into service). The point is they gives the possibility to fully test a component. But let's leave it for now and grab some component.

Let's say that our component is a list of items (e.g. tasks). It's a dumb component – all it does is rendering a list for given input. It's as simple as this:

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

@Component({
  selector: 'list',
  template: '<span *ngFor="let task of tasks">{{ task }}</span>',
})
class ListComponent {
  public tasks: string[] = [];
}
```

Angular 2 forces us to do a little bit now to create such a component in test:

```
describe('ListComponent', () => {

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [ListComponent]
    });

    this.fixture = TestBed.createComponent(ListComponent);
  });

  it('should render list', async(() => {
    const element = this.fixture.nativeElement;
    this.fixture.componentInstance.tasks = ['Learn ng2'];
    this.fixture.detectChanges();
    expect(element.querySelectorAll('span').length).toBe(1);
  }));

});
```

We don't want to go in depth when it comes to the provided API, but you should really know these methods of `ComponentFixture`:

- `nativeElement` - allows to access DOM
- `componentInstance` - allows to access properties of a component
- `detectChanges` - force Angular to run change detection

Moreover, `async` allows us to have `Promise` in a test suite. This way we can not only wait for promises in test cases, but Angular can also use its Zone. The rest is pretty straightforward. We set array to be some mocked value, say to Angular to detect changes and then check if everything rendered successfully.

As part of successful rendering we should also test events attached inside a particular component. Now consider a component, which is just listening to the `click` event:

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

@Component({
  selector: 'click-handler',
  template: '<button (click)="sendEvent()"></button>',
})
class ClickHandlerComponent {
  public sendEvent(): void {
      // do nothing
  }
}
```

To test it we need to introduce the `spyOn` function that tracks the function invocations. You should do the following:

```
describe('ClickHandlerComponent', () => {

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [ClickHandlerComponent]
    });

    this.fixture = TestBed.createComponent(ClickHandlerComponent);
  });

  it('should invoke a method when clicked', async(() => {
    spyOn(this.fixture.componentInstance, 'sendEvent');
    const element = this.fixture.nativeElement;
    const clickableElement = element.querySelector('button');
    clickableElement.click();
    this.fixture.detectChanges();
    expect(this.fixture.componentInstance.sendEvent).toHaveBeenCalled();
  }));

});
```

Now we are sure that the event handler was attached successfully and the method has been called properly. Further, we can use more advanced *Jasmine* matchers to check whether methods were invoked with proper arguments and so on.

Event Emitters and Observables

Many of the components in your app will be just a dumb components. It means they only have to know about the output and the input. Testing `@Input()` is nothing more just setting proper value. With `@Output` we do have a similar thing. In unit testing we don't really want to test internal Angular 2 functions, we just want to make sure our code is working properly. We'll cover it according to the previous example. Consider the following component:

```
import { Component, EventEmitter, Input, Output } from '@angular/core';

@Component({
  selector: 'click-handler',
  template: '<button (click)="sendEvent()"></button>',
})
class ClickHandlerComponent {
  @Output() public onButtonClicked: EventEmitter<boolean> = new EventEmitter<boolean>();

  public sendEvent(): void {
    this.onButtonClicked.emit(true);
  }
}
```

The point is to test whether `onButtonClicked` is emitting a value each time `sendEvent` is invoked. To achieve this we can inject component and subscribe to the event emitter:

```
describe('ClickHandlerComponent', () => {

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [ClickHandlerComponent]
    });

    this.fixture = TestBed.createComponent(ClickHandlerComponent);
  });

  it('should emit event when a sendEvent method is called', async(() => {
    this.fixture.componentInstance.onButtonClicked.subscribe((event: boolean) => {
      expect(event).toBe(true);
    });
    this.fixture.componentInstance.sendEvent();
  }));

});
```

Now we're using the jasmine async version of the `it` wrapped into `async` to achieve the desired result. Usage of `EventEmitter` implies such a syntax as it's no more a `Promise` to be resolved. So the first thing inside the actual test is to subscribe to the emitter and then invoke a method that should use that emitter. Note that we're not using `@Output()` anywhere in the test. We're just checking the emitter behavior.

Pipes

Every entity in Angular 2 Pipe is also a class with proper annotation. The nice thing about Pipes is they are very small, easy to understand, and easy to test. Let's say we have to truncate some text and implement a pipe for that:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'truncate'
})
class TrancatePipe implements PipeTransform {
  public transform(value: string): string {
    return value.length > 10 ? `${value.substring(0, 10)}...` : value;
  }
}
```

The purpose of the above pipe is to cut everything after the 10th letter and add an ellipsis. To make a proper test we can use what we've learned so far. Pipe is just a specific class. So all we need is to test the `transform` method:

```
describe('TrancatePipe', () => {

  beforeEach(() => {
    this.pipe = new TrancatePipe();
  });

  it('should truncate long text', () => {
    expect(this.pipe.transform('very long text')).toBe('very long ...');
  });

  it('should NOT truncate short text', () => {
    expect(this.pipe.transform('short one')).toBe('short one');
  });

});
```

The fact that a given pipe is stateless and allowed as to be initialized once not in beforeEach. Ideally, unit tests should be as separated as possible, but it sometimes conflicts with DRY (Don't Repeat Yourself) so we strongly advise you have balance there. Sometimes it's worth it to copy some init in a test due to the readability and reasoning about the code.

Router

Everything above is fine up to the point when you meet `Router`. This one requires to add some additional steps to make tests work. Let's start with the following snippet:

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

import { ChildComponent } from './components/child.component';

@Component({
  selector: 'app',
  template: ``,
})
class AppComponent {}

const AppRoutes: Routes = ([
  { path: '/', component: ChildComponent }
]);
```

Note, we just have added empty template. Test for that component would start with this:

```
describe('AppComponent', () => {

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [
        AppComponent
      ]
    });
  });

  it('should be able to test', async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    fixture.whenStable().then(() => {
      expect(true).toBe(true);
    });
  }));

});
```

It's still fine. But the problem is that `Routes` is just exposed to the module and doesn't make our component work as expected. We have to add a place where the component for the given path will be rendered. There's a directive for that called `RouterOutlet`. Here it goes:

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

import { ChildComponent } from './components/child.component';

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

const AppRoutes: Routes = ([
  { path: '/', component: ChildComponent }
]);
```

Now our test fails with:

```
'router-outlet' is not a known element
```

To fix that we have to provide a fake router. To achieve this we should import `RouterTestingModule` and simply add it into the `TestBed` imports in `beforeEach` section:

```
describe('AppComponent', () => {

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [AppComponent],
      imports: [RouterTestingModule]
    });
  });

  it('should be able to test', async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    fixture.whenStable().then(() => {
      expect(true).toBe(true);
    });
  }));

});
```

As you can see, it is pretty straightforward to satisfy the app about the router. The problem really begins when you want to test the routing itself.

HTTP

The common pattern is to keep a connection with the back-end outside the component and simply not to inject `http` directly into the component. The right place seems to be a service. So let's create one:

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

@Injectable()
export class TestService {
  constructor(private http: Http) {}

  getUsers() {
    return this.http.get('http://foo.bar');
  }
}
```

You know pretty much everything that happened right now, so let's move to the test. These imports will be useful:

```
import { BaseRequestOptions, Response, ResponseOptions } from '@angular/http';
import { MockBackend, MockConnection } from '@angular/http/testing';
```

Now, let's use it and prepare proper services to be injected:

```
beforeEach(() => {
  TestBed.configureTestingModule({
    providers: [
      TestService,
      BaseRequestOptions,
      MockBackend,
      {
        provide: Http,
        useFactory: (backend: MockBackend, defaultOptions: BaseRequestOptions) => {
          return new Http(backend, defaultOptions);
        },
        deps: [MockBackend, BaseRequestOptions]
      },
    ]
  });
});
```

One thing to notice is the `Http` mock. Each time when something needs the `Http` we are using our concrete implementation. Moreover, we are injecting dependencies into the factory. It's worth it to add that in case factory returns an instance of class it won't be a singleton.

With mocked `Http` there's one thing left before the actual test. We've mocked all HTTP calls to return a simple string:

```
beforeEach(inject([MockBackend], (backend: MockBackend) => {
  const baseResponse = new Response(new ResponseOptions({ body: 'got response' }));
  backend.connections.subscribe((c: MockConnection) => c.mockRespond(baseResponse));
}));
```

And finally the test itself:

```
it('should return response when subscribed to getUsers',
  inject([TestService], (testService: TestService) => {
    testService.getUsers().subscribe((res: Response) => {
      expect(res.text()).toBe('got response');
    });
  })
);
```

We just tested that every time we call `getUsers` it makes an HTTP call.

Summary

Testing Angular 2 is still in an experimental stage. Most of the things in tests are related to `TestBed`, which gives you the possibility to control real components of the app in their dependencies to make tests separated to each other. However, we have been given a complete solution that should work the way it was meant to. It's highly recommended to use `@angular/testing` package features instead of any custom solutions in this field.

Next up is our final chapter on Migration.

Chapter 14. Migration

Angular 2.0 is a complete rewrite of Angular 1.x, as widely explained throughout the entire book. This raises the question of what to do with our “old” Angular 1.x applications (some of them may be two years old or less).

Building the same applications from scratch is sometimes impossible due to time and priority constraints. Therefore, the Angular team has come with a way to enable us with the incremental migration of our application.

Throughout the migration stage, the application is able to work with both Angular 1.x and Angular 2.0 frameworks, while we can migrate our components / services / filters one by one without hurting our application’s functionality.

Preparation

To start with the migration process we first need to prepare our Angular 1.x application.

Follow the Angular 1.x styleguide

The Angular 1.x styleguide is widely explained at the following address: https://github.com/johnpapa/angular-styleguide/blob/master/a1/README.md

At this moment, we'll take some of these recommendations, the ones that help us the most in our migration process.

1. Single responsibility - Have one component in one file. Moreover, each component / service / controller / directive etc. has to be within its own module. Try not to register multiple components in a single module, like it was common in early stages of Angular 1.x.
2. Small functions - No function should be more than 75 lines of code long.
3. IFEE - Use IFEE for small scopes. Don't "litter" the global scope.
4. Avoid naming collision - Use namespacing and unique names for your components.
5. When getting or setting modules, don't assign them to variables.
6. Prefer named functions over anonymous ones.
7. Use *controllerAs* syntax. Do not assign methods and members directly to the *$scope*.
8. Place your bindable members on top of your controller.
9. Prefer function declaration over fuction expression - remember that functions declaration are hoisted.
10. Move logic of services - keep controllers slim.
11. Assign controllers to routes. Avoid dynamic routes.
12. On services, use *this* keyword, and don't return the object.
13. Return promises from data calls.
14. On directives, limit one per file.
15. Manipulate DOM only in a directive.

Using module loader

Since Angular 2.0 uses a module loader built in, your migration will be much easier if you use a module loader now. You may have your *tsconfig.json* file for your Angular 2.0 application set to use the same module loader.

“Typescriptify” your code

Suprising as it sounds, this task is not so hard. *Typescript* is in fact a superset of JavaScript, so with a little change you will be able to have your Angular 1.x application built with *TypeScript*.

In Chapter 2, you have seen *Typescript* in action. Asumming you are familiar with *tsc* and with the *tsconfig.json* file, all you need to do is to implement them in your Angular 1.x application. You may just change the file extension of each .js file to .ts and try to compile it.

However, after doing so, you may want to use *Typescript* features and not just change the extension. Let’s see what you can do.

1. Each controller, service, component, directive etc. can be a *Typescript* class.
2. Each prototype member can be a class method.
3. On services, each method assigned as a member on the service instance may be a class method.
4. A component’s controller (Angular 1.5) may be a class itself.
5. Its prototype members are class members, and the same as controllers and services.
6. Precede the Every class with the *Export* keyword so you will be able to import it later on.
7. If you already use a module loader, set your *Typescript* compiler to use it (in *tsconfig.json*), and now you may use the *Typescript import* keyword whenever you need.
8. You need also to install the Angular 1.x type definition, along with the jQuery type definition and Angular route type definition. You may machine all of these by running the following command:

```
  npm run typings install dt~jquery dt~angular dt~angular-route \
dt~angular-resource dt~angular-mocks dt~angular-animate \
dt~jasmine -- --save --global
```

For example, let’s see how we can alter a controller code.

```
(function(angular){
function DashboardController(socketControll, dataService) {
        this._socketControll = socketControll;
        this._data = dataService;
         this.users = _data.users;
          _socketControl.validateAndOpenListeners();
          }



}

angular.module('edge-app.dashboard',['edge-app.scocketControll', 'edge-app.dataService']).controller('dashBc
}(window.angular));
```

Now after *Typescript* migration:

```
(function(angular){
class DashboardController{
   users: any;
   _socketControll: any;
   _data: any;

   constructor(socketControll, dataService) {
        this._socketControll = socketControll;
        this._data = dataService;
         this.users = _data.users;
          _socketControl.validateAndOpenListeners();
          }




}

angular.module('edge-app.dashboard',['edge-app.scocketControll', 'edge-app.dataService']).controller('dashBo
['socketControllService', 'dataService',DashboardController]);
}(window.angular));
```

Let’s have a look at a service:

```
(function(angular){
 function DataService(api){
       this._api = api;
    }

    DataService.prototype.getAllData = getAllData;
    DataService.prototype.getUsers = getUsers;

    function getAllData() {
        return Promise.all([this.getRooms(), this.getUsers()])
    }

    function getRooms() {
        return this._api.send('getRooms').then(function(res)
               {
                    this.rooms = res.data;

                });
        }


    function getUsers() {
        return this._api.send('getUsers').then(function(res)
                 {
                    this.users = res.data;

                });


    }


    angular.module('edge-app.services.dataService',
    ['edge-app.services.apiService']).service('edge-app.dataService',
    ['adge-app.apiService', DataService]);

}(window.angular));
```

Now, let’s migrate this to Typescript:

```
(function(angular){
 class DataService {
    constructor(
        private _api: any
    ) {}
```

```
    public rooms: any;
    public users: any;

    getAllData() {
        return Promise.all([this.getRooms(), this.getUsers()])
    }

    getRooms() {
        return new Promise((resolve, reject) => {
            this._api.send('getRooms').subscribe(
                res => {
                    this.rooms = res.data;
                    resolve(res)
                },
                err => reject(err)
            )
        })
    }

    getUsers() {
        return new Promise((resolve, reject) => {
            this._api.send('getUsers').subscribe(
                res => {
                    this.users = res.data;
                    resolve(res)
                },
                err => reject(err)
            )
        })
    }

}

 angular.module('edge-app.services.dataService',
 ['edge-app.services.apiService']).service('edge-app.dataService',
 ['adge-app.apiService', DataService])
 ;

 }(window.angular));
```

Migrating

In this section we will actually blend in the Angular 2.0 boilerplate to have an Angular 2.0 application. Then we will use the *UpgradeAdapter* from the *'@angular/upgrade'* module.

We will add Angular 2.0 packages to our project, the same ones that are in our chat application. We will also add the *system.config.js* file to our project. This will not affect our Angular 1.x applcation, yet.

Using *UpgradeAdapter*

The *UpgradeAdapter* class is the class that will help blend Angular 1.x and Angular 2.0 into one application built on both foundations. We will do it by bootstrapping our application in a slightly different way.

You may have used the *ng-app* directive on your HTML file, or the angular.boostrap method. To bootstrap the Angular 1.x application so it can work with Angular 2.0, we must in our main.ts file create an instance of the UpgradeAdapter.

With that instance, we will bootstrap our application module:

```
let upgradeAdapter = new UpgradeAdapter();

upgradeAdapter.bootstrap(document.documentElement, ['edgeAppModule']);
```

Note: The difference here is the use of upgradeAdapter.bootstrap instead of angular.bootstrap.

Making Angular 1.x service available to Angular 2.0

UpgradeAdapter also helps us make an Angular 1.x service available to Angular 2.0.

In this example we wish to use the old Angular routing module's to get route parameters, therefore we need the *$routeParams* service.

```
upgradeProvider.upgradeNg1Provider('$routeParams');
```

And vise versa:

```
angular.module('edgeApp').controller('edgeAppController',
  upgradeAdapter.downgradeNg2Provider(YourProviderType));
```

Now your service is almost set to work with Angular 1.x, but one thing you still need to do is add it to your provider's list. Remember that unlike Angular 1.x, services in Angular 2.0 aren't singleton, but an instance is created whenever they're in the provider's list. To create an instance of them available to Angular 1.x you need to use *addProvider* method.

Here's an example for adding HTTP client of Angular 2.0 to Angular 1.x.

```
upgradeAdapter.addProvider(HTTP_PROVIDERS);
```

Now we can work with Angular 1.x and Angular 2.0 side by side. What has been left is to change the code from Angular 1.x to Angular 2.0.

Migrating a component

The Angular 1.5 component has a template (or template URL), controller and binding object. The method of migrating it to Angular 2.0 component is almost straight forward:

1. The template stays the same and is moved into the template member of the *ViewMetaData* object being sent to the *Component* decorator. The data binding, of course, is changed to the Angular 2.0 syntax.
2. The controller object is the class being exported. We need to replace the controller function to class (if haven't done so already) and export it.
3. All binding members are now component class members decorated with the *Input* decorator.

Migrating a directive

When dealing with an attribute directive, the migration process is a little different than the component, but not by far.

1. The link method can be copied into the directive constructor.
2. If you're using *$scope.$watch* it is highly recommended that you switch to properties. You must do so in Angular 2.0, because there is no `$scope.$watch` there.
3. Scope members are now class members decorated with `@Input()`.
4. The directive name is now on the *selector* member of the metadata object being sent to the *Directive* decorator. Please note that there is no difference between what is put in the decorator and the directive usage besides the square brackets for the attribute.

For the Angular 1.x component directive (the one used before Angular 1.5) the process is almost like the component process, but scope members relace the binding members.

Migrating a filter (to pipe)

1. In filters, the function being returned needs to go to the pipe's *transform* method, while the pipe class implements the *PipeTransform* interface. There are no changes in the arguments order.
2. Decorate your class with the *Pipe* decoratior and give the filter name to the *name* member of the metadata object.

Migrating a service

The process of service migration is the easiest one, note that if you have migrated your service to be written in *Typescript*, all you need to do is to skip registering it, and have it in the providers array of your app's main component.

1. The service function becomes an exported class decorated with *Injectable*.
2. All function members are class members.
3. Register your service in the providers array of your app main component.
4. Remove the angular.module(...).service(...) registration.

Summary

Angular 2.0 has been entirely rewritten. The Angular team has decided to do so after Angular 1.x had performance issues. 1.x was built on legacy standards and there was a need to go further with web technology that has been changing rapidly over the past few years.

Angular 2.0 is built on the newest standards of modern web browesrs. It supports HTML5 web components. Its change detection has improved significantly based on rxJs and zoneJs. It reduced significantly the change detection cycles by detecting changes only top down, making us using events when we actually know when a change has occurred.

Although it is possible to write an Angular 2.0 application in ES5, ES6 and dart. Typescript is known to be the cleanest JavaScript transpiler there is. Moreover, you will see that Typescript is the most documented language in the angular.io site. Therefore, you may choose your favorite scripting language, because our application is built on Typescript.

In this book we've seen angular 2.0 building blocks such as components, directives, injectable services and pipes. We have become familier with the new routing mechanism and forms. We have also elaborated on change detection and RxJs, which has brought us the Observable type for better change detection.

However, since Angular 1.x was very popular, there were many concerns about moving ahead to the next Angular version, because at the beginning there was no migration path. However, complying with the Angular 1.x styleguide that has been started written on July 2014, you can have an Angular 1.x application that can be easily migrated to Angular 2.0. The Angular team has developed the `UpgradeAdapter` module that can assist us in working with both Angular 1.x and Angular 2.0 code in the same application. This may cause our application to work slower than it would working solely with Angular 2.0, however, it can buy us a significant time for migration without compromising our functionality.

Preface

1. Angular 2 Introduction
    - AngularJS
    - Angular 2
    - Performance
    - Ecosystem
    - Is Angular 2 the right choice for my project?

2. Choosing your scripting language
    - What is TypeScript?
    - History of TypeScript
    - Features of TypeScript
        - Type Annotationa
        - Type Inference
        - Type Erasure
        - Interfaces
        - Enumerated Types
        - Generics
        - Mixins
        - Tuple Types
    - What is ES6?
    - History of ECMAScript
    - Features of ES6
        - Classes
        - Enhanced Object Literals
        - Arrows
        - Template Strings
        - Destructuring
        - Generators
        - Modules
        - Symbols
        - For...Of loops
        - Map + Set + WeakMap + WeakSet
        - Proxies

        - Promises
    - What are shims?
    - What are polyfills?
    - Should You Use TypeScript or ECMAScript?
    - Summary

3. Introducing Our App
    - Folder Structure
    - Application Setup
    - Hello World
    - Template Syntax
        - Interpolation
        - Square bracket syntax
        - Event Binding
        - Two-way Data Binding
    - Built-in Directives
        - NgIf
    - Introduction To Services
        - Dependency Injection
    - Basic Routing
    - RouterConfig
    - RouterModule.forRoot()
        - RouterLink
    - Angular CLI
        - Installing Angular CLI
        - Creating your Angular 2 application using CLI
        - Build & Serve
        - Creating a model & service
        - Router
        - Final thoughts on Angular CLI

4. Modules
    - What is a module?
    - Module declaration

- Why Modules again?
  - What does compilation means?
- Compiling in build time
  - TreeShaking
- Use JiT for development and AoT for production
- Summary

5. Components
   - Component decorator
   - Component Usage:
   - Using injectable services
   - Pipes
     - Built in pipes
     - Writing custom pipes
   - Directives
   - Inter-component communication
     - Using Events
   - Component lifecycle
   - Summary

6. Directives
   - Attribute Directives
     - Replicating NgClass
   - Structural Directives
   - Summary

7. Services
   - HTTP and Observables
   - Communicating with local storage and event emitters in services
   - Importing external libraries
   - Summary

8. Pipes
   - Built-in Pipes

        - String -> String
        - Number -> String
        - Object -> String
        - Misc
    - Custom Pipes
        - Create pipes
        - Use custom pipes

9. Routing
    - Configuration
    - Router Outlet
    - Router Link
    - Route Parameters
    - Accessing the Router state
    - Resolve
    - Guards
    - Conclusion

10. Forms
    - FormControls
    - FormGroup
    - Your First Form
    - FormsModule & ReactiveFormsModule
    - The Component
    - The Template
    - FormBuilder and form
    - Formbuilder in Your Views
    - Validators
        - Form Messages
        - Field Message
        - Field Coloring
        - Specific Validation
    - Personalized Validations
        - Writing a Personalized Validator
    - ngModel, the directive

Summary

11. Change Detection

What's Change Detection?
Changes and Events
Zones
Change Detectors and Components

OnChanges

Change Detection Tuning

Change Detection Strategy
OnPush Strategy
Mutable and Immutable
Using Observable

Summary

12. RxJS

What is Observable?
Hot and cold observables
Reactive Angular 2

Event streams flow
HTTP
Forms

Summary

13. Testing

First tests
Services
Mocking Providers
Components
Event Emitters and Observables
Pipes
Router
HTTP
Summary

14. Migration
- Preparation
  - Follow the Angular 1.x styleguide
  - Using module loader
  - “Typescriptify” your code
- Migrating
  - Using UpgradeAdapter
  - Making Angular 1.x service available to Angular 2.0
  - Migrating a component
  - Migrating a directive
  - Migrating a filter (to pipe)
  - Migrating a service
- Summary