Become a

NINJA

with

ANGULAR 2

ninja squad

Become a ninja with Angular2

Ninja Squad

Table of Contents

1. Introduction 1
2. A gentle introduction to ECMASCRIPT 6 3
    2.1. Transpilers 3
    2.2. let 4
    2.3. Constants 5
    2.4. Creating objects 6
    2.5. Destructuring assignment 6
    2.6. Default parameters and values 8
    2.7. Rest operator 10
    2.8. Classes 11
    2.9. Promises 13
    2.10. Arrow functions 17
    2.11. Sets and Maps 20
    2.12. Template literals 21
    2.13. Modules 21
    2.14. Conclusion 23
3. Going further than ES6 24
    3.1. Dynamic, static and optional types 24
    3.2. Enters TypeScript 25
    3.3. A practical example with DI 25
4. Diving into TypeScript 28
    4.1. Types as in TypeScript 28
    4.2. Enums 29
    4.3. Return types 29
    4.4. Interfaces 30
    4.5. Optional arguments 30
    4.6. Functions as property 31
    4.7. Classes 31
    4.8. Working with other libraries 33
    4.9. Decorators 34
5. The wonderful land of Web Components 37
    5.1. A brave new world 37
    5.2. Custom elements 37
    5.3. Shadow DOM 38
    5.4. Template 39
    5.5. HTML imports 40

5.6. Polymer and X-tag 40
6. Grasping Angular's philosophy 43
7. From zero to something 47
7.1. Developing and building a TypeScript app 47
7.2. Our first component 49
7.3. Bootstrapping the app 51
7.4. From zero to something better with angular-cli 54
8. The templating syntax 56
8.1. Interpolation 56
8.2. Using other components in our templates 60
8.3. Property binding 62
8.4. Events 65
8.5. Expressions vs statements 69
8.6. Local variables 69
8.7. Structural directives 70
8.8. Other template directives 74
8.9. Canonical syntax 75
8.10. Summary 75
9. Dependency injection 79
9.1. DI yourself 79
9.2. Easy to develop 79
9.3. Easy to configure 82
9.4. Other types of provider 86
9.5. Hierarchical injectors 89
9.6. Binding multiple values 91
9.7. DI without types 92
10. Pipes 93
10.1. Pied piper 93
10.2. json 93
10.3. slice 94
10.4. uppercase 96
10.5. lowercase 96
10.6. number 96
10.7. percent 97
10.8. currency 98
10.9. date 98
10.10. async 99
10.11. Creating your own pipes 100

11. Reactive Programming . . . 103
11.1. Call me maybe . . . 103
11.2. General principles . . . 103
11.3. RxJS . . . 104
11.4. Reactive programming in Angular 2 . . . 106
12. Building components and directives . . . 108
12.1. Introduction . . . 108
12.2. Directives . . . 108
12.3. Components . . . 120
12.4. Making a component available everywhere . . . 122
13. Services . . . 124
13.1. Title service . . . 124
13.2. Making your own service . . . 124
14. Testing your app . . . 126
14.1. The problem with troubleshooting is that trouble shoots back . . . 126
14.2. Unit test . . . 126
14.3. Fake dependencies . . . 132
14.4. Testing components . . . 133
14.5. Testing with fake templates, directives... . . . 137
14.6. End-to-end tests (e2e) . . . 138
15. Forms . . . 140
15.1. Forms, dear forms . . . 140
15.2. Model-driven . . . 142
15.3. Template-driven . . . 147
15.4. Adding some validation . . . 152
15.5. Errors and submission . . . 154
15.6. Add some style . . . 158
15.7. Creating a custom validator . . . 160
15.8. Grouping fields . . . 164
15.9. Reacting on changes . . . 166
15.10. Summary . . . 168
16. Send and receive data with Http . . . 169
16.1. Getting data . . . 169
16.2. Transforming data . . . 171
16.3. Advanced options . . . 172
16.4. Jsonp . . . 173
16.5. Tests . . . 174
17. Router . . . 176

17.1. En route 176
17.2. Navigation 178
18. This is the end 180

Chapter 1. Introduction

So you want to be a ninja, huh? Well, you're in good hands!

But we have a long road, you and me, with lots of things to learn :).

We're living exciting times in Web development. There is a new Angular. A complete rewrite of the good old AngularJS. Why a complete rewrite? Was AngularJS 1.x not enough?

I like the old AngularJS very much. In our small company, we have built several projects with it, contributed code to the core framework, trained hundred of developers (yes, really), and even written a book about it (in French, but that still counts).

AngularJS is incredibly productive once you have mastered it. Despite all of this, it doesn't prevent us from seeing its weaknesses. AngularJS is not perfect, with some very difficult concepts to grasp, and traps hard to avoid.

Most of all, the Web has changed since AngularJS was conceived. JavaScript has changed. New frameworks have emerged, with great ideas, or better implementation. We are not the kind of developers to tell you that you should use this tool instead of that one. We just happen to know some tools very well, and know what fits the project. AngularJS was one of those tools, allowing us to build well-tested web applications, and to build them fast. We also tried to bend it where it didn't fit. Don't blame us, it happens to the best of us.

Will Angular 2 be the tool we will use without hesitation in our future projects? It's hard to say right now, because the framework is really young and the ecosystem only just blooming.

But Angular 2 has a lot of interesting points, and a vision that few other frameworks have. It has been designed for the Web of tomorrow, with ECMAScript 6, Web Components and Mobile in mind. When it was first announced, I was, like many, sad at first that the 2.0 version would not be a simple update (I'm sorry if you're just learning about it).

But I was also eager to see what solution the talented Google team would come up with.

So I started to write this ebook, pretty much after the first commits, reading the design docs, watching the conference videos, reviewing every commit since the beginning. When I wrote my first ebook, about AngularJS 1.x, it was already a stable and known beast. This one is very different, it started when Angular 2 was not even clear in the minds of its designers. Because I knew I would learn a lot, not only about Angular but also about the concepts that would shape the future of Web development, some of which have nothing to do with Angular. And I did. I had to dig a lot about some of these concepts, and I hope that you will enjoy the journey of learning about them, and how they relate to Angular, as much as I did.

The ambition of this ebook is to evolve with Angular. If it turns out that Angular is the great framework we hope, you will receive updates with the best practices and some new features as they emerge (and with less typos, because, despite our countless reviews, there are probably some left…).

And I would love to hear back from you - if some chapters aren't clear enough, if you spot a mistake or if you have a better way for some parts.

I'm fairly confident about the code samples, though, as they are all in a real project, with several hundred unit tests. It was the only way to write an ebook with a newborn framework, and to be able to catch all the problems that inevitably arose with each release.

Even if you are not convinced by Angular in the end, I'm pretty sure you will have learnt a thing or two along your read.

If you have bought the "Pro package" (thank you!), you'll build a small application piece by piece along the book. This application is called **PonyRacer**, and it is a website where you can bet on pony races. You can even test the application here! Go on, I'll wait for you.

Fun, isn't it?

But it's not just a fun application, it's a complete one. You'll have to write components, forms, tests, use the router, call an HTTP API (that we have already built) and even do Web Sockets. It has all the pieces you'll need for writing a real app. Each exercise will come with a skeleton, a set of instructions and a few tests. Once you have all the tests in success, you have completed the exercise!

If you did not buy the "Pro package" (but really you should), don't worry: you'll learn everything that's needed. But you will not build this awesome application with beautiful ponies in pixel art. Your loss :)!

You will quickly see that, beyond Angular itself, we have tried to explain the core concepts the framework uses. The first chapters don't even talk about Angular: they are what I call the "Concept Chapters", here to help you level up with the new and exciting things happening in our field.

Then we will slowly build our knowledge of the framework, with components, templates, pipes, forms, http, routing, tests…

And finally we will learn about the advanced topics. But that's another story.

Enough with the introduction, let's start with one of the things that will definitely change the way we code: ECMAScript 6.

Chapter 2. A gentle introduction to ECMASCRIPT 6

If you're reading this, we can be pretty sure you have heard of JavaScript. What we call JS is one implementation of a standard specification, called ECMAScript. The spec version you know the most about is the version 5, that has been used these last years.

But recently, a new version of the spec has been in the works, called ECMASCRIPT 6, ES6, or ECMASCRIPT 2015. From now on, I'll mainly say ES6, as it is the most popular way to reference it. It adds A LOT of things to JavaScript, like classes, constants, arrow functions, generators... It has so much stuff that we can't go through all of it, as it would take the whole book. But Angular 2 has been designed to take advantage of the brand new version of JavaScript. And, even if you can still use your old JavaScript, things will be more awesome if you use ES6. So we're going to spend some time in this chapter to get a grip on what ES6 is, and what will be useful to us when building an Angular app.

That means we're going to leave a lot of stuff aside, and we won't be exhaustive on the rest, but it will be a great starting point. If you already know ES6, you can skip these pages. And if you don't, you will learn some pretty amazing things that will be useful to you even if you end up not using Angular in the future!

2.1. Transpilers

ES6 has just reached its final state, so it's not yet fully supported by every browser. And, of course, some browsers will always be late to this game (do I really need to name it?). You might be thinking: what's the use of all this, if I need to be careful on what I can use? And you'd be right, because there aren't that many apps that can afford to ignore older browsers. But, since virtually every JS developer who has tried ES6 wants to write ES6 apps, the community has found a solution: a transpiler.

A transpiler takes ES6 source code and generates ES5 code that can run in every browser. It even generates the source map files, which allows to debug directly the ES6 source code from the browser. At the time of writing, there are two main alternatives to transpile ES6 code:

- Traceur, a Google project
- Babeljs, a project started by a young developer, Sebastian McKenzie (17 years old at the time, yeah, that hurts me too), with a lot of diverse contributions.

Each has its own pros and cons. For example, Babeljs produces a more readable source code than Traceur. But Traceur is a Google project, so, of course, Angular and Traceur play well together. The source code of Angular 2 itself was at first transpiled with Traceur, before switching to TypeScript. TypeScript is an open source language developed by Microsoft. It's a typed superset of JavaScript that compiles to plain JavaScript, but we'll dive into it very soon.

Let's be honest Babel has waaaay more steam than Traceur, so I would advice you to use it. It is quickly

becoming the de-facto standard in this area.

So if you want to play with ES6, or set it up in one of your projects, take a look at these transpilers, and add a build step to your process. It will take your ES6 source files and generate the equivalent ES5 code. It works very well but, of course, some of the new features are quite hard or impossible to transform in ES5, as they just did not exist. However, the current state is largely good enough for us to use without worrying, so let's have a look at all these shiny new things we can do in JavaScript!

2.2. let

If you have been writing JS for some time, you know that the `var` declaration is tricky. In pretty much any other languages, a variable is declared where the declaration is done. But in JS, there is a concept, called "hoisting", which actually declares a variable at the top of the function, even if you declared it later.

So declaring a variable like `name` in the `if` block:

```
function getPonyFullName(pony) {
  if (pony.isChampion) {
    var name = 'Champion ' + pony.name;
    return name;
  }
  return pony.name;
}
```

is equivalent to declaring it at the top of the function:

```
function getPonyFullName(pony) {
  var name;
  if (pony.isChampion) {
    name = 'Champion ' + pony.name;
    return name;
  }
  // name is still accessible here,
  // and can have a value from the if block
  return pony.name;
}
```

ES6 introduces a new keyword for variable declaration, `let`, behaving much more like what you would expect:

```
function getPonyFullName(pony) {
  if (pony.isChampion) {
    let name = 'Champion ' + pony.name;
    return name;
  }
  // name is not accessible here
  return pony.name;
}
```

The variable `name` is now restricted to its block. `let` has been introduced to replace `var` in the long run, so you can pretty much drop the good old `var` keyword and start using `let` instead. The cool thing is, it should be painless to use `let`, and if you can't, you have probably spotted something wrong with your code!

2.3. Constants

Since we are on the topic of new keywords and variables, there is another one that can be of interest. ES6 introduces `const` to declare... constants! When you declare a variable with `const`, it has to be initialized and you can't assign another value later.

```
const PONIES_IN_RACE = 6;
```

```
PONIES_IN_RACE = 7; // SyntaxError
```

I used a snake_case, FULL CAPS, to name the constant, as it's done in Java. There is no obligation to do so, but it feels natural to have a convention for constants: find yours and stick to it!

As for variables declared with `let`, constants are not hoisted and are only declared at the block level.

One small thing might surprise you: you can initialize a constant with an object and later modify the object content.

```
const PONY = { };
PONY.color = 'blue'; // works
```

But you can't assign another object:

```
const PONY = { };
```

```
PONY = {color: 'blue'}; // SyntaxError
```

Same thing with arrays:

```
const PONIES = [];
PONIES.push({ color: 'blue' }); // works
```

```
PONIES = []; // SyntaxError
```

2.4. Creating objects

Not a new keyword, but it can also catch your attention when reading ES6 code. There is now a shortcut for creating objects, when the object property you want to create has the same name as the variable used as the value.

Example:

```
function createPony() {
  let name = 'Rainbow Dash';
  let color = 'blue';
  return { name: name, color: color };
}
```

can be simplified to

```
function createPony() {
  let name = 'Rainbow Dash';
  let color = 'blue';
  return { name, color };
}
```

2.5. Destructuring assignment

This new feature can also catch your attention when reading ES6 code. There is now a shortcut for assigning variables from objects or arrays.

In ES5:

```
var httpOptions = { timeout: 2000, isCache: true };
// later
var httpTimeout = httpOptions.timeout;
var httpCache = httpOptions.isCache;
```

Now, in ES6, you can do:

```
let httpOptions = { timeout: 2000, isCache: true };
// later
let { timeout: httpTimeout, isCache: httpCache } = httpOptions;
```

And you will have the same result. It can be a little disturbing, as the key is the property to look for into the object and the value is the variable to assign. But it works great! Even better: if the variable you want to assign has the same name as the property, you can simply write:

```
let httpOptions = { timeout: 2000, isCache: true };
// later
let { timeout, isCache } = httpOptions;
// you now have a variable named 'timeout'
// and one named 'isCache' with correct values
```

The cool thing is that it also works with nested objects:

```
let httpOptions = { timeout: 2000, cache: { age: 2 } };
// later
let { cache: { age } } = httpOptions;
// you now have a variable named 'age' with value 2
```

And the same is possible with arrays:

```
let timeouts = [1000, 2000, 3000];
// later
let [shortTimeout, mediumTimeout] = timeouts;
// you now have a variable named 'shortTimeout' with value 1000
// and a variable named 'mediumTimeout' with value 2000
```

Of course it also works for arrays in arrays, or arrays in objects, etc…

One interesting use of this can be for multiple return values. Imagine a function `randomPonyInRace` that returns a pony and its position in a race.

```
function randomPonyInRace() {
  let pony = { name: 'Rainbow Dash' };
  let position = 2;
  // ...
  return { pony, position };
}
let { position, pony } = randomPonyInRace();
```

The new destructuring feature is assigning the position returned by the method to the position variable, and the pony to the pony variable! And if you don't care about the position, you can write:

```
function randomPonyInRace() {
  let pony = { name: 'Rainbow Dash' };
  let position = 2;
  // ...
  return { pony, position };
}
let { pony } = randomPonyInRace();
```

And you will only have the pony!

2.6. Default parameters and values

One of the characteristics of JavaScript is that it allows developers to call a function with any number of arguments:

- if you pass more arguments than the number of the parameters, the extra arguments are ignored (well, you can still use them with the special `arguments` variable, to be accurate).
- if you pass less arguments than the number of the parameters, the missing parameter will be set to `undefined`.

The last case is the one that is the most relevant to us. Usually, we pass less arguments when the parameters are optional, like in the following example:

```
function getPonies(size, page) {
  size = size || 10;
  page = page || 1;
  // ...
  server.get(size, page);
}
```

The optional parameters usually have a default value. The OR operator will return the right operand if the left one is `undefined`, as will be the case if the parameter was not provided (to be completely

accurate, if it is *falsy*, i.e `0`, `false`, `""`, etc.). Using this trick, the function `getPonies` can then be called:

```
getPonies(20, 2);
getPonies(); // same as getPonies(10, 1);
getPonies(15); // same as getPonies(15, 1);
```

This worked alright, but it was not really obvious that the parameters were optional ones with default values, without reading the function body. ES6 introduces a more precise way to have default parameters, directly in the function definition:

```
function getPonies(size = 10, page = 1) {
  // ...
  server.get(size, page);
}
```

Now it is perfectly clear that the `size` parameter will be `10` and the `page` parameter will be `1` if not provided.

NOTE There is a small difference though, as now `0` or `""` are valid values and will not be replaced by the default one, as `size = size || 10` would have done. It will be more like `size = size === undefined ? 10: size;`.

The default value can also be a function call:

```
function getPonies(size = defaultSize(), page = 1) {
  // the defaultSize method will be called if size is not provided
  // ...
  server.get(size, page);
}
```

or even other variables, either global variables, or other parameters of the function:

```
function getPonies(size = defaultSize(), page = size - 1) {
  // if page is not provided, it will be set to the value
  // of the size parameter minus one.
  // ...
  server.get(size, page);
}
```

Note that if you try to access parameters on the right, their value is always `undefined`:

```
function getPonies(size = page, page = 1) {
  // size will always be undefined, as the page parameter is on its right.
  server.get(size, page);
}
```

This mechanism for parameters can also be applied to values, for example when using a destructuring assignment:

```
let { timeout = 1000 } = httpOptions;
// you now have a variable named 'timeout',
// with the value of 'httpOptions.timeout' if it exists
// or 1000 if not
```

2.7. Rest operator

ES6 introduces a new syntax to define variable parameters in functions. As said in the previous part, you could always pass extra arguments to a function and get them with the special `arguments` variable. So you could have done something like that:

```
function addPonies(ponies) {
  for (var i = 0; i < arguments.length; i++) {
    poniesInRace.push(arguments[i]);
  }
}
addPonies('Rainbow Dash', 'Pinkie Pie');
```

But I think we can agree that it's neither pretty nor obvious: since the `ponies` parameter is never used, how do we know that we can pass several ponies?

ES6 gives us a way better syntax, using the rest operator `...`:

```
function addPonies(...ponies) {
  for (let pony of ponies) {
    poniesInRace.push(pony);
  }
}
```

`ponies` is now a true array on which we can iterate. The `for ... of` loop used for iteration is also a new feature in ES6. It allows to be sure to iterate over the collection values, and not also on its properties as `for ... in` would do. Don't you think our code is prettier and more obvious now?

The rest operator can also work when destructuring data:

```
let [winner, ...losers] = poniesInRace;
// assuming 'poniesInRace' is an array containing several ponies
// 'winner' will have the first pony,
// and 'losers' will be an array of the other ones
```

The rest operator is not to be confused with the spread operator which, I'll give you that, looks awfully similar! But the spread operator is the opposite: it takes an array and spreads it in variable arguments. The only examples I have in mind are functions like min or max, that receive variable arguments, and that you might want to call on an array:

```
let ponyPrices = [12, 3, 4];
let minPrice = Math.min(...ponyPrices);
```

2.8. Classes

One of the most emblematic new features, and one that we will vastly use when writing an Angular app: ES6 introduces classes to JavaScript! You can now easily use classes and inheritance in JavaScript. You always could, using prototypal inheritance, but that it was not an easy task, especially for beginners.

Now it's very easy, take a look:

```
class Pony {
  constructor(color) {
    this.color = color;
  }
  toString() {
    return `${this.color} pony`;
    // see that? It is another cool feature of ES6, called template literals
    // we'll talk about these quickly!
  }
}
let bluePony = new Pony('blue');
console.log(bluePony.toString()); // blue pony
```

Class declarations, unlike function declarations, are not hoisted, so you need to declare a class before using it. You may have noticed the special function `constructor`. It is the function being called when we create a new pony, with the `new` operator. Here it needs a color, and we create a new Pony instance with the color set to "blue". A class can also have methods, callable on an instance, as the method `toString()` here.

It can also have static attributes and methods:

```
class Pony {
  static defaultSpeed() {
    return 10;
  }
}
```

Static methods can be called only on the class directly:

```
let speed = Pony.defaultSpeed();
```

A class can have getters and setters, if you want to hook on these operations:

```
class Pony {
  get color() {
    console.log('get color');
    return this._color;
  }
  set color(newColor) {
    console.log(`set color ${newColor}`);
    this._color = newColor;
  }
}
let pony = new Pony();
pony.color = 'red'; // set color red
console.log(pony.color); // get color
```

And, of course, if you have classes, you also have inheritance out of the box in ES6.

```
class Animal {
  speed() {
    return 10;
  }
}
class Pony extends Animal {

}
let pony = new Pony();
console.log(pony.speed()); // 10, as Pony inherits the parent method
```

Animal is called the base class, and Pony the derived class. As you can see, the derived class has the methods of the base class. It can also override them:

```
class Animal {
  speed() {
    return 10;
  }
}
class Pony extends Animal {
  speed() {
    return super.speed() + 10;
  }
}
let pony = new Pony();
console.log(pony.speed()); // 20, as Pony overrides the parent method
```

As you can see, the keyword `super` allows calling the method of the base class, with `super.speed()` for example.

The `super` keyword can also be used in constructors, to call the base class constructor:

```
class Animal {
  constructor(speed) {
    this.speed = speed;
  }
}
class Pony extends Animal {
  constructor(speed, color) {
    super(speed);
    this.color = color;
  }
}
let pony = new Pony(20, 'blue');
console.log(pony.speed); // 20
```

2.9. Promises

Promises are not so new, and you might know them or use them already, as they were a big part of AngularJS 1.x. But since you will use them a lot in Angular 2, and even if you're just using JS, I think it's important to make a stop.

Promises aim to simplify asynchronous programming. Our JS code is full of async stuff, like AJAX requests, and usually we use callbacks to handle the result and the error. But it can get messy, with callbacks inside callbacks, and it makes the code hard to read and to maintain. Promises are much nicer than callbacks, as they flatten the code, and thus make it easier to understand. Let's consider a simple use case, where we need to fetch a user, then its rights, then update a menu when we have these.

With callbacks:

```
getUser(login, function(user) {
  getRights(user, function(rights) {
    updateMenu(rights);
  });
});
```

Now, let's compare it with promises:

```
getUser(login)
  .then(function(user) {
    return getRights(user);
  })
  .then(function(rights) {
    updateMenu(rights);
  })
```

I like this version, because it executes as you read it: I want to fetch a user, then get its rights, then update the menu.

As you can see, a promise is a 'thenable' object, which simply means it has a `then` method. This method takes two arguments: one success callback and one reject callback. The promise has three states:

- pending: while the promise is not done, for example, our server call is not completed yet.
- fulfilled: when the promise is completed with success, for example, the server call returns an OK HTTP status.
- rejected: when the promise has failed, for example, the server returns a 404 status.

When the promise is fulfilled, then the success callback is called, with the result as an argument. If the promise is rejected, then the reject callback is called, with a rejected value or an error as the argument.

So, how do you create a promise? Pretty simple, there is a new class called `Promise`, whose constructor expects a function with two parameters, `resolve` and `reject`.

```
let getUser = function(login) {
  return new Promise(function(resolve, reject) {
    // async stuff, like fetching users from server, returning a response
    if (response.status === 200) {
      resolve(response.data);
    } else {
      reject('No user');
    }
  });
};
```

Once you have created the promise, you can register callbacks, using the `then` method. This method can receive two parameters, the two callbacks you want to call in case of success or in case of failure. Here we only pass a success callback, ignoring the potential error:

```
getUser(login)
  .then(function(user) {
    console.log(user);
  })
```

Once the promise is resolved, the success callback (here simply logging the user on the console) will be called.

The cool part is that it flattens the code. For example, if your resolve callback is also returning a promise, you can write:

```
getUser(login)
  .then(function(user) {
    return getRights(user) // getRights is returning a promise
      .then(function(rights) {
        return updateMenu(rights);
      });
  })
```

but more beautifully:

```
getUser(login)
  .then(function(user) {
    return getRights(user); // getRights is returning a promise
  })
  .then(function(rights) {
    return updateMenu(rights);
  })
```

Another interesting thing is the error handling, as you can use one handler per promise, or one for all the chain.

One per promise:

```
getUser(login)
  .then(function(user) {
    return getRights(user);
  }, function(error) {
    console.log(error); // will be called if getUser fails
    return Promise.reject(error);
  })
  .then(function(rights) {
    return updateMenu(rights);
  }, function(error) {
    console.log(error); // will be called if getRights fails
    return Promise.reject(error);
  })
```

One for the chain:

```
getUser(login)
  .then(function(user) {
    return getRights(user);
  })
  .then(function(rights) {
    return updateMenu(rights);
  })
  .catch(function(error) {
    console.log(error); // will be called if getUser or getRights fails
  })
```

You should seriously look into Promises, because they are going to be the new way to write APIs, and every library will use them. Even the standard ones: the new Fetch API does for example.

2.10. Arrow functions

One thing I like a lot in ES6 is the new arrow function syntax, using the 'fat arrow' operator (⇒). It is SO useful for callbacks and anonymous functions!

Let's take our previous example with promises:

```
getUser(login)
  .then(function(user) {
    return getRights(user); // getRights is returning a promise
  })
  .then(function(rights) {
    return updateMenu(rights);
  })
```

can be written with arrow functions like this:

```
getUser(login)
  .then(user => getRights(user))
  .then(rights => updateMenu(rights))
```

How cool is it? THAT cool!

Note that the return is also implicit if there is no block: no need to write `user ⇒ return getRights(user)`. But if we did have a block, we would need the explicit return:

```
getUser(login)
  .then(user => {
    console.log(user);
    return getRights(user);
  })
  .then(rights => updateMenu(rights))
```

And it has a special trick, a great power over normal functions: the `this` stays lexically bounded, which means that these functions don't have a new `this` as other functions do. Let's take an example, where you are iterating over an array with the `map` function to find the max.

In ES5:

```
var maxFinder = {
  max: 0,
  find: function(numbers) {
    // let's iterate
    numbers.forEach(
      function(element) {
        // if the element is greater, set it as the max
        if (element > this.max) {
          this.max = element;
        }
      });
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);
```

You would expect this to work, but it doesn't. If you have a good eye, you may have noticed that the `forEach` in the `find` function uses `this`, but the `this` is not bound to an object. So `this.max` is not the `max` of the `maxFinder` object... Of course you can fix it easily, using an alias:

```
var maxFinder = {
  max: 0,
  find: function(numbers) {
    var self = this;
    numbers.forEach(
      function(element) {
        if (element > self.max) {
          self.max = element;
        }
      });
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);
```

or binding the `this`:

```
var maxFinder = {
  max: 0,
  find: function(numbers) {
    numbers.forEach(
      function(element) {
        if (element > this.max) {
          this.max = element;
        }
      }.bind(this));
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);
```

or even passing it as a second parameter of the `forEach` function (as it was designed for):

```
var maxFinder = {
  max: 0,
  find: function(numbers) {
    numbers.forEach(
      function(element) {
        if (element > this.max) {
          this.max = element;
        }
      }, this);
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);
```

But there is now an even more elegant solution with the arrow function syntax:

```
let maxFinder = {
  max: 0,
  find: function(numbers) {
    numbers.forEach(element => {
      if (element > this.max) {
        this.max = element;
      }
    });
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);
```

That makes the arrow functions the perfect candidates for anonymous functions in callbacks!

2.11. Sets and Maps

This is a short one: you now have proper collections in ES6. Yay \o/! We used to have dictionaries filling the role of a map, but we can now use the class `Map`:

```
let cedric = { id: 1, name: 'Cedric' };
let users = new Map();
users.set(cedric.id, cedric); // adds a user
console.log(users.has(cedric.id)); // true
console.log(users.size); // 1
users.delete(cedric.id); // removes the user
```

We also have a class `Set`:

```
let cedric = { id: 1, name: 'Cedric' };
let users = new Set();
users.add(cedric); // adds a user
console.log(users.has(cedric)); // true
console.log(users.size); // 1
users.delete(cedric); // removes the user
```

You can iterate over a collection, with the new syntax `for ⋯ of`:

```
for (let user of users) {
  console.log(user.name);
}
```

You'll see that the `for ... of` syntax is the one the Angular team chose to iterate over a collection in a template.

2.12. Template literals

Composing strings has always been painful in JavaScript, as we usually have to use concatenation:

```
let fullname = 'Miss ' + firstname + ' ' + lastname;
```

Template literals are a new small feature, where you have to use backticks (`) instead of quotes or simple quotes, and you have a basic templating system, with multiline support:

```
let fullname = `Miss ${firstname} ${lastname}`;
```

The multiline support is especially great when your are writing HTML strings, as we will do for our Angular components:

```
let template = `<div>
  <h1>Hello</h1>
</div>`;
```

2.13. Modules

A standard way to organize functions in namespaces and to dynamically load code in JS has always been lacking. NodeJS has been one of the leaders in this, with a thriving ecosystem of modules using the CommonJS convention. On the browser side, there is also the AMD (Asynchronous Module Definition) API, used by RequireJS. But none of these were a real standard, thus leading to endless discussions on what's best.

ES6 aims to create a syntax using the best from both worlds, without caring about the actual implementation. The Ecma TC39 committee (which is responsible for evolving ES6 and authoring the specification of the language) wanted to have a nice and easy syntax (that's arguably CommonJS's strong suit), but to support asynchronous loading (like AMD), and a few goodies like the possibility to statically analyze the code by tools and support cyclic dependencies nicely. The new syntax handles how you export and import things to and from modules.

This module thing is really important in Angular 2, as pretty much everything is defined in modules,

that you have to import when you want to use them. Let's say I want to expose a function to bet on a specific pony in a race and a function to start the race.

In races_service.js:

```
export function bet(race, pony) {
  // ...
}
export function start(race) {
  // ...
}
```

As you can see, this is fairly easy: the new keyword `export` does a straightforward job and exports the two functions.

Now, let's say one of our application components needs to call these functions:

In races_service.js

```
import { bet, start } from 'races_service';
```

```
// later
bet(race, pony1);
start(race);
```

That's what is called a *named export*. Here we are importing the two functions, and we have to specify the filename containing these functions - here 'races_service'. Of course, you can import only one method if you need, you can even give it an alias:

```
import { start as startRace } from 'races_service';
```

```
// later
startRace(race);
```

And if you need to import all the methods from the module, you can use a wildcard '*'.

As you would do with other languages, use the wildcard with care, only when you really want all the functions, or most of them. As this will be analyzed by our IDEs, we will see auto-import soon and that will free us from the bother to import the right things.

With a wildcard, you have to use an alias, and I kind of like it, because it makes the rest of the code

clearer:

```
import * as racesService from 'races_service';
```

```
// later
racesService.bet(race, pony1);
racesService.start(race);
```

If your module exposes only one function or value or class, you don't have to use named export, and you can leverage the default keyword. It works great for classes for example:

```
// pony.js
export default class Pony { }
// races_service.js
import Pony from 'pony';
```

Notice the lack of curly braces to import a default. You can import it with the alias you want, but to be consistent, it's better to call the import with the module name (except if you have multiple modules with the same name of course, then you can chose an alias that allows to distinguish them). And of course, you can mix default export with named ones, but obviously with only one default per module.

In Angular 2, you're going to use a lot of these imports in your app. Each component and service will be a class, generally isolated in their own file and exported, and then imported when needed in other components.

2.14. Conclusion

That ends our gentle introduction to ES6. We skipped some other parts, but if you're comfortable with this chapter, you will have no problem writing your apps in ES6. If you want to have a deeper understanding of this, I highly recommend Exploring JS by Axel Rauschmayer or Understanding ES6 from Nicholas C. Zakas ... Both ebooks can be read online for free, but don't forget to buy it to support their authors, they have done a great work! Actually I've re-read Speaking JS, Axel's previous book, and I again learned a few things, so if you want to refresh your JS skills, I definitely recommend it!

Chapter 3. Going further than ES6

3.1. Dynamic, static and optional types

You may have heard that Angular 2 apps can be written in ES5, ES6 or TypeScript. And you may be wondering what TypeScript is, or what it brings to the table.

JavaScript is dynamically typed. That means you can do things like:

```
let pony = 'Rainbow Dash';
pony = 2;
```

And it works. That's great for all sort of things, as you can pass pretty much any object to a function and it works, as long as the object has the properties the function needs:

```
let pony = { name: 'Rainbow Dash', color: 'blue' };
let horse = { speed: 40, color: 'black' };
let printColor = animal => console.log(animal.color);
// works as long as the object has a `color` property
```

This dynamic nature allows wonderful things but it is also a pain for a few others compared to more statically-typed languages. The most obvious might be when you call an unknown function in JS from another API, you pretty much have to read the doc (or, worse, the function code) to know what the parameter should look like. Take a look at our previous example: the method `printColor` needs a parameter with a `color` property. That can be hard to guess, and of course it is much worse in day-to-day work, where we use various libraries and services developed by fellow developers. One of Ninja Squad's co-founders is often complaining about the lack of types in JS, and finds it regrettable he can't be as productive and write as good code as he would in a more statically-typed environment. And he is not entirely wrong, even if he is sometimes ranting for the sake of it too! Without type information, IDEs have no real clue if you're doing something wrong, and tools can't help you find bugs in your code. Of course, we have tests in our apps, and Angular has always been keen on making testing easy, but it's nearly impossible to have a perfect test coverage.

That leads to the maintainability topic. JS code can become hard to maintain, despite tests and documentation. Refactoring a huge JS app is no easy task, compared to what could be done in other statically-typed languages. Maintainability is a very important topic, and types help humans and tools to avoid mistakes when writing and maintaining code. Google has always been keen to push new solutions in that direction: it's easy to understand as they have some of the biggest web apps of the world, with GMail, Google apps, Maps... So they have tried several approaches to front-end maintainability: GWT, Google Closure, Dart... All trying to help writing big webapps.

For Angular 2, the Google team wanted to help us writing better JS, by adding some type information to

our code. It's not a very new concept in JS, it was even the subject of the ECMASCRIPT 4 specification, which was later abandoned. At first they announced AtScript, as a superset of ES6 with annotations (types annotations and another kind I'll discuss later). They also announced the support of TypeScript, the Microsoft language, with additional type annotations. And then, a few months later, the TypeScript team announced that they had worked closely with the Google team, and the new version of the language (1.5) would have all the shiny new things AtScript had. And the Google team announced that AtScript was officially dropped, and that TypeScript was the new top-notch way to write Angular 2 apps!

3.2. Enters TypeScript

I think this was a smart move for several reasons. For one, no one really wants to learn another language extension. And TypeScript was already there, with an active community and ecosystem. I never really used it before Angular 2, but I heard good things on it, from various people. TypeScript is a Microsoft project. But it's not the Microsoft you have in mind, from the Balmer and Gates years. It's the Microsoft of the Nadella era, the one opening to its community, and, well, open-source. Google knows this, and it's far better for them to contribute to an existing project, rather than to have to bear the burden to maintain their own. And the TypeScript framework will gain a huge popularity boost: win-win, as your manager would say.

But the main reason to bet on TypeScript is the type system it offers. It's an optional type system that helps without getting in the way. In fact, after coding some time with it, I forgot about it: you can do Angular 2 apps using TypeScript just for the parts where it really helps (more on that in a second) and simply ignore it everywhere else and write plain JS (ES6 in my case). But I do like what they have done, and we will have a look at what TypeScript offers in the next section. At the end, you'll have enough understanding to read any Angular 2 code, and you'll be able to choose whether you want to use it or not (or just a little), in your apps.

You may be wondering: why use typed code in Angular 2 apps? Let's take an example. Angular 1 and 2 have been built around a powerful concept named "dependency injection". You might already be familiar with it, as it is a common design pattern used in several frameworks for different languages and, as I said, already used in AngularJS 1.x.

3.3. A practical example with DI

To sum up what dependency injection is, think about a component of the app, let's say `RaceList`, needing to access the races list that the service `RaceService` can give. You would write `RaceList` like this:

```
class RaceList {
  constructor() {
    this.raceService = new RaceService();
    // let's say that list() returns a promise
    this.raceService.list()
    // we store the races returned into a member of `RaceList`
      .then(races => this.races = races);
      // arrow functions, FTW!
  }
}
```

But it has several flaws. One of them is the testability: it is now very hard to replace the `raceService` by a fake (mock) one, to test our component.

If we use the Dependency Injection (DI) pattern, we delegate the creation of the `RaceService` to the framework, and we simply ask for an instance. The framework is now in charge of the creation of the dependency, and, well, injects it:

```
class RaceList {
  constructor(raceService) {
    this.raceService = raceService;
    this.raceService.list()
      .then(races => this.races = races);
  }
}
```

Now, when we test this class, we can easily pass a fake service to the constructor:

```
// in a test
let fakeService = {
  list: () => {
    // returns a fake list
  }
};
let raceList = new RaceList(fakeService);
// now we are sure that the race list
// is the one we want for the test
```

But how does the framework know what to inject in the constructor? Good question! AngularJS 1.x relied on the parameter's names, but it had a severe limitation, because minification of your code would have changed the param name… You could use the array syntax to fix this, or add a metadata to the class:

```
RaceList.inject = ['RaceService'];
```

We had to add some metadata for the framework to understand what classes needed to be injected with. And that's exactly what type annotations give: a metadata giving the framework a hint it needs to do the right injection. In Angular 2, using TypeScript, we can write our `RaceList` component like:

```
class RaceList {
  raceService: RaceService;
  races: Array<string>;

  constructor(raceService: RaceService) {
    // the interesting part is `: RaceService`
    this.raceService = raceService;
    this.raceService.list()
      .then(races => this.races = races);
  }
}
```

Now the injection can be done! You don't have to use TypeScript in Angular 2, but clearly part of your code will be more elegant if you do. You can always do the same thing in plain ES6 or ES5, but you will have to manually add the metadata in another way (we'll come back on this in more details).

That's why we're going to spend some time learning TypeScript (TS). Angular 2 is clearly built to leverage ES6 and TS 1.5+, so we will have the easiest time writing our apps using it. And the Angular team really hopes to submit the type system to the standard committee, so maybe one day we'll have types in JS, and all this will be usual.

Let's dive in!

Chapter 4. Diving into TypeScript

TypeScript has been around since 2012. It's a superset of JavaScript, adding a few things to ES5. The most important one is the type system, giving TypeScript its name. From version 1.5, released in 2015, the library is trying to be a superset of ES6, including all the shiny features we saw in the previous chapter, and a few new things as well, like decorators. Writing TypeScript feels very much like writing JavaScript. By convention, TypeScript files are named with a `.ts` extension, and they will need to be compiled to standard JavaScript, usually at build time, using the TypeScript compiler. The generated code is very readable.

```
npm install -g typescript
tsc test.ts
```

But let's start with the beginning.

4.1. Types as in TypeScript

The general syntax to add type info in TypeScript is rather straightforward:

```
let variable: type;
```

The types are easy to remember:

```
let poneyNumber: number = 0;
let poneyName: string = 'Rainbow Dash';
```

In such cases, the types are optional because the TS compiler can guess them (it's called "type inference") from the values.

The type can also be coming from your app, as with the following class `Pony`:

```
let pony: Pony = new Pony();
```

TypeScript also supports what some languages call "generics", for example for an array:

```
let ponies: Array<Pony> = [new Pony()];
```

The array can only contain ponies, and the generic notation, using `<>`, indicates this. You may be wondering what the point of doing this is. Adding types information will help the compiler catch

possible mistakes:

```
ponies.push('hello'); // error TS2345
// Argument of type 'string' is not assignable to parameter of type 'Pony'.
```

So, if you need a variable to have multiple types, does it mean you're screwed? No, because TS has a special type, called `any`.

```
let changing: any = 2;
changing = true; // no problem
```

It's really useful when you don't know the type of a value, either because it's from a dynamic content or from a library you're using.

If your variable can only be of type `number` or `boolean`, you can use a union type:

```
let changing: number|boolean = 2;
changing = true; // no problem
```

4.2. Enums

TypeScript also offers `enum`. For example, a race in our app can be either `ready`, `started` or `done`.

```
enum RaceStatus {Ready, Started, Done}
let race: Race = new Race();
race.status = RaceStatus.Ready;
```

The enum is in fact a numeric value, starting at 0. You can set the value you want, though:

```
enum Medal {Gold = 1, Silver, Bronze}
```

4.3. Return types

You can also set the return type of a function:

```
function startRace(race: Race): Race {
  race.status = RaceStatus.Started;
  return race;
}
```

If the function returns nothing, you can show it using `void`:

```
function startRace(race: Race): void {
  race.status = RaceStatus.Started;
}
```

4.4. Interfaces

That's a good first step. But as I said earlier, JavaScript is great for its dynamic nature. A function will work if it receives an object with the correct property:

```
function addPointsToScore(player, points) {
  player.score += points;
}
```

This function can be applied to any object with a `score` property. How do you translate this in TypeScript? It's easy: you define an interface, like the "shape" of the object.

```
function addPointsToScore(player: { score: number; }, points: number): void {
  player.score += points;
}
```

It means that the parameter must have a property called `score` of the type `number`. You can name these interfaces, of course:

```
interface HasScore {
  score: number;
}
function addPointsToScore(player: HasScore, points: number): void {
  player.score += points;
}
```

4.5. Optional arguments

Another treat of JavaScript is that arguments are optional. You can omit them, and they will become `undefined`. But if you define a function with typed parameter in TypeScript, the compiler will shout at you if you forget them:

```
addPointsToScore(player); // error TS2346
// Supplied parameters do not match any signature of call target.
```

To show that a parameter is optional in a function (or a property in an interface), you can add `?` after the parameter. Here, the `points` parameter could be optional:

```
function addPointsToScore(player: HasScore, points?: number): void {
  points = points || 0;
  player.score += points;
}
```

4.6. Functions as property

You may also be interested in describing a parameter that must have a specific function instead of a property:

```
function startRunning(pony) {
  pony.run(10);
}
```

The interface definition will be:

```
interface CanRun {
  run(meters: number): void;
}
function startRunning(pony: CanRun): void {
  pony.run(10);
}

let pony = {
  run: (meters) => logger.log(`pony runs ${meters}m`)
};
startRunning(pony);
```

4.7. Classes

A class can implement an interface. For us, the `Pony` class should be able to run, so we can write:

```
class Pony implements CanRun {
  run(meters) {
    logger.log(`pony runs ${meters}m`);
  }
}
```

The compiler will force us to implement a `run` method in the class. If we implement it badly, by expecting a `string` instead of a `number` for example, the compiler will yell:

```
class IllegalPony implements CanRun {
  run(meters: string) {
    console.log(`pony runs ${meters}m`);
  }
}
// error TS2420: Class 'IllegalPony' incorrectly implements interface 'CanRun'.
// Types of property 'run' are incompatible.
```

You can also implement several interfaces if you want:

```
class HungryPony implements CanRun, CanEat {
  run(meters) {
    logger.log(`pony runs ${meters}m`);
  }
  eat() {
    logger.log(`pony eats`);
  }
}
```

And an interface can extend one or several others:

```
interface Animal extends CanRun, CanEat {}

class Pony implements Animal {
  // ...
}
```

When you're defining a class in TypeScript, you can have properties and methods in your class. You may realize that properties in classes are not a standard ES6 feature, it is only possible in TypeScript.

```
class SpeedyPony {
  speed: number = 10;
  run() {
    logger.log(`pony runs at ${this.speed}m/s`);
  }
}
```

Everything is public by default, but you can use the `private` keyword to hide a property or a method. If you add `private` or `public` to a constructor parameter, it is a shortcut to create and initialize a private or public member:

```
class NamedPony {
  constructor(public name: string,
              private _speed: number) {}

  run() {
    logger.log(`pony runs at ${this._speed}m/s`);
  }
}
let pony = new NamedPony('Rainbow Dash', 10);
// defines a public property name with 'Rainbow Dash'
// and a private one speed with 10
```

These shortcuts are really useful and we'll rely on them a lot in Angular 2!

NOTE	I tend to add an underscore before the name of the private variables, like `_speed` in the example before. It is not an obligation, but it is a common practice among TypeScript developers. The Angular 2 codebase does use this convention by the way.

4.8. Working with other libraries

When working with external libraries written in JS, you may think we are doomed because we don't know what types of parameter the function in that library will expect. That's one of the cool things with the TypeScript community: its members have defined interfaces for the types and functions exposed by the popular JavaScript libraries!

The files containing these interfaces have a special `.d.ts` extension. They contain a list of the library's public functions. A good place to look for these files is DefinitelyTyped. For example, if you want to use TS in your AngularJS 1.x apps, you can download the proper file from the repo:

```
tsd query angular --action install --save
```

and then include the file at the top of your code, and enjoy the compilation checks:

```
/// <reference path="angular.d.ts" />
angular.module(10, []); // the module name should be a string
// so when I compile, I get:
// Argument of type 'number' is not assignable to parameter of type 'string'.
```

`/// <reference path="angular.d.ts" />` is a special comment recognized by TS, telling the compiler to look for the interface `angular.d.ts`. Now, if you misuse an AngularJS method, the compiler will complain, and you can fix it on the spot, without having to manually run your app!

Even cooler, since TypeScript 1.6, the compiler will auto-discover the interfaces if they are packaged in your `node_modules` directory in the dependency. More and more projects are adopting this approach, and so is Angular 2. So you don't even have to worry about including the interfaces in your Angular 2 project: the TS compiler will figure it out by itself if you are using NPM to manage your dependencies!

4.9. Decorators

This is a fairly new feature, added only in TypeScript 1.5, to help supporting Angular. Indeed, as we will shortly see, Angular 2 components can be described using decorators. You may not have heard about decorators, as not every language has them. A decorator is a way to do some meta-programming. They are fairly similar to annotations which are mainly used in Java, C# and Python, and maybe other languages I don't know. Depending on the language, you add an annotation to a method, an attribute, or a class. Generally, annotations are not really used by the language itself, but mainly by frameworks and libraries.

Decorators are really powerful: they can modify their target (method, classes, etc...), and for example alter the parameters of the call, tamper with the result, call other methods when the target is called or add metadata for a framework (which is what Angular 2 decorators do). Until now, it was not something possible in JavaScript. But the language is evolving and there is now an official proposal for `decorators`, which may be standardized one day in the future (possibly in ES7/ES2016). Note that the TypeScript implementation goes slightly further than the proposed standard.

In Angular 2, we will use the decorators provided by the framework. Their role is fairly basic: they add some metadata to our classes, attributes or parameters to say things like "this class is a component", "this is an optional dependency", "this is a custom property", etc... It's not required to use them, as you can add the metadata manually (if you want to stick to ES5 for example), but the code will definitely be more elegant using decorators, as provided by TypeScript.

In TypeScript, decorators start with an `@`, and can be applied to a class, a class property, a function or a function parameter. They can't be applied to a constructor, but can be applied to its parameters.

To have a better grasp on this, let's try to build a simple decorator, `@Log()`, that will log something every time a method is called.

It will be used like this:

```
class RaceService {

  @Log()
  getRaces() {
    // call API
  }

  @Log()
  getRace(raceId) {
    // call API
  }
}
```

To define it, we have to write a method returning a function like this:

```
let Log = function () {
  return (target: any, name: string, descriptor: any) => {
    logger.log(`call to ${name}`);
    return descriptor;
  };
};
```

Depending on what you want to apply your decorator to, the function will not have exactly the same arguments. Here we have a method decorator that takes 3 parameters:

- `target`: the method targeted by our decorator
- `name`: the name of the targeted method
- `descriptor`: a descriptor of the targeted method (is the method enumerable, writable, etc...)

Here we simply log the method name, but you could do pretty much whatever you want: interfere with the parameters, the result, calling another function, etc...

So, in our simple example, every time the `getRace()` or `getRaces()` methods are called, we'll see a trace in the browser logs:

```
    raceService.getRaces();
    // logs: call to getRaces
    raceService.getRace(1);
    // logs: call to getRace
```

As a user, let's look at what a decorator in Angular 2 looks like:

```
@Component({selector: 'home'})
class Home {

  constructor(@Optional() hello: HelloService) {
    logger.log(hello);
  }

}
```

The `@Component` decorator is added to the class `Home`. When Angular 2 loads our app, it will find the class `Home` and will understand that it is a component, based on the metadata the decorator will add. Cool, huh? As you can see, a decorator can also receive parameters, here a configuration object.

I just wanted to introduce the raw concept of decorators; we'll look into every decorator available in Angular all along the book.

I have to point out that you can use decorators with Babel as a transpiler instead of TypeScript. There is even a plugin to support all the Angular 2 decorators: angular2-annotations. Babel also supports class properties, but not the type system offered by TypeScript. You can use Babel, and write "ES6+" code, but you will not be able to use the types, and they are very useful for the dependency injection for example. It's completely possible, but you'll have to add more decorators to replace the types.

So my advice would be to give TypeScript a try! All my examples from here will use it. It's not very intrusive, as you can use it just where it's useful and forget about it for the rest. If you really don't like it, it will not be very difficult to switch to ES6 with Babel or Traceur, or even ES5, if you are slightly crazy (but honestly, an Angular 2 app in ES5 has pretty ugly code).

Chapter 5. The wonderful land of Web Components

Before going further, I'd like to make a brief stop to talk about Web Components. You don't have to know about Web Components to write Angular 2 code. But I think it's a good thing to have an overview of what they are, because some choices in Angular 2 have been made to facilitate the integration with Web Components, or to make the components we will build similar to Web Components. Feel free to skip this part if you have no interest in this topic; however, I do believe you'll learn a thing or two that will be useful for the rest of the road.

5.1. A brave new world

Components are an old fantasy in development. Something you can grab off the shelves and drop into your app, something that would work right away and bring a needed functionality to your users.

My friends, this time has come.

Well, maybe. At least, there is the start of something.

That's not completely new. We have had components in web development for quite some time, but they usually require some kind of dependency, like jQuery, Dojo, Prototype, AngularJS, etc… Not necessarily libraries you wanted to add to your app.

Web Components attempt to solve this problem: let's have reusable and encapsulated components.

They rely on a set of emerging standards that browsers don't perfectly support yet. But, still, it's an interesting topic, even if there's a chance that we'll have to wait a few years to use them fully, or even that the concept never takes off.

This emerging standard is defined in 4 specifications:

- Custom elements
- Shadow DOM
- Template
- HTML imports

Note that the samples are most likely to work in a recent Chrome or Firefox browser.

5.2. Custom elements

Custom elements are a new standard allowing developers to create their own DOM elements, making something like `<pony-cmp><pony-cmp>` a perfectly valid HTML element. The specification defines how to declare such elements, how to make them extend existing elements, how to define your API, etc…

Declaring a custom element is done with a simple `document.registerElement('pony-cmp')`:

```
// new element
var PonyCmp = document.registerElement('pony-cmp');
// insert in current body
document.body.appendChild(new PonyCmp());
```

Note that the name must contain a dash, so that the browser knows it is a custom element. Of course, your custom element can have properties and methods, and it also has lifecycle callbacks, to be able to execute code when the component is inserted or removed, or when one of its attributes changes. It can also have a template of its own. Maybe the `pony-cmp` displays an image of the pony or just its name:

```
// let's extend HTMLElement
var PonyCmpProto = Object.create(HTMLElement.prototype);
// and add some template using a lifecycle
PonyCmpProto.createdCallback = function() {
  this.innerHTML = '<h1>General Soda</h1>';
};

// new element
var PonyCmp = document.registerElement('pony-cmp', {prototype: PonyCmpProto});
// insert in current body
document.body.appendChild(new PonyCmp());
```

If you try to look at the DOM, you'll see `<pony-cmp><h1>General Soda</h1></pony-cmp>`. But that means the CSS and JavaScript logic of your app can have undesired effects on your component. So, usually, the template is hidden and encapsulated in something called Shadow DOM, and you'll only see `<pony-cmp><pony-cmp>` if you inspect the DOM, despite the fact that the browser displays the pony's name.

5.3. Shadow DOM

With a mysterious name like this, you expect something with great powers. And surely it is. The Shadow DOM is a way to encapsulate the DOM of our component. This encapsulation means that the stylesheet and JavaScript logic of your app will not apply on the component and ruin it inadvertently. It gives us the perfect tool to hide the internals of a component, and be sure that nothing leaks from the component to the app, or vice-versa.

Going back to our previous example:

```
var PonyCmpProto = Object.create(HTMLElement.prototype);

// add some template in the Shadow DOM
PonyCmpProto.createdCallback = function() {
  var shadow = this.createShadowRoot();
  shadow.innerHTML = '<h1>General Soda</h1>';
};

var PonyCmp = document.registerElement('pony-cmp', {prototype: PonyCmpProto});
document.body.appendChild(new PonyCmp());
```

If you try to inspect it now you should see:

```
<pony-cmp>
  #shadow-root (open)
    <h1>General Soda</h1>
</pony-cmp>
```

Now, even if you try to add some style to the `h1` elements, the visual aspect of the component won't change at all: that's because the Shadow DOM acts like a barrier.

Until now, we just used a string as a template of our web component. But that's usually not the way you do that. Instead, the best practice is to use the <template> element.

5.4. Template

A template specified in a `<template>` element is not displayed in your browser. Its main goal is to be cloned in an element at some point. What you declare inside will be inert: scripts don't run, images don't load, etc... Its content can't be queried by the rest of the page using usual method like `getElementById()` and it can be safely placed anywhere in your page.

To use a template, it needs to be cloned:

```
<template id="pony-tpl">
  <style>
    h1 { color: orange; }
  </style>
  <h1>General Soda</h1>
</template>

var PonyCmpProto = Object.create(HTMLElement.prototype);

// add some template using the template tag
PonyCmpProto.createdCallback = function() {
  var template = document.querySelector('#pony-tpl');
  var clone = document.importNode(template.content, true);
  this.createShadowRoot().appendChild(clone);
};

var PonyCmp = document.registerElement('pony-cmp', {prototype: PonyCmpProto});
document.body.appendChild(new PonyCmp());
```

Maybe we could declare this in a single file, and we would have a perfectly encapsulated component… Let's do this with HTML imports!

5.5. HTML imports

This is the last specification. HTML imports allow to import HTML into HTML. Something like `<link rel="import" href="pony-cmp.html">`. This file, `pony-cmp.html`, would contain everything needed: the template, the scripts, the styles, etc…

If someone wants to use our wonderful component, they just have to use an HTML import and they are good to go!

5.6. Polymer and X-tag

All these things put together make the Web Components. I'm far from being an expert on this topic, and there are all sorts of twisted traps on this road.

As Web Components are not fully supported by every browser, there is a polyfill you can include in your app to make sure it will work. The polyfill is called web-component.js, and it's worth noting that it is a joint effort from Google, Mozilla and Microsoft among others.

On top of this polyfill, a few libraries have seen the light. All aim to facilitate working with Web Components, and often come with some ready-to-use Web Components.

Among the most notable initiatives, you find:

- Polymer from Google
- X-tag from Mozilla and Microsoft

I won't go into the details, but you can easily use an already existing Polymer Component. Let's say you want a Google Map in your app:

```
<!-- Polyfill Web Components support for older browsers -->
<script src="webcomponents.js"></script>

<!-- Import element -->
<link rel="import" href="google-map.html">

<!-- Use element -->
<body>
  <google-map latitude="45.780" longitude="4.842"></google-map>
</body>
```

There are a LOT of components out there. You can have an overview on https://customelements.io/.

Polymer also helps you build your own components:

```
<dom-module id="pony-cmp">
  <template>
    <h1>[[name]]</h1>
  </template>
  <script>
    Polymer({
      is: 'pony-cmp',
      properties: {
        name: String
      }
    });
  </script>
</dom-module>
```

and use them:

```
<!-- Polyfill Web Components support for older browsers -->
<script src="webcomponents.js"></script>

<!-- Polymer -->
<link rel="import" href="polymer.html">

<!-- Import element -->
<link rel="import" href="pony-cmp.html">

<!-- Use element -->
<body>
  <pony-cmp name="General Soda"></pony-cmp>
</body>
```

You can do a lot of cool things with Polymer, like two-way data binding, default values for attributes, emit custom events, react on attribute changes, repeat elements if we give a collection to a component, etc...

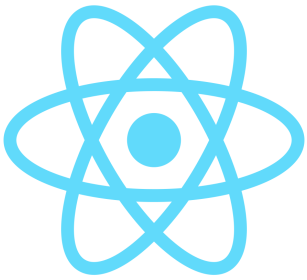
That's obviously far too short a chapter to tell you everything there is to say on Web Components, but you'll see that some of the concepts are going to pop out along your read. And you'll definitely see that the Google team designed Angular 2 to make it easy to use Web Components along our Angular 2 components.

Chapter 6. Grasping Angular's philosophy

To write an Angular 2 application, you have to grasp a few things on the framework's philosophy.

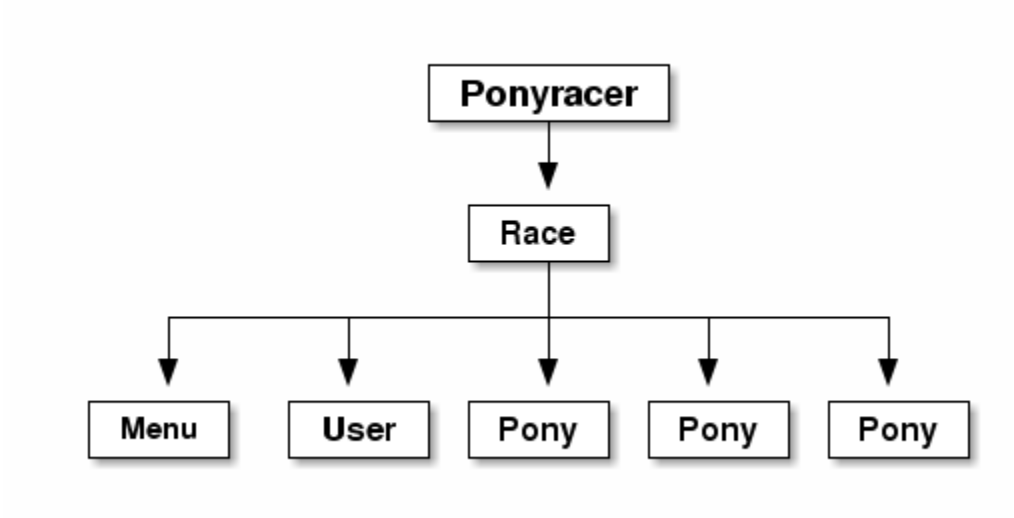
First and foremost, Angular 2 is component-oriented. You will write tiny components and, together, they will constitute a whole application. A component is a group of HTML elements in a template, dedicated to a particular task. For this, you will usually also need to have some logic linked to that template, to populate data, and react to events for example. For the veterans of AngularJS 1.x, it's a bit like a 'template/controller' duo, or a directive.

It has to be said that a standard has been established around this component thing: the Web Component standard. Even if it's not completely supported by browsers yet, you can build small and isolated components, reusable in different applications - an old dream of computer programming. This component orientation is something that is becoming widely shared across front-end frameworks: ReactJS, the latest cool kid from Facebook, has been doing it that way from the beginning; EmberJS and AngularJS have their way of doing something similar; and newcomers like Aurelia or Vue.js are betting on building small components too.



Angular 2 is not alone in this, but they are among the first (they might actually be the first?) to really care about the integration of Web Components (the standard ones). But let's forget about this for now, as it is a more advanced topic.

Your components will be arranged in a hierarchical way, like the DOM is. A root component will have child components, each of them will also have children, etc… If you want to display a pony race (who wouldn't?), you'll have something like an app (Ponyracer), with a child view (Race), displaying a menu (Menu), the logged in user (User), and, of course, the ponies (Pony) in the races:



Writing components will be your everyday work, so let's see what it looks like. The Angular team wanted to harness another goodness of today's web development: ES6 (or ES2015, whatever you like to call it). So you can write your components in ES5 (but that's not very cool) or in ES6 (way cooler!). But that was not enough for them, they wanted to use a feature that is not a standard (yet): decorators. So they worked closely with the transpiler teams (Traceur and Babel) and the TypeScript team at Microsoft, to enable us to use decorators in our Angular 2 apps. A few decorators are available, allowing to easily declare a component for example. I hope you already know all of that, as I just spent two chapters on these things!

For example, if we simplify, the Race component could look like this:

```
import {Component} from 'angular2/core';
import {Pony} from './components';
import {RacesService} from './services';

@Component({
  selector: 'race',
  templateUrl: 'race/race.html',
  directives: [Pony]
})
export class RaceCmp {

  race: any;

  constructor(racesService: RacesService) {
    racesService.get()
      .then(race => this.race = race);
  }
}
```

And the template looks like this:

```
<div>
  <h2>{{ race.name }}</h2>
  <div>{{ race.status }}</div>
  <div *ngFor="#pony of race.ponies">
    <pony [pony]="pony"></pony>
  </div>
</div>
```

If you already know AngularJS 1.x, the template should look familiar, with the same expression in curly braces `{{ }}`, that will be evaluated and replaced by the according value. Some things have changed though: no more `ng-repeat` for example. I don't want to go too deep for now, merely just give you a feel of what the code looks like.

A component is a very isolated piece of your app. Your app *is* a component like the others. In a perfect world, you will take available components from the community and just put them in your app, by adding them in the hierarchy somewhere.

In the next chapters, we are going to explore how to get started, how to build a small component, and the templating syntax.

There is another concept that is at the core, and that is Dependency injection (often called by its little name, DI). This a very powerful pattern, and you will quickly get used to it after reading the dedicated chapter. It is especially useful to test your application, and I love doing tests, watching the progress bar go all green in my IDE. It makes me feel I'm doing a good job. So there will be an entire chapter on

testing everything: your components, your services, your UI…

Angular still has the magic feeling it had in v1, where changes were automatically detected by the framework and applied to the model and the views. But it is done in a very different way than it was then: the change detection now uses a concept called `zones`. We will look into this, of course.

Angular is also a complete framework which provides a lot of help for performing common tasks in web development. Writing forms, calling an HTTP backend, routing, interacting with other libraries, animations, you name it: you're covered.

Well, that's a lot of things to learn! We should start with the beginning: bootstrap an app and write our first component.

Chapter 7. From zero to something

7.1. Developing and building a TypeScript app

Let's start by creating our first Angular 2 app and our first component, with a minimum of tooling. You'll have to install Node.js and NPM on your system. The best way to do that depends on your operating system - you can find more information on the official website. Make sure you have a recent enough version of Node.js (by executing `node --version`), something like 4.2+. We'll write our app in TypeScript, so you'll have to install it via `npm`:

```
npm install -g typescript
```

Then, create a new, empty folder for our experiment, and use `tsc` from that new empty folder to initialize a project. `tsc` stands for **TypeScript Compiler**. It's provided by the `typescript` NPM module we just installed globally:

```
tsc --init --target es5 --sourceMap --experimentalDecorators --emitDecoratorMetadata
```

This will create a file, `tsconfig.json`, which stores the TypeScript compilation options. As we saw in the previous chapters, we are using TypeScript with decorators (hence the last two flags), and we want our code to transpile to ECMASCRIPT 5, allowing it to run in every browser. The `sourceMap` option allows generating source maps, i.e. files that contain a mapping between the generated ES5 code and the original TypeScript code. Those source maps are used by the browser to let you debug the ES5 code it executes by stepping through the original TypeScript code that you have written.

We now want to start using our preferred IDE. You can use pretty much anything you want, but you should activate the TypeScript support for maximum comfort (and make sure you are using TypeScript 1.5+). Pick your favorite IDE: Webstorm, Atom, VisualStudio Code… All of them have great support for TypeScript.

The TypeScript compiler (and usually the IDE) relies on the `tsconfig.json` file to know what options it should use. The file should look like the following:

```
{
  "compilerOptions": {
    "target": "es5",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "sourceMap": true,
    "module": "commonjs",
    "noImplicitAny": false,
    "outDir": "built",
    "rootDir": "."
  },
  "exclude": [
    "node_modules"
  ]
}
```

You can see that a few options have been added by default. An interesting one is the `module` option, telling us that our code will be packaged in CommonJS modules. It will be important in a moment.

Now we are ready to launch the TypeScript compiler, using the watch mode to compile the files when we save. Sometimes your IDE will do that for you.

```
tsc --watch
```

You should see something like:

```
Compilation complete. Watching for file changes.
```

You can let this run in the background and open a new terminal for what comes next.

We now need to add the Angular 2 library and our code. For the Angular 2 library, we are going to download it using NPM, a great tool to manage dependencies.

To avoid a few problems, we'll use NPM version 2. Check which version you have with:

```
npm -v
```

If you have NPM version 3, you can easily install NPM version 2:

```
npm install -g npm@2
```

Now that's done, let's start by creating the `package.json` file, containing all the information that NPM

needs. You can answer `Enter` to every question.

```
npm init
```

Then, let's install `angular2` and its dependencies:

```
npm install --save angular2
```

You can have a look at your `package.json` file, it should now contain the following dependencies:

- `reflect-metadata`, as we are using decorators.
- `es6-shim` and `es6-promise` are included, as we want to be sure that our browser will have everything needed.
- `rxjs`, a really cool library called RxJS for reactive programming. We have a dedicated chapter on this topic.
- and finally, the `zone.js` module, doing the heavy lifting of running our code in isolated zones for detecting the changes (we'll dive into this later also).

The tooling is now in place, let's create our first component!

7.2. Our first component

Create a new file, called `ponyracer_app.ts`.

When you save your file, you should see a new file `ponyracer_app.js` popping in the `built` directory: it's the TypeScript compiler doing its job. If not, you probably killed your TypeScript compiler watching for changes, so you might run it again with `tsc --watch`, and leave it running in the background.

As we saw in the previous section, a component is a combination of a view (the template) and some logic (our TS class). Let's create a class:

```
export class PonyRacerApp {

}
```

Our application itself is a simple component. To tell Angular that it is a component, we use the `@Component` decorator. To be able to use it, we have to import it:

```
import {Component} from 'angular2/core';

@Component()
export class PonyRacerApp {

}
```

If your IDE supports it, code completion should work as the Angular 2 dependency has its own `d.ts` files in the `node_modules` directory, and TypeScript is able to detect it. You can even navigate to the type definitions if you want to.

TypeScript will bring its type-checking to the table, so you'll see what mistakes you make as you type. But the errors are not necessarily blocking: if you forget to add the type information to your variable, the code will still compile to JavaScript and run properly.

I try to keep the TypeScript errors count to 0, but you can do as you want. As we are using source maps, you can see the TS code directly from your browser, and even debug your app by setting breakpoints in the TypeScript code.

The `@Component` decorator is expecting a configuration object. We'll see later in details what you can configure here, but for now only one property is expected: the `selector` one. It will tell Angular what to look for in our HTML pages. Every time the selector we have defined is found in our HTML, Angular is going to replace the element selected by our component:

```
import {Component} from 'angular2/core';

@Component({
  selector: 'ponyracer-app'
})
export class PonyRacerApp {

}
```

So, here, every time our HTML will contain an element like `<ponyracer-app></ponyracer-app>`, Angular will instantiate a new instance of our `PonyRacerApp` class.

NOTE

There is not a clear naming convention established yet. I tend to suffix my component classes with `Cmp`. A component's selector should have a dash, like `pony-cmp`, even if that's not mandatory. But, if you want to let other developers use your component and avoid potential name clashes, you should adopt a convention like "namespace-component". The namespace should be a short one, like "ns" for Ninja Squad for example. That would give a reusable component with a selector `ns-pony`. Finally, you can add a suffix to your filenames to see their role at first glance, `pony.component.ts` or `race.service.ts` for example.

A component must also have a template. We could externalize the template in another file, but for our first time, let's keep it simple, and inline it:

```
import {Component} from 'angular2/core';

@Component({
  selector: 'ponyracer-app',
  template: '<h1>PonyRacer</h1>'
})
export class PonyRacerApp {

}
```

Don't forget to import the `Component` decorator if you want to use it. You may forget to do so at the beginning, but it won't last, as the compiler will yell at you! ;)

You'll see that most of the things we need are in the `angular2/core` module, but that's not always the case. For example, when dealing with HTTP, we'll use imports from `angular2/http`; or, if we use the router, we'll import from `angular2/router`, etc...

7.3. Bootstrapping the app

Finally, we need to start our app. For this, there is a `bootstrap` method available. You have to import it too, from `angular2/platform/browser`. Now, that's a strange module! Why is it not `angular2/core`? Good question: it's because you might want to run your app somewhere else than in a browser, as Angular 2 supports server-side rendering or running in a Web Worker for example. And in these cases, the bootstrap logic would be a bit different. But we'll see this later, as we are just focusing on the browser right now.

Let's create another file, for example `bootstrap.ts`, to separate the bootstrap logic:

```
import {bootstrap} from 'angular2/platform/browser';
import {PonyRacerApp} from './ponyracer_app';

bootstrap(PonyRacerApp)
 .catch(err => console.log(err)); // useful to catch the errors
```

Yay! But wait a second. We don't have any HTML file, do we? You're right about that!

Create another file named `index.html` and add the following content:

```
<html>

<head></head>

<body>
  <ponyracer-app>
    You will see me while Angular starts the app!
  </ponyracer-app>
</body>

</html>
```

We now need to add scripts to our HTML files. In AngularJS 1.x, it was simple: you just needed to add a script for angular.js, and a script for every JS file you wrote, and you were ready to go. There was a downside though: everything had to be loaded statically, at startup, which could lead to long startup times for big applications.

With Angular 2, things are more complicated, but complexity comes with additional power. Angular is now bundled in modules, and these modules can be loaded dynamically. Our app is also bundled in modules, as we saw earlier.

There are a few problems though:

- modules don't exist in ES5, and browsers only support ES5 at the moment;
- the ES6 designers have decided to specify how modules are defined, imported, etc. But they have not yet specified how they're supposed to be packaged and loaded by the browsers.

To load our modules, we will thus need to rely on a tool: SystemJS. SystemJS is a small module loader: you add it (statically) into your HTML page, you tell it where modules are located on your server, and you load one of them. It automatically figures out the dependencies between modules, and downloads the ones used by your application.

This will lead to a bazillion of JS file downloads. That is fine during development, but it is a problem for production. Fortunately, SystemJS also comes with a tool that can pack several small modules into bigger bundles. When a module is needed, the bundle containing that module (and several other ones) will then be downloaded.

Let's install SystemJS:

```
npm install --save systemjs
```

We need to load `SystemJS` statically, and to tell it where is our bootstrap module is (in `built/bootstrap`). We also need to show it where to find the dependencies of our application, like `angular2`. But first, we need to include `angular2-polyfills`, a special script that needs to load first, to include `reflect-metadata`

and `zone.js`:

```
<html>

<head>
  <script src="node_modules/angular2/bundles/angular2-polyfills.js"></script>
  <script src="node_modules/systemjs/dist/system.js"></script>
  <script>
  System.config({
    // we want to import modules without writing .js at the end
    defaultJSExtensions: true,
    // the app will need the following dependencies
    map: {
      'angular2': 'node_modules/angular2',
      'rxjs': 'node_modules/rxjs'
    }
  });
  // and to finish, let's boot the app!
  System.import('built/bootstrap');
  </script>
</head>

<body>
  <ponyracer-app>
    You will see me while Angular starts the app!
  </ponyracer-app>
</body>

</html>
```

OK! Let's start an HTTP server to serve this mini app. I'm going to use `http-server`, a node tool that does pretty much what its name says. But you may of course use whatever web server you prefer: Apache, Nginx, Tomcat, etc. To install it, use `npm`:

```
npm install -g http-server
```

To start it, go to your directory, and enter:

```
http-server
```

Now it's time for the show! Open your browser to http://localhost:8080.

You should now briefly see "You will see this while Angular start the app!", and then "PonyRacer" should appear! Your first component is a success!

It's not really a dynamic app, and we could have done the same in one second in a static HTML page, I'll give you that. So let's jump to the next sections, and learn all about dependency injection and templating.

7.4. From zero to something better with angular-cli

In a real project, you'll probably have to set up several other things like:

- some tests to check if we're not breaking things
- a build tool, to orchestrate the various tasks (compile, test, package, etc...)

And it's a bit cumbersome to setup everything yourself, even if I think it's necessary to do it once to understand what's going on.

These past few years, a lot of small project generators have seen the light, pretty much all using the great Yeoman. It used to be the case for AngularJS 1.x, and there are already a few attempts for Angular 2.

But this time, the Google team has been working on this issue, and they have come up with something: `angular-cli`.

`angular-cli` is a command line utility to easily quick start a project, already configured with a build tool (Broccoli), tests, packaging, etc...

The idea is not new, and is in fact borrowed from another popular framework: EmberJS and its popularly acclaimed `ember-cli`.

The tool is still under development, but I think it will be the *de facto* standard to create Angular 2 apps in the future, so you can give it a try:

```
npm i -g angular-cli
ng new ponyracer
```

This will create a project skeleton. You can start your app with:

```
ng serve
```

This will start a small HTTP server locally, with a hot reload configuration. That means every time you are going to modify and save a file, the app will refresh in your browser.

A few other possibilities are available, like creating a component skeleton:

```
ng generate component pony
```

This will create a component file, with its associated template, stylesheet and test file.

The tool is not only here to help us develop the application, it also comes with a plugin system that will simplify a few tasks like deployment. For example, you can quickly deploy on Github Pages, using the `github-pages` plugin:

```
ng github-pages:deploy
```

In the long term, this is going to be great! We'll have the same code organization across projects, a common way to build and deploy apps, and probably a huge eco-system of plugins for simplifying some tasks.

So feel free to have a look at `angular-cli`!

Chapter 8. The templating syntax

We've seen that a component needs to have a view. To define a view, you can define a template inline or in a separate file. You're probably familiar with a templating syntax, maybe even the one from AngularJS 1.x. To simplify things, a template helps us to render HTML with some dynamic parts depending on our data.

Angular 2 has its own templating syntax that we need to learn before going further.

Let's take a simple example. Our first component looked like:

```
import {Component} from 'angular2/core';

@Component({
  selector: 'ponyracer-app',
  template: '<h1>PonyRacer</h1>'
})
export class PonyRacerApp {

}
```

Now we want to display some dynamic data on this first page, maybe the number of users registered into our app. Later we'll see how to get data from a server, but for now we'll say that this number of users is directly hard-coded in our class:

```
@Component({
  selector: 'ponyracer-app',
  template: '<h1>PonyRacer</h1>'
})
export class PonyRacerApp {

  numberOfUsers: number = 146;

}
// end:users-cmp[]
```

Now, how do we change our template to display this variable? The answer is interpolation.

8.1. Interpolation

Interpolation is a big word for a simple concept.

Quick example:

```
@Component({
  selector: 'ponyracer-app',
  template: `
    <h1>PonyRacer</h1>
    <h2>{{numberOfUsers}} users</h2>
  `
})
export class PonyRacerApp {

  numberOfUsers: number = 146;

}
```

We have a `PonyRacerApp` component that will be activated every time Angular finds a `<ponyracer-app>` tag. The `PonyRacerApp` class has a property, `numberOfUsers`. And the template has been augmented with an `<h2>` tag, using the famous double curly braces (a.k.a. "mustaches") to indicate that an expression has to be evaluated. This kind of templating is called interpolation.

We should now see in the browser:

```
<ponyracer-app>
  <h1>PonyRacer</h1>
  <h2>146 users</h2>
</ponyracer-app>
```

as `{{numberOfUsers}}` will be replaced by its value. When Angular detects a `<ponyracer-app>` element in the page, it creates an instance of the `PonyRacerApp` class, and this instance is the evaluation context of the template's expressions. Here the `PonyRacerApp` instance sets the `numberOfUsers` property to '146', so we have '146' displayed on screen.

The magic is that, whenever the value of `numberOfUsers` changes in our object, the template will be automatically updated! That's called 'change detection', and it's one of the great features of Angular.

One important fact to remember, though: if we try to display a variable that does not exist, then, instead of displaying `undefined`, Angular is going to display an empty string. The same will happen for a `null` variable.

Let's say that, instead of a simple value, our first component has a more complex `user` object, reflecting the current user.

```
@Component({
  selector: 'ponyracer-app',
  template: `
    <h1>PonyRacer</h1>
    <h2>Welcome {{user.name}}</h2>
  `
})
export class PonyRacerApp {

  user: any = { name: 'Cédric' };

}
```

As you can see, we can interpolate more complex expressions, like accessing the property of an object.

```
<ponyracer-app>
  <h1>PonyRacer</h1>
  <h2>Welcome Cédric</h2>
</ponyracer-app>
```

What happens if we have a typo in our template, with a property that does not exist in the class?

```
@Component({
  selector: 'ponyracer-app',
  // typo: users is not user!
  template: `
    <h1>PonyRacer</h1>
    <h2>Welcome {{users.name}}</h2>
  `
})
export class PonyRacerApp {

  user: any = { name: 'Cédric' };

}
```

When loading the app, you will have an error, telling you that this property does not exist:

```
Cannot read property 'name' of undefined in [{{users.name}} in PonyRacerApp]
```

That's great, because now you are quite sure that your templates are correct. One of the most often encountered problem in AngularJS 1.x was that this type of error could not be detected, and you could

lose quite some time trying to figure out what was going on (usually a typo, like `{{users.name}}` instead of `{{user.name}}`). We have given quite a few training sessions, and I can assure you that 30% of beginners were having this problem on the first day. I got a bit tired of it, and I even submitted a pull-request to display a warning when the parser would find an unknown variable, which was refused for valid reasons and with a comment from the core team saying they had an idea on how to solve this in Angular 2. And they did!

One last little but handy feature. What happens if my `user` object is in fact fetched from the server, and thus initialized to `undefined` before being valued with the result of the server call? Is there a way to avoid the errors when the template is compiled?

Yes, there is: instead of writing `user.name`, you write `user?.name`:

```
@Component({
  selector: 'ponyracer-app',
  // user is undefined
  // but the ?. will avoid the error
  template: `
    <h1>PonyRacer</h1>
    <h2>Welcome {{user?.name}}</h2>
  `
})
export class PonyRacerApp {

  user: any;

}
```

And you don't have errors anymore! The `?.` is sometimes called the "Safe Navigation Operator".

So we can write our templates more safely, and be assured that they will behave properly.

Let's go back to our example. We are now displaying a greeting message. Maybe we can go a step further and display the upcoming pony races.

That should lead us to write our second component. For now, we'll just make it simple:

```
// in another file, races_cmp.ts
import {Component} from 'angular2/core';

@Component({
  selector: 'races-cmp',
  template: `<h2>Races</h2>`
})
class RacesCmp {

}
```

Nothing fancy: a simple class, decorated with `@Component` to give it a `selector` to match and an inline template.

Now we want to include this component in our `PonyRacerApp` template. What do we need to do?

8.2. Using other components in our templates

We have our app component, `PonyRacerApp`, where we want to display the pony races component, `RacesCmp`.

```
// in ponyracer_app.ts
import {Component} from 'angular2/core';

@Component({
  selector: 'ponyracer-app',
  // added the RacesCmp component
  template: `
    <h1>PonyRacer</h1>
    <races-cmp></races-cmp>
  `
})
export class PonyRacerApp {

}
```

As you can see, we added the `RacesCmp` component in the template, by including a tag whose name matches the selector we defined for the component.

Buuuuuut, that will not work: your browser will not display the races component.

Let's get this out of the way, as it is probably what I like the least about Angular 2. **If you want to use another component in your template, you will have to declare it in your `@Component` decorator too.**

Here, we have to declare the `RacesCmp` component we are using in your template, using the `directives`

attributes in the `@Component` decorator, as follows:

```
// in ponyracer_app.ts
import {Component} from 'angular2/core';

// do not forget to import the component
import {RacesCmp} from './races_cmp';

@Component({
  selector: 'ponyracer-app',
  template: `
    <h1>PonyRacer</h1>
    <races-cmp></races-cmp>
  `,
  // declare all the components you use in your template
  directives: [RacesCmp]
})
export class PonyRacerApp {

}
```

The `@Component` decorator has a `directives` attribute that can receive an array. This array must contain all the components we are using in the view template. Note that you will pass the class directly, so you'll have to import it.

In order to import `RacesCmp` in the class `PonyRacerApp`, you need to export the class `RacesCmp` in its source file `races_cmp.ts` (read the section about ES6 modules again if that's not clear for you). So `RacesCmp` will look like:

```
// in another file, races_cmp.ts
import {Component} from 'angular2/core';

@Component({
  selector: 'races-cmp',
  template: `<h2>Races</h2>`
})
export class RacesCmp {

}
```

Now, our races component will proudly be displayed in our browser:

```
<ponyracer-app>
  <h1>PonyRacer</h1>
  <races-cmp>
    <h2>Races</h2>
  </races-cmp>
</ponyracer-app>
```

Having to declare every component we use is tedious, and you will forget from time to time, losing minutes to figure out why the hell this component is not displayed! Believe me, it happens all too often…

So remember this: **if you add a component in your template, you must also add it in the `directives` attributes of your `@Component` decorator.**

In AngularJS 1.x, you did not have to do that, because there was a global registry of all the components and directives. In Angular 2, this registry is per component, to have a better encapsulation, but it comes with a cost. We will also use directives, hence the name of the decorator attribute. We'll come back to directives in a few minutes.

Also, we'll see that we don't need to import components and directives of the framework: they are already available. This mandatory declaration only applies to our components.

Now that this is out the way, let's talk about the good stuff.

8.3. Property binding

Interpolation is only one of the ways to have dynamic parts in your template.

In fact, the interpolation we just saw is just an easy way to use what is the core of Angular 2 templating system: property binding.

In Angular 2, every DOM property can be written to via special attributes on HTML elements surrounded with square brackets `[]`. It looks weird at first, but in fact it is valid HTML (it surprised me too). An HTML attribute can start with pretty much anything you want except a few characters like quotes, apostrophes, slashes, equals, spaces…

I'm talking about DOM properties, but maybe this is not clear for you. We usually write to HTML attributes, right? Right, usually we do. Let's take this simple HTML:

```
<input type="text" value="hello">
```

The `input` tag above has two *attributes*: a `type` attribute and a `value` attribute. When the browser parses this tag, it creates a corresponding DOM node (an HTMLInputElement if we want to be accurate), which has the matching *properties* `type` and `value`. Each standard HTML attribute has a corresponding

property in the DOM node. But the DOM node also has additional properties, which don't have a corresponding attribute. For example: `childElementCount`, `innerHTML` or `textContent`.

The interpolation we had above to display the user's name:

```
<p>{{user.name}}</p>
```

is just sugar syntax for the following:

```
<p [textContent]="user.name"></p>
```

The square bracket syntax allows to modify the DOM property `textContent`, and we give it the value `user.name` which will be evaluated in the context of the current component instance, as it was for the interpolation.

Note that the parser is case-sensitive, so you have to write the property name with the correct case: `textcontent` or `TEXTCONTENT` will not work, it has to be `textContent`.

DOM properties have a great advantage over HTML attributes: they have up-to-date values. In my input example, the `value` *attribute* will always contain 'hello', whereas the `value` *property* of the DOM node is dynamically modified by the browser, and thus contains whatever the user has entered in the text field.

Finally, properties can have boolean values, whereas some attributes can only reflect it by being present or absent on the start tag. For example, you have the `selected` attribute on the `<option>` tag: no matter what value you give it, it will select the option, as long as it is present.

```
<option selected>Rainbow Dash</option>
<option selected="false">Rainbow Dash</option> <!-- still selected -->
```

With properties access like Angular 2 gives us, you can write:

```
<option [selected]="isPonySelected" value="Rainbow Dash">Rainbow Dash</option>
```

And the pony will be selected if `isPonySelected` is `true`, and not be selected if it is `false`. And whenever the value of `isPoneySelected` changes, the selected property will be updated.

You can do a lot of cool things with this, things that were cumbersome in AngularJS 1.x. For example, having a dynamic source URL for an image.

```
<img src="{{pony.avatar.url}}">
```

This syntax has a major problem: the browser will try to fetch the image as soon as it reads the `src` attribute. You can see that it will fail: it will make an HTTP request to `{{pony.avatar.url}}` which is not a valid URL...

In AngularJS 1.x, there was a special directive to take care of that: `ng-src`.

```
<img ng-src="{{pony.avatar.url}}">
```

Having `ng-src` instead of `src` did solve the problem, as it tricked the browser into ignoring it. Once AngularJS had compiled the app, it added the `src` attribute with a correct URL, hence triggering the image download. Cool! But it had two downsides:

- first, you had to know, as a developer, what value to give to `ng-src`. Was it 'https://gravatar.com'? '"https://gravatar.com"'? 'pony.avatar.url'? '{{pony.avatar.url}}'? No way to know, except by reading the documentation.
- second, the Angular team had to create a directive for each standard attribute. They did, and we had to learn them. But we are now in a world where your HTML can also contain external Web Component, looking like:

```
<pony-cmp name="Rainbow Dash"></pony-cmp>
```

If this a Web Component that you want to use, you have no easy way to pass a dynamic value with most JS frameworks, except if the developer of the Web Component had taken extra care to make it possible. Read the chapter on Web Components for more information.

A Web component should act like a browser element. They have a DOM API based on properties, events and methods. With Angular 2, you can do:

```
<pony-cmp [name]="pony.name"></pony-cmp>
```

And it works!

Angular will maintain the properties and attributes in sync.

No more directives to learn! If you wish to hide an element, you can use the standard `hidden` property:

```
<div [hidden]="isHidden">Hidden or not</div>
```

And the `div` will be hidden only when `isHidden` is `true`, as Angular will work directly with the `hidden` property. No more `ng-hide`, and this is just one of the dozens of directives that were used in Angular 1.

You can also access nested properties like the `color` attribute of the `style` property.

```
<p [style.color]="foreground">Friendship is Magic</p>
```

If the `foreground` attribute is changing to 'green', then the text will update its color to 'green' too!

So Angular 2 is using properties. Which values can we pass? We already saw the interpolation `property="{{expression}}"`:

```
<pony-cmp name="{{pony.name}}"></pony-cmp>
```

is the same thing than `[property]="expression"` (that you will usually prefer):

```
<pony-cmp [name]="pony.name"></pony-cmp>
```

If you want to write 'Pony' followed by the pony's name, you have two options:

```
<pony-cmp name="Pony {{pony.name}}"></pony-cmp>
<pony-cmp [name]="'Pony ' + pony.name"></pony-cmp>
```

If your value is not a dynamic one, you can simply write `property="value"`:

```
<pony-cmp name="Rainbow Dash"></pony-cmp>
```

All of these are equivalent, and the syntax doesn't depend on how the developer chose to design its component, as it was the case in AngularJS 1.x where you had to know if the component was expecting a value or a reference for example.

Of course, the expression can also contain function calls:

```
<pony-cmp name="{{pony.fullName()}}"></pony-cmp>
<pony-cmp [name]="pony.fullName()"></pony-cmp>
```

8.4. Events

If you're developing a webapp, you know that displaying things is just one part of the job: you also have to deal with user interactions. To allow this, the browser fires events, which you can listen to: `click`, `keyup`, `mousemove`, etc... AngularJS 1.x had one directive per event: `ng-click`, `ng-keyup`, `ng-mousemove`, etc... In Angular 2, this is simpler, no more directives to remember.

Going back to our `RacesCmp`, we now want to have a button that will display the races when clicked.

Reacting on an event can be done as follows:

```
<button (click)="onButtonClick()">Click me!</button>
```

A click on the button of the example above will trigger a call to the component method `onButtonClick()`.

Let's add this to our component:

```
@Component({
  selector: 'races-cmp',
  template: `
    <h2>Races</h2>
    <button (click)="refreshRaces()">Refresh the races list</button>
    <p>{{races.length}} races</p>
  `
})
export class RacesCmp {
  races: any = [];

  refreshRaces() {
    this.races = [{name: 'London'}, {name: 'Lyon'}];
  }
}
```

If you try this in your browser, you should initially see:

```
<ponyracer-app>
  <h1>PonyRacer</h1>
  <races-cmp>
    <h2>Races</h2>
    <button (click)="refreshRaces()">Show me the races</button>
    <p>0 races</p>
  </races-cmp>
</ponyracer-app>
```

And after your click, '0 races' should become '2 races'. Yay \o/

The statement can be a function call, but it can be any executable statement, or even a sequence of executable statements, like:

```
<button (click)="firstName = 'Cédric'; lastName = 'Exbrayat'">
  Click to change name to Cédric Exbrayat
</button>
```

However I would not advise you to do this. Using methods is a better way of encapsulating the behavior: it makes your code easier to maintain and test, and it makes the view simpler.

The cool thing is that it works with standard DOM events, but also with custom events that might fire from your Angular components or from web components. We'll see later how to fire custom events.

For the moment, let's say the `RacesCmp` component emits a custom event to notify the app that a new race is available.

Our template in the `PonyRacerApp` component would then look like:

```
@Component({
  selector: 'ponyracer-app',
  template: `
    <h1>PonyRacer</h1>
    <races-cmp (newRaceAvailable)="onNewRace()"></races-cmp>
  `,
  directives: [RacesCmp]
})
export class PonyRacerApp {
  onNewRace() {
    // add a flashy message for the user.
  }
}
```

We can easily figure that the `<races-cmp>` component has a custom event `newRaceAvailable` and that, when this event is fired, the method `onNewRace()` of our `PonyRacerApp` is called.

Angular will listen for the event on the element and on its children, so it will react to events that bubble. Consider the template:

```
<div (click)="onButtonClick()">
  <button>Click me!</button>
</div>
```

Even though the user clicks on the button embedded inside the div, the `onButtonClick()` method will be called, because the event bubbles up.

Oh, and you can access the event in the method called! You just have to pass `$event` to your method:

```
<div (click)="onButtonClick($event)">
  <button>Click me!</button>
</div>
```

Then you can handle the event in your component class:

```
onButtonClick(event) {
  console.log(event);
}
```

By default, the event will continue to bubble up, eventually triggering other event listeners up in the hierarchy.

You can use the event to prevent the default behavior and/or cancel propagation if you want:

```
onButtonClick(event) {
  event.preventDefault();
  event.stopPropagation();
}
```

One cool feature is that you can also easily handle keyboard events with:

```
<textarea (keydown.space)="onSpacePress()">Press space!</textarea>
```

Every time you will press the `space` key, the `onSpacePress()` method will be called. And you can do crazy combo, like `(keydown.alt.space)`, etc...

To conclude this part, I want to point that there is a big difference between something like:

```
<component [property]="doSomething()"></component>
```

and

```
<component (event)="doSomething()"></component>
```

In the first case, with property binding, the `doSomething()` value is called an expression, and will be evaluated at each change detection cycle to see if the property needs to be updated.

In the second case, however, with event binding, the `doSomething()` value is called a statement, and will be evaluated **only when the event is triggered**.

By definition they have completely different goals and, as you can suspect, they have different restrictions.

8.5. Expressions vs statements

Expressions and statements differ in several ways.

An expression will be executed many times, as part of the change detection. It should thus be as fast as possible. Basically, an Angular expression is a simplified version of an expression you could write in JavaScript.

If you are using `user.name` as an expression, `user` should be defined, otherwise Angular will throw an error.

An expression must be single: you can't chain several ones separated with a semi-colon.

An expression should not have any side effect. That means it can't be an assignment, for example.

```
<!-- forbidden, as the expression is an assignment -->
<!-- this will throw an error -->
<component [property]="user = 'Cédric'"></component>
```

It can not contains keywords, like `if`, `var`, etc...

A statement, on the other hand, is triggered on the matching event. If you try to use a statement like `race.show()` where `race` is `undefined`, you will have an error. You can chain several statements, separated with a semi-colon. A statement can, and generally should, have side effects. That's the point of reacting to an event: to make something happen. A statement can be a variable assignment, and can contain keywords.

8.6. Local variables

When I say that Angular will look in the component instance to find a variable, it is not technically correct. In fact, it will check the component instance and the local variables. Local variables are variables that you can dynamically declare in your template using the `#` syntax.

Let's say you want to display the value of an input:

```
<input type="text" #name>
{{ name.value }}
```

Using the `#` syntax, we are creating a local variable `name` referencing the DOM object `HTMLInputElement`. This local variable can be used anywhere in the template. As it has a `value` property, we can display this property in an interpolated expression. I'll come back to this example later.

Another useful usage of local variables is when you want to execute some kind of action on another element.

For example, you may want to give the focus on an element when you click on a button. This was a bit cumbersome in AngularJS 1.x, as you had to create a custom directive and so on.

The `focus()` method is a standard part of the DOM API, and we can leverage this. Using a local variable, it's a no-brainer in Angular 2:

```
<input type="text" #name>
<button (click)="name.focus()">Focus the input</button>
```

It can also be used with a custom component - one we created in our app, imported from another project, or even with a real Web Component:

```
<google-youtube #player></google-youtube>

<button (click)="player.play()">Play!</button>
```

Here, the button can start playing the video of the `<google-youtube>` component. This is actually a real Web Component written with Polymer! This component has a `play()` method that Angular will call when you click on the button, which is pretty cool!

Local variables have a few use cases, and we will gradually see them. One of them is described in the very next section.

8.7. Structural directives

Now, our `RacesCmp` is still not displaying the races :) The "proper way" in Angular 2 would imply to create another component `RaceCmp` to display each race. We are going to do something slightly simpler, and just write a simple `<ul><li>` list.

Property and event binding is great, but it does not let us change the DOM structure, like iterating over a collection and adding an element per item. To do so, we need to use **structural directives**. A directive in Angular is really close to a component, but does not have a template. It is used to add a behavior to an element.

The structural directives provided by Angular 2 rely on using a `template` element, a standard tag from the HTML specification:

```
<template>
  <div>Races list</div>
</template>
```

Here we have defined a template, displaying a simple `div`. Alone, it does not have much use, as the browser will not display it. But if we add one 'template' element in a view, then Angular 2 can use its content. The structural directives have the ability to do simple actions with this content, like displaying it or not, repeating it, etc...

Let's see which directives are available!

8.7.1. NgIf

We might want this template instantiated only if a condition is matched. For this, we will use the directive `ngIf`:

```
<template [ngIf]="races.length > 0">
  <div><h2>Races</h2></div>
</template>
```

The framework provides a few directives, like `ngIf`, and you can define yours if needed. We'll come back to custom directives later.

Here, the template will be instantiated only if `races` has at least one element, that is to say if there are races. As this syntax is a bit long, there is a shorter version:

```
<div *ngIf="races.length > 0"><h2>Races</h2></div>
```

And you will virtually always use this shorter version.

The syntax uses `*` to show it is a structural directive. The `ngIf` will now display or not the `div` whenever the value of `races` is changing: if there are no more races, the `div` will disappear.

The directives provided by the framework are already pre-loaded for us so we don't need to import and declare `NgIf` in the `directives` attribute of the `@Component` decorator.

```
import {Component} from 'angular2/core';

@Component({
  selector: 'races-cmp',
  template: `<div *ngIf="races.length > 0"><h2>Races</h2></div>`
})
export class RacesCmp {
  races: Array<any> = [];
}
```

8.7.2. NgFor

Working with real data will inevitably lead you to display a list of something. That's when `NgFor` proves very useful: it allows to instantiate one template per item in a collection. Our `RacesCmp` component contains a field `races` which, as you can probably guess, is an array of races to display.

```
import {Component} from 'angular2/core';

@Component({
  selector: 'races-cmp',
  template: `<div *ngIf="races.length > 0">
  <h2>Races</h2>
  <ul>
    <li *ngFor="#race of races">{{race.name}}</li>
  </ul>
  </div>`
})
export class RacesCmp {
  races: Array<any> = [{name: 'London'}, {name: 'Lyon'}];
}
```

And now we have a beautiful list, with one `li` tag per item in our collection!

```
<ul>
  <li>London</li>
  <li>Lyon</li>
</ul>
```

Note that `NgFor` is using a particular syntax, called a microsyntax.

```
<ul>
  <li *ngFor="#race of races">{{race.name}}</li>
</ul>
```

It is the equivalent of the more wordy (that we'll never use):

```
<ul>
  <template ngFor #race [ngForOf]="races">
    <li>{{race.name}}</li>
  </template>
</ul>
```

Here you can recognize:

- the `template` element to declare an inline template,
- the `NgFor` directive applied to it
- the `NgForOf` property where we feed the collection to display
- the local variable `#race` allowing us to use it in the interpolation expression, and reflecting the current element.

Instead of reminding all these parts, it is easier to use the shorter form:

```
<ul>
  <li *ngFor="#race of races">{{race.name}}</li>
</ul>
```

It is possible to declare another local variable bound to the index of the current element:

```
<ul>
  <li *ngFor="#race of races; #i=index">{{i}} - {{race.name}}</li>
</ul>
```

The local variable `i` will receive the index of the current element, starting at zero. `index` is an exported variable. Some directives export variables that you can then affect to a local variable to be able to use them in your template:

```
<ul>
  <li>0 - London</li>
  <li>1 - Lyon</li>
</ul>
```

There are also other exported variables that can be useful:

- `even`, a boolean that is true if the element has an even index
- `odd`, a boolean that is true if the element has an odd index
- `last`, a boolean that is true if the element is the last of the collection

8.7.3. NgSwitch

As you can guess from its name, this directive allows to switch between different templates based on a condition.

```
<div [ngSwitch]="messageCount">
  <p *ngSwitchWhen="0">You have no message</p>
  <p *ngSwitchWhen="1">You have a message</p>
  <p *ngSwitchDefault>You have some messages</p>
</div>
```

As you can see, `ngSwitch` takes a condition and the `*ngSwitchWhen` take the possible values. You can also have `*ngSwitchDefault` that will be displayed if none of the values matched.

8.8. Other template directives

Two other directives can be useful when writing a template, but they are not structural directives like the ones we just saw. These directives are standard directives.

8.8.1. NgStyle

The first one is `ngStyle`. We already saw that we can act on the style of an element using:

```
<p [style.color]="foreground">Friendship is Magic</p>
```

If you need to set several styles at the same time, you can use the `ngStyle` directive:

```
<div [ngStyle]="{fontWeight: fontWeight, color: color}">I've got style</div>
```

Note that the directive expects an object whose keys are the styles to set. The keys can either be in camelCase (`fontWeight`) or in dash-case (`'font-weight'`).

8.8.2. NgClass

In the same spirit, the `ngClass` directive allows to add or remove classes dynamically on an element.

As for the style, you can either set one class using property binding:

```
<div [class.awesome-div]="isAnAwesomeDiv()">I've got style</div>
```

Or, if you want to set several at the same time, you can use `ngClass`:

```
<div [ngClass]="{'awesome-div': isAnAwesomeDiv(), 'colored-div': isAColoredDiv()}">I've
got style</div>
```

8.9. Canonical syntax

Every syntax we have seen has a longer equivalent called the canonical syntax. This is mainly useful if your server side templating system is having trouble with the `[]` or `()` syntax, or if you really can't bear to use `[]`, `()`, `*`...

If you want to declare a property binding, you can do:

```
<pony-cmp [name]="pony.name"></pony-cmp>
```

or, using the canonical syntax:

```
<pony-cmp bind-name="pony.name"></pony-cmp>
```

For event binding, you can do:

```
<button (click)="onButtonClick()">Click me!</button>
```

or, using the canonical syntax:

```
<button on-click="onButtonClick()">Click me!</button>
```

And for local variables, you can use `var-`:

```
<input type="text" var-name>
<button on-click="name.focus()">Focus the input</button>
```

instead of the shorter form:

```
<input type="text" #name>
<button (click)="name.focus()">Focus the input</button>
```

8.10. Summary

The Angular 2 templating system gives us a powerful syntax to express the dynamic part of our HTML. It allows to express data and property binding, event binding and templating concerns, in a clear way, each with their own symbols:

- `{{}}` for interpolation

- `[]` for property binding
- `()` for event binding
- `#` for variable declaration
- `*` for structural directives

It provides a way to interact with standard Web Components like no other framework does. As there is no ambiguity between the various meanings, we will see our tools and IDEs gradually improve to give us meaningful warnings on what we are writing in our templates.

All these symbols are shorter versions of their canonical counterparts, which you can also use if you wish.

It takes some time to be fluent in this syntax, but you will soon be up to speed, and then it's easy to read and write.

Let's go through a complete example before moving on.

I want to write a `PoniesCmp` component, displaying a list of ponies. Each pony should be represented by a `PonyCmp` component, but we haven't seen yet how to pass parameters to a component. So, for now, we are going to display a simple list. The list should be displayed only if it's not empty, and I'd like to have some color for the even lines of my list. Finally, I want to be able to refresh the list with a button click.

Ready?

We start to write our component, in its own file:

```
import {Component} from 'angular2/core';

@Component({
  selector: 'ponies-cmp',
  template: ``
})
export class PoniesCmp {

}
```

You can add it to the `PonyRacerApp` component we wrote in the previous chapter to test it. You will have to import it, add it to the directives and insert the tag `<ponies-cmp></ponies-cmp>` in the template.

Our new component has a list of ponies:

```
import {Component} from 'angular2/core';

@Component({
  selector: 'ponies-cmp',
  template: ``
})
export class PoniesCmp {
  ponies: Array<any> = [{name: 'Rainbow Dash'}, {name: 'Pinkie Pie'}];
}
```

We are going to display this list, using `NgFor`:

```
import {Component} from 'angular2/core';

@Component({
  selector: 'ponies-cmp',
  template: `<ul>
    <li *ngFor="#pony of ponies">{{pony.name}}</li>
  </ul>`
})
export class PoniesCmp {
  ponies: Array<any> = [{name: 'Rainbow Dash'}, {name: 'Pinkie Pie'}];
}
```

One thing is missing, the refresh button:

```
import {Component} from 'angular2/core';

@Component({
  selector: 'ponies-cmp',
  template: `<ul>
  <button (click)="refreshPonies()">Refresh</button>
  <li *ngFor="#pony of ponies">{{pony.name}}</li>
  </ul>`
})
export class PoniesCmp {
  ponies: Array<any> = [{name: 'Rainbow Dash'}, {name: 'Pinkie Pie'}];

  refreshPonies() {
    this.ponies = [{name: 'Fluttershy'}, {name: 'Rarity'}];
  }
}
```

And of course, a touch of color to finish, with the use of `[style.color]` and the `even` exported variable:

```
import {Component} from 'angular2/core';

@Component({
  selector: 'ponies-cmp',
  template: `<ul>
    <button (click)="refreshPonies()">Refresh</button>
    <li *ngFor="#pony of ponies; #isEven=even"
      [style.color]="isEven ? 'green' : 'black'">
      {{pony.name}}
    </li>
  </ul>`
})
export class PoniesCmp {
  ponies: Array<any> = [{name: 'Rainbow Dash'}, {name: 'Pinkie Pie'}];

  refreshPonies() {
    this.ponies = [{name: 'Fluttershy'}, {name: 'Rarity'}];
  }
}
```

As you can see, we have used all the range of the templating syntax, and we have a perfectly working component. Our data are still hard-coded though: soon, we are going to see how to use a service to fetch them! This implies to learn about dependency injection first, so that we can use the HTTP service!

Chapter 9. Dependency injection

9.1. DI yourself

Dependency injection is a well-known design pattern. Let's take a component of our application. This component may need some features offered by other parts of our app (let's say a service). That's what we call a dependency. Instead of letting the component in charge of the creation of its dependencies, the idea is to let the framework create the dependency, and provide it to the component. That is known as "inversion of control".

It has several interesting features:

- it allows easy development, by just saying what we want and where we want it.
- it allows easy testing, by replacing dependencies with mock ones.
- it allows easy configuration, by swapping implementation.

It's a concept vastly used on the server side, but AngularJS 1.x was one of the first to use it on the frontend side.

9.2. Easy to develop

To be able to use dependency injection, we need a few things:

- a way to register a dependency, to make it available to injection in another component/service.
- a way to declare what dependencies are needed in the current component/service.

The framework does the rest of the job. When we declare a dependency in a component, it will look into the registry if it can find it, will get the instance of the dependency or create one, and actually inject it in our component.

A dependency can be a service provided by Angular, or a service we have written ourselves.

Let's take an example with the `Http` service provided by the framework. We'll have a look into it later, but you can already guess that the service will be used to communicate with a backend over HTTP.

Using TypeScript, it's easy to declare a dependency for our component, we just have to use the type system.

```
import {Http} from 'angular2/http';

export class RaceService {

  constructor(private _http: Http) {}

}
```

Angular will fetch the `Http` service for us and inject it into our constructor. When `RaceService` will be needed, the constructor will be called, and we will have a `_http` field referencing the `Http` service.

Now, we can add a method `list()` to our service, which will call our backend using the `Http` service:

```
import {Http} from 'angular2/http';

export class RaceService {

  constructor(private _http: Http) {}

  list() {
    return this._http.get('http://localhost:9000/races').map(res => res.json());
  }
}
```

To inform Angular 2 that this service has some dependencies itself, we need to add a class decorator: `@Injectable()`:

```
import {Injectable} from 'angular2/core';
import {Http} from 'angular2/http';

@Injectable()
export class RaceService {

  constructor(private _http: Http) {}

  list() {
    return this._http.get('http://localhost:9000/races').map(res => res.json());
  }
}
```

Here, we are using the `Http` service that is provided by Angular 2. But we still need to "register" the `Http` service, so as to make it available for injection.

An easy way to do this is to use the second parameter of the `bootstrap` method we saw earlier. Until

now, we just specified the main class of our app, but you can have an array as a second parameter, containing the list of what you want to make available for injection. `Http` itself needs some dependencies, so the easiest way to register everything necessary is to add `HTTP_PROVIDERS`, an array containing `Http` and its dependencies.

```
import {HTTP_PROVIDERS} from 'angular2/http';

// ...

bootstrap(PonyRacerApp, [HTTP_PROVIDERS]);
```

Now, if we want to make our `RaceService` available for injection in other services or components, we have to register it too:

```
import {HTTP_PROVIDERS} from 'angular2/http';
import {RaceService} from './services/race_service';

// ...

bootstrap(PonyRacerApp, [HTTP_PROVIDERS, RaceService]);
```

And we're done!

We can use our new service wherever we want. Let's test it in the `PonyRacerApp` component:

```
import {bootstrap} from 'angular2/platform/browser';
import {Component} from 'angular2/core';
import {HTTP_PROVIDERS} from 'angular2/http';
import {RaceService} from './services/race_service';

@Component({
  selector: 'ponyracer-app',
  template: '<h1>PonyRacer</h1>'
})
export class PonyRacerApp {

  // add a constructor with RaceService
  constructor(private _raceService: RaceService) {}

  list() {
    return this._raceService.list();
  }

}
bootstrap(PonyRacerApp, [HTTP_PROVIDERS, RaceService]);
```

As we don't have a backend answering our HTTP request, you should see a failed HTTP request if try to call the `list()` method.

Maybe we can do something about it…

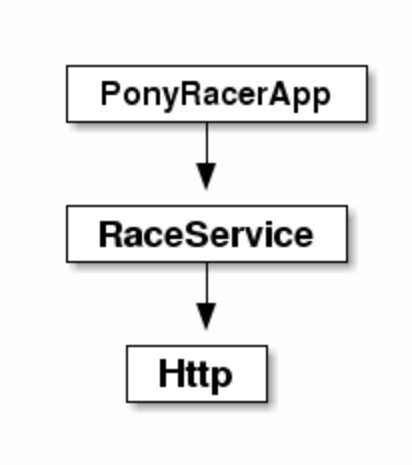
9.3. Easy to configure

I'll come back to the testability advantages brought by dependency injection in a following chapter, but we can have a look at the configuration problem. Here we are calling a backend that does not exist. Maybe the backend team is not ready yet, or you want to do it later. In any case, we would like to use some fake data.

DI provides a nice way to do this. Let's go back to the registration part:

```
bootstrap(PonyRacerApp, [HTTP_PROVIDERS, RaceService]);
```

We can represent the relations between component and services like this, where the arrows mean *depends on*:



In fact, what we wrote was the short form of:

```
import {provide} from 'angular2/core';

// ...

bootstrap(PonyRacerApp, [
  HTTP_PROVIDERS,
  provide(RaceService, {useClass: RaceService})
]);
```

We are telling the `Injector` that we want to create a link between a token (the type `RaceService`) and the class `RaceService`, using the `provide` method. The `Injector` is a service which keeps track of the injectable components by maintaining a registry and actually injects them when needed. The registry is a map that associates keys, called tokens, with classes. The tokens are not, like in many dependency injection frameworks, necessarily Strings. They can be anything, like Type references, for example. And that will usually be the case.

Since, in our example, the token and the class to inject are the same, you can write the same thing in the shorter form:

```
bootstrap(PonyRacerApp, [HTTP_PROVIDERS, RaceService]);
```

The token has to uniquely identify the dependency.

This injector is returned by the `bootstrap` promise, so we can play with it:

```
import {provide} from 'angular2/core';

// ...

bootstrap(PonyRacerApp, [
  HTTP_PROVIDERS,
  provide(RaceService, {useClass: RaceService}),
  // let's add another provider to the same class
  // with another token
  provide('RaceServiceToken', {useClass: RaceService})
]).then(
  // and play with the returned injector
  appRef => playWithInjector(appRef.injector)
);
```

The interesting part is in the `playWithInjector` function.

```
function playWithInjector(inj) {
  console.log(inj.get(RaceService));
  // logs "RaceService {_http: Http}"
  console.log(inj.get('RaceServiceToken'));
  // logs "RaceService {_http: Http}" again
  console.log(inj.get(RaceService) === inj.get(RaceService));
  // logs "true", as the same instance is returned every time for a token
  console.log(inj.get(RaceService) === inj.get('RaceServiceToken'));
  // logs "false", as the providers are different,
  // so there are two distinct instances
}
```

As you can see, we can ask the injector for a dependency with the `get` method and a token. As I have declared the `RaceService` twice, with two different tokens, we have two providers. The injector will create an instance of `RaceService` the first time it is asked to for a specific token, and then returns the same instance for this token every time. It will do the same for each provider, so here we will actually have two instances of `RaceService` in our app, one for each token.

However, you will not use the token very often, or even at all. In TypeScript, you rely on the types to get the job done, so the token is a Type reference, usually bound to the corresponding class. If you want to use another token, you have to use the decorator `@Inject()`: see the last part of this chapter for more information about this.

This whole example was just to point out a few things:

- a provider links a token to a service.
- the injector returns the same instance every time it is asked the same token.

- we can have a token name different than the class name

The fact that the instance returned is created on the first call and then always the same is also a well-known design pattern: it's called a *singleton*. This is really useful, as you can share information between components using a service, and you will be sure they share the same service instance.

Now, back to our fake `RaceService` problem. I can write a new class, doing the same job as `RaceService` but returning hardcoded data:

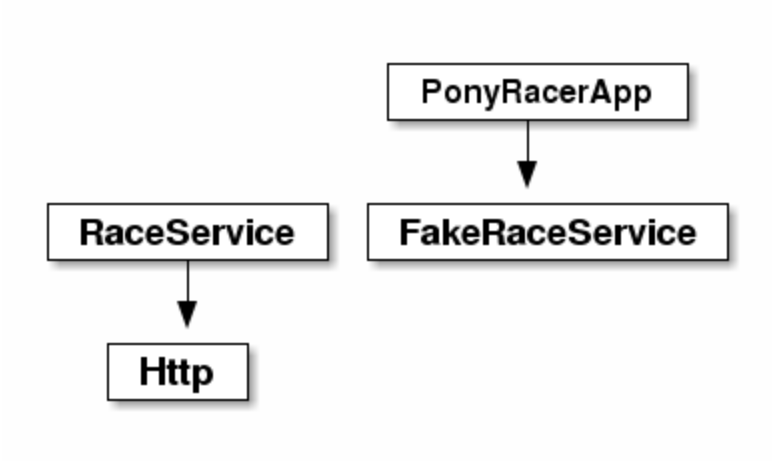
```
class FakeRaceService {
  list() {
    return Observable.of([{ name: 'London' }]);
  }
}
```

Don't worry too much about the `Observable` stuff: we'll see that later in the reactive programming chapter. It's just to return the same result type as the `Http` service. We can use the provider declaration to replace `RaceService` with our `FakeRaceService`:

```
import {provide} from 'angular2/core';

// ...

bootstrap(PonyRacerApp, [
  HTTP_PROVIDERS,
  // we provide a fake service
  provide(RaceService, {useClass: FakeRaceService})
])
```

If you restart your app, you will see that there is no HTTP error as before, because our app is using the fake service instead of the first one!

Now we have a relation like this:



That can be really useful when you test your app manually or, as we will soon see, when you are writing automated tests.

9.4. Other types of provider

In our example, we might want to use `FakeRaceService` when we are developing our app, and use the real `RaceService` when we are in production. You can change it manually of course, but you can also use another type of provider: `useFactory`.

```
import {provide} from 'angular2/core';

// ...

// we just have to change this constant when going to prod
const IS_PROD = false;

bootstrap(PonyRacerApp, [
  HTTP_PROVIDERS,
  // we provide a factory
  provide(RaceService, {useFactory: () => IS_PROD ? new RaceService() : new
FakeRaceService()})
])
```

In this example, we are using `useFactory` instead of `useClass`. A factory is a function with one job, creating an instance. Our example tests a constant and returns the fake service or the real service.

But wait, if we switch back to the real service, as we are using `new` to create the `RaceService`, it will not have its `Http` dependency instantiated! Right, if we want to make this example work, we have to pass an `Http` instance to the constructor call. Good news: `useFactory` can be used with another property named `deps`, where you can specify an array of dependencies:

```
import {provide} from 'angular2/core';

// ...

// we just have to change this constant when going to prod
const IS_PROD = true;

bootstrap(PonyRacerApp, [
  HTTP_PROVIDERS,
  // we provide a factory
  provide(RaceService, {
    // the http instance will be injected in the factory
    // so we can pass the http instance to RaceService
    useFactory: http => IS_PROD ? new RaceService(http) : new FakeRaceService(),
    // array with the dependencies needed
    deps: [Http]
  })
])
```

Hooray!

NOTE	Be careful, the order of the parameters should be the same as the order in the array if you have several dependencies!

Of course, this example is just to demonstrate the use of `useFactory` and its dependencies. You could, and should, write:

```
import {provide} from 'angular2/core';

// ...

// we just have to change this constant when going to prod
const IS_PROD = true;

bootstrap(PonyRacerApp, [
  HTTP_PROVIDERS,
  provide(RaceService, {useClass: IS_PROD ? RaceService : FakeRaceService})
])
```

Declaring a constant for `IS_PROD` is really bothering: maybe we can use dependency injection too? I'm pushing things a bit as you can see :) You don't necessarily need to force all things in DI, but this is just to show you another provider type: `useValue`.

```
import {provide} from 'angular2/core';

// ...

bootstrap(PonyRacerApp, [
  HTTP_PROVIDERS,
  provide('IS_PROD', {useValue: true}),
  provide(RaceService, {
    useFactory: (IS_PROD, http) => IS_PROD ? new RaceService(http) : new FakeRaceService
(),
    deps: ['IS_PROD', Http]
  })
])
```

There is one last type of provider. Do you remember when I was playing with the injector and declaring two providers for the same class with two different tokens? I hope so, that was like 5 minutes ago. I said that each provider would be resolved to its own `RaceService` instance.

But how can we declare *two different tokens* referencing the *same instance*?

That's where `useExisting` comes into play.

```
import {provide} from 'angular2/core';

// ...

bootstrap(PonyRacerApp, [
  HTTP_PROVIDERS,
  provide('RaceServiceToken', {useClass: RaceService}),
  // let's add another token this time using `useExisting`
  provide(RaceService, {useExisting: 'RaceServiceToken'})
])
```

The results of `playWithInjector` are slightly different:

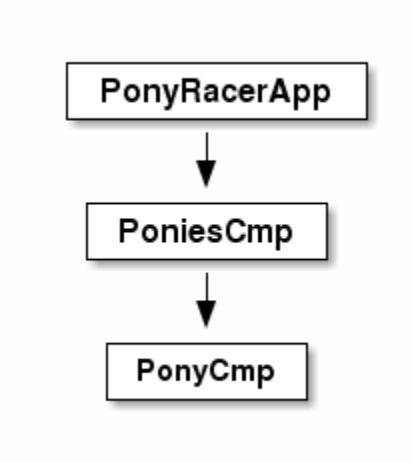
```
function playWithInjector(inj) {
  console.log(inj.get(RaceService));
  // logs "RaceService {_http: Http}"
  // same as we had before
  console.log(inj.get('RaceServiceToken'));
  // logs "RaceService {_http: Http}" again
  // same as we had before
  console.log(inj.get(RaceService) === inj.get(RaceService));
  // logs "true", as the same instance is returned every time for a token
  // same as we had before
  console.log(inj.get('RaceServiceToken') === inj.get(RaceService));
  // logs "true", as the second provider is for an existing token,
  // so there is just one instance
  // where it was "false" when using `useClass`
}
```

Not something you will need every day, though.

9.5. Hierarchical injectors

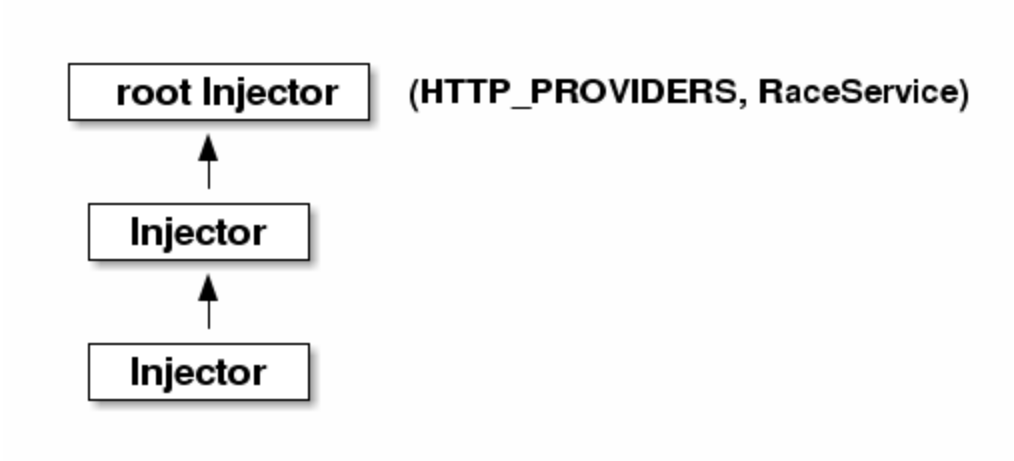
One last crucial thing to understand in Angular 2: there are several injectors in your app. In fact, there is one injector per component, and this injector inherits from the injector of its parent.

Let's say we have an app looking like:



We have a root component `PonyRacerApp`, with a child component `PoniesCmp`. `PoniesCmp` itself has a child component: `PonyCmp`.

When we bootstrap the app, we create the root injector. Then, every component will create its own injector, inheriting its parent one.



It means that when we are declaring a dependency in a component, Angular 2 will begin its search in the current injector. If it finds the dependency, perfect, it returns it. If not, it will do the same in the parent injector, and again, until it finds the dependency. It it doesn't, it will throw an exception.

From this, we can deduce two things:

- the dependencies declared in the root injector are available for every component in the app. For example, `HTTP_PROVIDERS` and `RaceService` can be used everywhere.
- we can declare dependencies at another level than the root component. How do we do this?

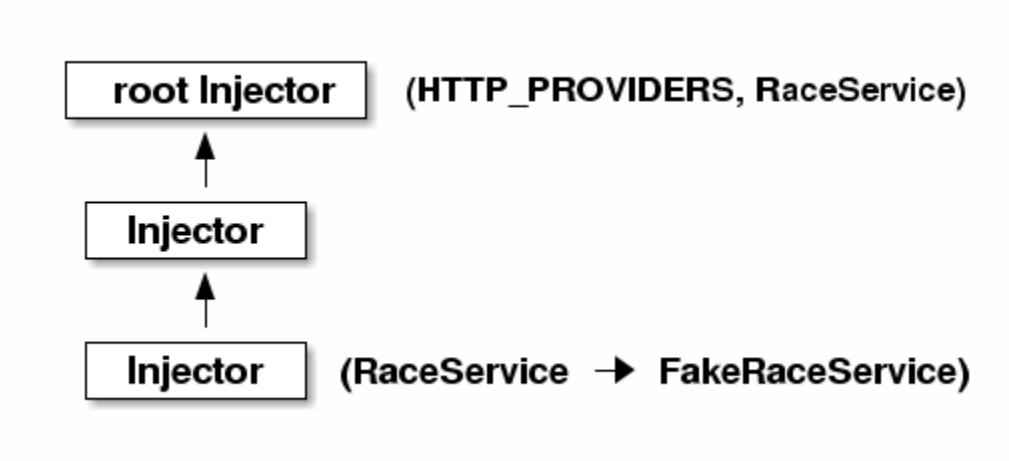
The `@Component` decorator can take another configuration option, called `providers`. This `providers` attribute can take an array with a list of dependencies, as we did for the `bootstrap` method.

We can imagine a `RacesCmp` that would declare its own `RaceService` provider:

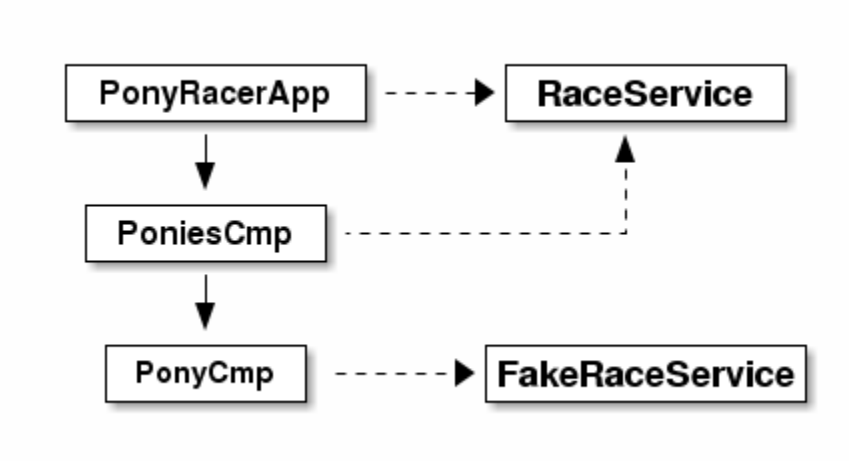
```
@Component({
  selector: 'races-cmp',
  providers: [provide(RaceService, {useClass: FakeRaceService})],
  template: `<strong>Races list</strong>`
})
export class RacesCmp {

  constructor(private _raceService: RaceService) {}

  list() {
    return this._raceService.list();
  }
}
```

In this component, the provider with the token `RaceService` will always give an instance of `FakeRaceService`, whatever was defined in the root injector. It's really useful if you want to have a different instance of a service for a given component, or if you want to have perfectly encapsulated components that declare everything they need.

Here we have:



The injection will then be resolved as:



As a rule of thumb, if only one component needs to have access to a service, it's a good idea to only provide this service in the component's injector, using the `providers` attribute. If the dependency can be used by the whole app, declare it in the bootstrap method.

9.6. Binding multiple values

As you saw, you can have only one value per token: if you define another one, it will replace the first binding. But, sometimes, you want to declare a collection of values and be able to add an element to the collection.

This mechanism is used in a few places as we'll see: to add a custom pipe or a custom directive or a custom validator to the default ones...

To do this, we have a useful option called `multi`. By default, this option is `false`; but if a binding is declared using `multi: true`, then it's possible to add other values to this binding later.

For example, we can declare a token `'greetings'` bound to the value `'hello'`. Later we can add another

value to the `'greetings'` collection:

```
bootstrap(PonyRacerApp, [
  provide('greetings', {useValue: 'hello', multi: true}),
  // let's add another value to the collection
  provide('greetings', {useValue: 'hi', multi: true})
])
```

And, as you can see below, the injector returns the collection of values:

```
function playWithInjector(inj) {
  console.log(inj.get('greetings'));
  // logs "['hello', 'hi']"
}
```

It will be very handy in a few minutes!

9.7. DI without types

If you want to use a token and not rely on TypeScript types, you will have to add a decorator for each dependency you want to inject. The decorator is `@Inject()` and receives the token of the dependency you want to inject. The same `RaceService` can be written as follow to use an `Http` service registered with a string token `FakeHttp`:

```
import { Injectable, Inject } from 'angular2/core';
import { Http } from 'angular2/http';

@Injectable()
export class RaceService {

  constructor(@Inject(Http) http) {
    this._http = http;
  }

  list() {
    return this._http.get('http://localhost:9000/races').map(res => res.json());
  }
}
```

To make the decorators work in this example, you'll have to use babel with the `es7.decorator` and the plugin `angular2-annotations`.

Chapter 10. Pipes

10.1. Pied piper

Sometimes the raw data is not what we want to display in the view. We often want to transform them, filter them, limit their number, etc... AngularJS 1.x had a very handy feature to do this, very badly named 'filters'. Lessons have been learned and now these data transformers have a meaningful name! Nah, I'm just kidding, they are called 'pipes' :).

A pipe can be used either in HTML or in your applicative code. Let's take an example and see how we can use it.

10.2. json

A pipe that is not really useful in a production app, but very handy when you are debugging your app, is `JsonPipe`. Basically, this pipe applies `JSON.stringify()` to your data. If you have some data in your component, an array of ponies called `ponies`, for example, and you want to quickly see what's inside, you may want to try something like:

```
<p>{{ ponies }}</p>
```

Tough luck, it's going to display `[object Object]`...

But `JsonPipe` is here to rescue us. You can use it in your HTML, in any expression:

```
<p>{{ ponies | json }}</p>
```

And it will display the JSON representation of your object:

```
<p>[ { "name": "Rainbow Dash" }, { "name": "Pinkie Pie" } ]</p>
```

You can see where the name 'pipe' is coming from. To use a pipe, you have to add a pipe (`|`) character after your data, and then the name of the pipe you want to use. The expression is evaluated and the result goes through the pipe. It's possible to chain several pipes, one after another, like:

```
<p>{{ ponies | slice:0:2 | json }}</p>
```

We'll come back to the `slice` pipe, but you can see that we are chaining the `slice` pipe and then the `json` one.

You can use it in an interpolation expression or in a property expression, but **not** in an event statement.

```
<p [textContent]="ponies | json"></p>
```

You can also use it in your code, via dependency injection:

```
import {Component} from 'angular2/core';
// you need to import the pipe you want to use
import {JsonPipe} from 'angular2/common';

@Component({
  selector: 'ponies-cmp',
  template: `<p>{{poniesAsJson}}</p>`
})
export class PoniesCmp {
  ponies: Array<any> = [{name: 'Rainbow Dash'}, {name: 'Pinkie Pie'}];

  poniesAsJson: string;

  // inject the Pipe you want
  constructor(jsonPipe: JsonPipe) {
    // and then call the transform method on it
    this.poniesAsJson = jsonPipe.transform(this.ponies);
  }
}
```

As this will be the same for every pipe, I will now just show you the HTML examples using interpolation.

10.3. slice

If you want to display just a part of a list, `slice` is your friend. It works like the `slice` method in JavaScript, and takes two arguments: a start index and, optionally, an end index.

To pass an argument to a pipe, you have to add a colon `:`, then the first argument, then possibly, another colon and the second argument etc...

```
<p>{{ ponies | slice:0:2 | json }}</p>
```

This example will display the first two elements of my list of ponies.

`slice` works with arrays and strings, so you can also truncate a string:

```
<p>{{ 'Ninja Squad' | slice:0:5 }}</p>
```

and that will display only 'Ninja'.

You can give the `slice` pipe only one index n, and it will take the elements from n to the end.

```
<p>{{ 'Ninja Squad' | slice:3 }}</p>
<!-- will display 'ja Squad' -->
```

If you give it a negative integer, it will take the n **last** elements.

```
<p>{{ 'Ninja Squad' | slice:-5 }}</p>
<!-- will display 'Squad' -->
```

As we saw, you can also give the pipe an end index: it will take the elements until this index. If this index is negative, it will take the elements until the index, but starting from the end.

```
<p>{{ 'Ninja Squad' | slice:2:-2 }}</p>
<!-- will display 'nja Squ' -->
```

As you can use `slice` in any expression, you can use it even with `NgFor`:

```
import {Component} from 'angular2/core';

@Component({
  selector: 'ponies-cmp',
  template: `<div *ngFor="#pony of ponies | slice:0:2">{{pony.name}}</div>`
})
export class PoniesCmp {
  ponies: Array<any> = [
    {name: 'Rainbow Dash'},
    {name: 'Pinkie Pie'},
    {name: 'Fluttershy'}
  ];
}
```

The component will create only two `div` here, for the first two ponies, as we have applied the `slice` pipe to the collection.

The pipe argument can of course be a dynamic value:

```
import {Component} from 'angular2/core';

@Component({
  selector: 'ponies-cmp',
  template: `<div *ngFor="#pony of ponies | slice:0:size">{{pony.name}}</div>`
})
export class PoniesCmp {
  size: number = 2;
  ponies: Array<any> = [
    {name: 'Rainbow Dash'},
    {name: 'Pinkie Pie'},
    {name: 'Fluttershy'}
  ];
}
```

You can use this to create a dynamic display where your user chooses how much elements she/he wants to see.

10.4. uppercase

As its name makes it clear enough, this pipe transforms a string into its uppercase version:

```
<p>{{ 'Ninja Squad' | uppercase }}</p>
<!-- will display 'NINJA SQUAD' -->
```

10.5. lowercase

The counterpart of the previous one, this pipe transforms a string into its lowercase version:

```
<p>{{ 'Ninja Squad' | lowercase }}</p>
<!-- will display 'ninja squad' -->
```

10.6. number

This pipe allows to format a number.

It takes one parameter, a string, formatted as `{integerDigits}.{minFractionDigits}-{maxFractionDigits}`, but every part is optional. Each part indicates:

- how many numbers you want in the integer part
- how many numbers you want at least in the decimal part

- how many numbers you want at most in the decimal part

A few examples, starting with what we have with no pipe:

```
<p>{{ 12345 }}</p>
<!-- will display '12345' -->
```

Using the `number` pipe will group the integer part, even with no digits required:

```
<p>{{ 12345 | number }}</p>
<!-- will display '12,345' -->
```

The `integerDigits` parameter will left-pad the integer part with zeros if needed:

```
<p>{{ 12345 | number:'6.' }}</p>
<!-- will display '012,345' -->
```

The `minFractionDigits` is the minimum size of the decimal part, so it will pad zeros on the right until reached:

```
<p>{{ 12345 | number:'.2' }}</p>
<!-- will display '12,345.00' -->
```

The `maxFractionDigits` is the maximum size of the decimal part. You have to specify a `minFractionDigits`, even at 0, if you want to use it. If the number has more decimals than that, then it is rounded:

```
<p>{{ 12345.13 | number:'.1-1' }}</p>
<!-- will display '12,345.1' -->

<p>{{ 12345.16 | number:'.1-1' }}</p>
<!-- will display '12,345.2' -->
```

WARNING	The pipe (as well as `percent`, `currency` and `date`) relies on the Internationalization API of the browser, and this API is not available in every browser right now. This is a known problem, and I'm sure it will be fixed soon.

10.7. percent

Based on the same principle as `number`, `percent` allows to display... a percentage!

```
<p>{{ 0.8 | percent }}</p>
<!-- will display '80%' -->

<p>{{ 0.8 | percent:'.3' }}</p>
<!-- will display '80.000%' -->
```

10.8. currency

As you can imagine, this pipe allows to format an amount of money in the currency you want. You have to give it at least one parameter:

- the ISO string representing the currency ('EUR', 'USD'...)
- optionally, a boolean flag to say if you want to use the symbol ('€', '$') or the ISO string. By default, the flag is false, and the symbol will not be used.
- optionally also, a string to format the amount, using the same syntax as `number`.

```
<p>{{ 10.6 | currency:'EUR' }}</p>
<!-- will display 'EUR10.6' -->

<p>{{ 10.6 | currency:'USD':true }}</p>
<!-- will display '$10.6' -->

<p>{{ 10.6 | currency:'USD':true:'.2' }}</p>
<!-- will display '$10.60' -->
```

10.9. date

The `date` pipe formats a date value to a string of the desired format. The date can be a `Date` object or a number of milliseconds. The format specified can be either a pattern like 'ddMMyyyy', 'MMyy' or one of the predefined symbolic names available like 'short', 'longDate', etc...:

```
<p>{{ birthday | date:'ddMMyyyy' }}</p>
<!-- will display '07/16/1986' -->

<p>{{ birthday | date:'longDate' }}</p>
<!-- will display 'July 16, 1986' -->
```

Of course, you can also display the time portion of the date:

```
<p>{{ birthday | date:'HHmmss' }}</p>
<!-- will display '15:30:00' -->

<p>{{ birthday | date:'shortTime' }}</p>
<!-- will display '3:30 PM' -->
```

WARNING

When you specify a pattern, only the components of the pattern matter: the separator and the order are locale-dependent. That means:

```
<p>{{ birthday | date:'ddMMyyyy' }}</p>
<p>{{ birthday | date:'dd-MM-yyyy' }}</p>
<p>{{ birthday | date:'MM-dd-yyyy' }}</p>
```

will all display the same result (`07/16/1986` for me, but well, it depends of your locale of course).

10.10. async

The `async` pipe allows data obtained asynchronously to be displayed. Under the hood, it uses `PromisePipe` or `ObservablePipe` depending if your async data comes from a Promise or an Observable. I hope you now know what a Promise is (otherwise go back to the ES6 chapter), and we'll come back to Observable quickly.

The async pipe returns an empty string until the data is finally available (i.e. until the promise is resolved, in case of a promise). Once resolved, the resolved value is returned. More importantly, it triggers a change detection check once the data is available.

The following example uses a Promise:

```
import {Component} from 'angular2/core';

@Component({
  selector: 'greeting-cmp',
  template: `<div>{{ asyncGreeting | async}}</div>`
})
export class GreetingCmp {
  asyncGreeting: Promise<string> = new Promise(resolve => {
    // after 1 second, the promise will resolve
    window.setTimeout(() => resolve('hello'), 1000);
  });
}
```

You can see the `async` pipe is applied to the variable `asyncGreeting`. This one is a promise, resolved after 1 second. Once the promise is resolved, our browser will display:

```
<div>hello</div>
```

Even more interesting, if the source is an Observable, then the pipe will do the unsubscribe part itself when the component is destroyed (for example when the user navigates to another component).

10.11. Creating your own pipes

Of course, you can also create your own pipes. That's sometimes very useful. In AngularJS 1.x, we often used custom filters. For example, we built one to display how much time elapsed since an action the user did (like `12 seconds ago` or `3 days ago`) in several of our apps. Let's see how we would do this in Angular 2!

First we need to create a new class. It should implement the `PipeTransform` interface, which forces us to have a `transform()` method, the one doing the heavy lifting.

Does not sound too hard, let's give it a try!

```
import {PipeTransform, Pipe} from 'angular2/core';

export class FromNowPipe implements PipeTransform {
  transform(value, args) {
    // do something here
  }
}
```

We are going to use Moment.js `fromNow` function to display how much time has elapsed since the date.

You can install Moment.js using `NPM` if you want:

```
npm install moment
```

And add it to our SystemJS configuration:

```
map: {
  'angular2': 'node_modules/angular2',
  'rxjs': 'node_modules/rxjs',
  'moment': 'node_modules/moment/moment'
}
```

As we are using TypeScript, we also need the interface:

```
tsd install moment --save
```

and then add `/// <reference path="typings/moment/moment.d.ts" />` to the file `ponyracer_app.ts`.

```
import {PipeTransform, Pipe} from 'angular2/core';
import * as moment from 'moment';

export class FromNowPipe implements PipeTransform {
  transform(value, args) {
    return moment(value).fromNow();
  }
}
```

Now, we need to register the pipe in our app. For this, there is a special decorator we can use: `@Pipe`.

```
import {PipeTransform, Pipe} from 'angular2/core';
import * as moment from 'moment';

@Pipe({name: 'fromNow'})
export class FromNowPipe implements PipeTransform {
  transform(value, args) {
    return moment(value).fromNow();
  }
}
```

The chosen name will be the one allowing to use the pipe in the template. To use the pipe in a template, you then have to declare it in the `@Component` decorator attribute `pipes`:

```
@Component({
  selector: 'test-cmp',
  template: `
  <p>{{ birthday | fromNow }}</p>
  <!-- will display 'one month ago' -->
  `,
  pipes: [FromNowPipe]
})
class TestCmp {
  birthday: Date = new Date('2015-07-16T15:30:00');
}
```

Voilà, you have your first pipe!

As a small bonus, you can make your pipe available everywhere and remove the burden of having to declare it in the `pipes` attribute of the `@Component` decorator.

For this, you can add your custom pipes to the default collection of pipes which is loaded for us at startup and makes these pipes available in our templates without having to declare them.

This collection is called `PLATFORM_PIPES` and it contains all the pipes we saw. To add our custom pipe we use dependency injection, as usual, with a provider to add a new value to the existing collection.

And, maybe you've guessed, we are going to use the `multi` option:

```
bootstrap([
    provide(PLATFORM_PIPES, {useValue: FromNowPipe, multi: true})
])
```

Now we don't have to declare the `pipes` attribute anymore:

```
@Component({
  selector: 'test-cmp',
  template: `
  <p>{{ birthday | fromNow }}</p>
  <!-- will display 'one month ago' -->
  `
})
class TestCmp {
  birthday: Date = new Date('2015-07-16T15:30:00');
}
```

Chapter 11. Reactive Programming

11.1. Call me maybe

You may have heard of reactive or functional reactive programming lately. It has become quite popular in several languages platforms, like in .Net with the *Reactive Extensions* library, which is now available in pretty much every language (RxJava, RxJS, etc…).

Reactive programming is not really new. It is a way to build an app using events and reacting to them (hence the name). The events can be composed, filtered, grouped, etc… using functions like `map`, `filter`, etc… That's why you sometimes find the terms "functional reactive programming". But, to be accurate, reactive programming is not really functional programming, as it does not necessarily include the concepts of immutability, the lack of side-effects etc… Reacting on events is something you may have done:

- in the browser, when setting listeners to user events;
- on the backend side, reacting to events coming from a message bus.

In reactive programming, all data coming in will be in a stream. These streams can be listened to, modified of course (filtered, merged…), and can even become a new stream that can be listened to. This technique allows for fairly decoupled programs: you don't have to worry much about the consequences of your method call, you just raise an event, and every part of your app interested in this business will react accordingly. And maybe one of these parts will raise an event also, etc…

Now, why am I telling you about that? What does it have to do with Angular 2?

Well, Angular 2 is built using reactive programming, and we will use this technique for some parts as well. Reacting on a HTTP request? Reactive programming. Spawning a custom event for our component? Reactive programming. Dealing with value changes in our forms? Reactive programming.

So let's focus on this topic for a few minutes. Nothing hard to handle, but it's better to have a clear mind on this.

11.2. General principles

In reactive programming, everything is a stream. A stream is an ordered sequence of events. These events represent values (look, another value!), errors (that went bad) or completion events (ok, I'm done). All these are pushed from the data producer to the consumer. As a developer, your job will be to *subscribe* to these streams, i.e. defining a listener capable of handling the three possibilities. Such a listener is called an *observer*, and the stream, an *observable*. These terms were coined a long time ago, as it is a well-known design pattern: the *observer* pattern.

They are different from promises, even if they look a bit similar, as they both handle asynchronous values. But an observer is not a one-time thing like a promise: it will continue to listen until it receives

a 'completion' event.

For now, observables aren't part of the official ECMAScript specification, but they might be part of ES7 (ES2016), as there is an effort done to standardize it.

Observables are very close to arrays. An array is a collection of values, like an observable. An observable only adds the concept of values over time: in an array, you have all the values at once, while the values will come over time in an observable, maybe every few minutes.

The most popular reactive programming library in the JavaScript ecosystem is RxJS. And that's the one that Angular 2 relies on and lets us use.

So let's have a look.

11.3. RxJS

Every observable, just like every array, can be transformed using functions you may have already encountered:

- `take(n)` will pick the n first events.
- `map(fn)` will apply `fn` to each event and return the result.
- `filter(predicate)` will only let through the events that fulfill the predicate.
- `reduce(fn)` will apply `fn` to every event to reduce the stream to a single value.
- `merge(s1, s2)` will merge the streams.
- `subscribe(fn)` will apply `fn` to each event it receives.
- and much more...

So, if you take an array of numbers and want to multiply each by 2, filter those under 5, and print them, you can do:

```
[1, 2, 3, 4, 5]
  .map(x => x * 2)
  .filter(x => x > 5)
  .forEach(x => console.log(x)); // 6, 8, 10
```

RxJS let us build an observable from an array. And, as you can see, we can do the exact same thing:

```
Observable.fromArray([1, 2, 3, 4, 5])
  .map(x => x * 2)
  .filter(x => x > 5)
  .subscribe(x => console.log(x)); // 6, 8, 10
```

But an observable is more than a collection. It is an asynchronous collection, where the events arrive over time. A good example is browser events. They will happen over time, so they are a good candidate to use an observable. Here is an example using jQuery:

```
let input = $('input');

Observable.fromEvent(input, 'keyup')
  .subscribe(() => console.log('keyup!'));

input.trigger('keyup'); // logs "keyup!"
input.trigger('keyup'); // logs "keyup!"
```

You can build observables from AJAX requests, browser events, Web sockets responses, a promise, whatever you can think of. And from a function of course:

```
let observable = Observable.create((observer) => observer.next('hello'));

observable.subscribe((value) => console.log(value));
// logs "hello"
```

`Observable.create` takes a function that will emit events on the `observer` given as parameter. Here it simply emits one event for the demonstration.

You can also handle errors, because your observable may go wrong. The `subscribe` method can take another callback, one designed to handle errors.

Here the `map` method throws an exception, so the second handler of the `subscribe` method will log it.

```
Observable.range(1, 5)
  .map(() => {
    throw new Error('something went wrong');
  })
  .filter(x => x > 5)
  .subscribe(x => console.log(x), error => console.log(error)); // something went wrong
```

Once the observable is done, it will emit a completion event that you can catch with a third handler. Here, the `range` method we are using to create the events will iterate from 1 to 5 and then emit the 'completed' signal:

```
Observable.range(1, 5)
  .map(x => x * 2)
  .filter(x => x > 5)
  .subscribe(x => console.log(x), error => console.log(error), () => console.log('done'
));
// 6, 8, 10, done
```

And you can do many, many things with an observable:

- transformation (delaying, debouncing…)
- combination (merge, zip, combineLatest…)
- filtering (distinct, filter, last…)
- maths (min, max, average, reduce…)
- conditions (amb, includes…)

We would need a whole book to go through it all! If you want to go further, have a look at the Rx Book. It contains the best introduction I've found on the subject. And if you want to have a good visual representation of what each function does, go to rxmarbles.com.

Now let's have a look at how we will use observables in Angular 2.

11.4. Reactive programming in Angular 2

Angular 2 uses RxJS, and it allows us to use it too. The framework provides an adapter around the `Observable` object: `EventEmitter`. The `EventEmitter` has a method `subscribe()` to react to events and this method can receive three parameters:

- a method to react on events.
- a method to react on errors.
- a method to react on completion

The `EventEmitter` can emit an event by calling the `emit()` method.

```
let emitter = new EventEmitter();

emitter.subscribe(
  value => console.log(value),
  error => console.log(error),
  () => console.log('done')
);

emitter.emit('hello');
emitter.emit('there');
emitter.complete();

// logs "hello", then "there", then "done"
```

Note that the `subscribe` method returns a subscription object, with a method `unsubscribe` to... unsubscribe.

```
let emitter = new EventEmitter();

let subscription = emitter.subscribe(
  value => console.log(value),
  error => console.log(error),
  () => console.log('done')
);

emitter.next('hello');
subscription.unsubscribe(); // unsubscribe
emitter.next('there');

// logs "hello" only
```

Now that we know a little bit more about reactive programming, and the `EventEmitter`, let's see how Angular 2 uses it.

Chapter 12. Building components and directives

12.1. Introduction

So far, we have seen some small components. And of course, you can feel that, as they are the backbone of our apps, they can be more complex than what we have seen. How do we pass data? How do we manage the lifecycle of our component? What are the good practices to build these things?

Directives: What do they do? Do they do things? Let's find out!

12.2. Directives

A directive is very much like a component, except it does not have a template. In fact, the `Component` class inherits from the `Directive` class in the framework.

So it makes sense to start by studying directives, as everything we will see regarding directives also applies to components. We will look into the configuration options you are most likely to use. The more advanced ones are in a later chapter, ready for you when you master the basics.

As for a component, your directive will be annotated with a decorator, but instead of `@Component`, it will be `@Directive`.

Directives are very small pieces. You can think of them as decorators for your HTML: they will attach a behavior to elements in the DOM. You can have multiple directives on the same element.

A directive must have a CSS selector, which indicates to the framework where to activate it in our template.

12.2.1. Selectors

Selectors can be of various types:

- an element, as it's usually the case for components: `footer`.
- a class, not so frequent: `.alert`.
- an attribute, the most frequent for directives: `[color]`.
- an attribute with a specific value: `[color=red]`.
- a combination of the above: `footer[color=red]` matches an element named `footer` having an attribute `color` whose value is `red`. `[color],footer.alert` matches any element having an attribute `color` or (`,`) any element named `footer` with the CSS class `alert`. `footer:not(.alert)` matches any element named `footer` that does not (`:not()`) have the CSS class `alert`.

For example, this is a very simple directive that does nothing but gets activated if the attribute `doNothing` is on an element:

```
@Directive({
  selector: '[doNothing]'
})
export class DoNothingDirective {

  constructor() {
    console.log('Do nothing directive');
  }
}
```

Such a directive will be activated in a component like this `TestCmp`:

```
@Component({
  selector: 'test-cmp',
  template: '<div doNothing>Click me</div>',
  directives: [DoNothingDirective]
})
export class TestCmp {

}
```

A more complex selector could be:

```
@Directive({
  selector: 'div.loggable[logText]:not([notLoggable=true])'
})
export class ComplexSelectorDirective {

  constructor() {
    console.log('Complex selector directive');
  }
}
```

Here it will match all `div` elements with a `loggable` class and a `logText` attribute that don't have an attribute `notLoggable` with a `true` value.

So this template will trigger the directive:

```
<div class="loggable" logText="text">Hello</div>
```

But this one will not:

```
<div class="loggable" logText="text" notLoggable="true">Hello</div>
```

Let's be honest, though: if you are writing something like this, there is something wrong! :)

NOTE	CSS selector like descendants, siblings, ids, wildcards and pseudos (other than `:not`) are not supported.
NOTE	Don't start your attribute selectors with `bind-`, `on-` or `var-`: they have other meanings for the parser, as they are part of the canonical templating syntax.

Great, we know how to declare a directive. Let's make one that actually does something.

12.2.2. Inputs

Data binding is usually a big part of the job of creating a component or a directive. Every time you want a top component to pass data to one of its children, you will use a property binding.

To do this, we will define all the properties that accept data binding, using the `inputs` attribute of the `@Directive` decorator. This attribute accepts an array of strings, each one of the form `property: binding`. `property` represents the directive instance property and `binding` is the DOM property that will contain the expression.

For example, this directive is binding the DOM property `logText` to the directive instance property `text`:

```
@Directive({
  selector: '[loggable]',
  inputs: ['text: logText']
})
export class SimpleTextDirective {
}
```

If the property does not exist in your directive, it is created for you. Then, every time the input changes, the property is updated automatically.

```
<div loggable logText="Some text">Hello</div>
```

If you want to be notified when the property changes, you can add a setter to your directive. The setter will be called every time the `logText` property changes.

```
@Directive({
  selector: '[loggable]',
  inputs: ['text: logText']
})
export class SimpleTextDirectiveWithSetter {

  set text(value) {
    console.log(value);
  }
}
```

So if we use it:

```
<div loggable logText="Some text">Hello</div>
// our directive will log "Some text"
```

There is also another way we'll see in a few minutes.

Here the text is static, but of course it could easily be a dynamic value, with an interpolation:

```
<div loggable logText="{{expression}}">Hello</div>
// our directive will log the value of 'expression' in the component
```

or the square bracket syntax:

```
<div loggable [logText]="expression">Hello</div>
// our directive will log the value of 'expression' in the component
```

That's one of the greatest features of the new template syntax: as a component developer, you don't care how your component is used, you just define which properties are bound (if you wrote some AngularJS 1.x, you know it was slightly different with all the '@' and '=' syntax).

You can also use pipes in your bindings:

```
<div loggable [logText]="expression | uppercase">Hello</div>
// our directive will log the value of 'expression' in the component in uppercase
```

If you want to bind a DOM property to an attribute of your directive that has the same name, you can simply write `property` instead of `property: binding`:

```
@Directive({
  selector: '[loggable]',
  inputs: ['logText']
})
export class SameNameInputDirective {

  set logText(value) {
    console.log(value);
  }
}
```

```
<div loggable logText="Hello">Hello</div>
// our directive will log "Hello"
```

There is another way to declare an input in your directive: with the `@Input` decorator. I like it very much, so a lot of examples will use it from now on, but feel free to use whatever you prefer.

The examples above can be rewritten as:

```
@Directive({
  selector: '[loggable]'
})
export class InputDecoratorDirective {

  @Input('logText') text: string;

  set text(value) {
    console.log(value);
  }
}
```

or, if the property and the binding have the same name:

```
@Directive({
  selector: '[loggable]'
})
export class SameNameInputDecoratorDirective {

  @Input() logText: string;

  set logText(value) {
    console.log(value);
  }
}
```

Inputs are great to pass data from a top element to a bottom element. For example, if you want to have a component displaying a list of ponies, it is very likely that you will have a top component containing the list, and another component to display a pony:

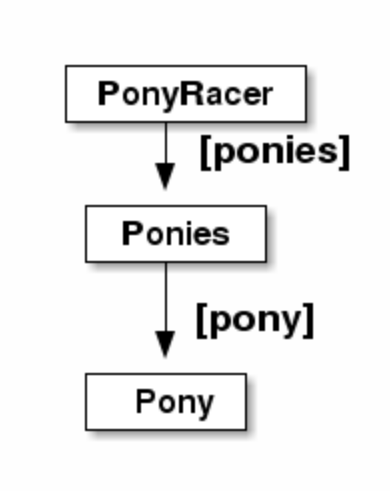
```
@Component({
  selector: 'pony-cmp',
  template: `<div>{{pony.name}}</div>`
})
export class PonyCmp {
  @Input() pony: Pony;
}

@Component({
  selector: 'ponies-cmp',
  template: `<div>
    <h2>Ponies</h2>
    // the pony is handed to PonyCmp via [pony]="pony"
    <pony-cmp *ngFor="#pony of ponies" [pony]="pony"></pony-cmp>
  </div>`,
  directives: [PonyCmp]
})
export class PoniesCmp {
  ponies: Array<Pony> = [
    {id: 1, name: 'Rainbow Dash'},
    {id: 2, name: 'Pinkie Pie'}
  ];
}
```

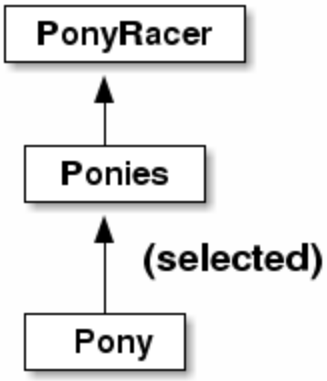
Ok, now what about passing data up? We can't use property to pass data from `PonyCmp` to `PoniesCmp`. But we can use events!

12.2.3. Outputs

Let's go back to our latest example, and say we want to be able to select a pony by clicking on it, and inform the parent component. For this, we will use a custom event.

This is important. In Angular 2, data flows into a component via properties, and flows out of a component via events.





You remember the previous chapter on reactive programming? Cool, that's going to be useful! Custom events are emitted using an `EventEmitter`, and must be declared in the decorator, using the `outputs` attribute. Like the `inputs` attribute, it accepts an array with the list of events you want your directive/component to emit.

Let's say we want to emit an event called `ponySelected`. We have three things to do:

- declare the output in the decorator
- create an `EventEmitter`
- emit an event when the pony is selected

```
@Component({
  selector: 'pony-cmp',
  inputs: ['pony'],
  // we declare the custom event as an output
  outputs: ['ponySelected'],
  // the method `selectPony()` will be called on click
  template: `<div (click)="selectPony()">{{pony.name}}</div>`
})
export class SelectablePonyCmp {
  pony: Pony;
  // the EventEmitter is used to emit the event
  ponySelected: EventEmitter<Pony> = new EventEmitter<Pony>();

  /**
   * Selects a pony when the component is clicked.
   * Emits a custom event.
   */
  selectPony() {
    this.ponySelected.emit(this.pony);
  }
}
```

To use it in the template:

```
<pony-cmp [pony]="pony" (ponySelected)="betOnPony($event)"></pony-cmp>
```

In the above example, every time the user clicks on the pony name, it emits an event `ponySelected`, with the pony as a value (the parameter of the `emit()` method). The parent component is listening to this event, as you can see in the template, and will call its `betOnPony` method with the value of the event `$event`. `$event` is the syntax you have to use to access the event emitted: here, it will be the emitted pony.

The component must then have a method `betOnPony()`, which will be called with the selected pony:

```
betOnPony(pony) {
  // do something with the pony
}
```

If you wish, you can specify an event name different than the event emitter name, with the syntax `emitter: event`:

```
@Component({
  selector: 'pony-cmp',
  inputs: ['pony'],
  // the emitter is called `emitter`
  // and the event `ponySelected`
  outputs: ['emitter: ponySelected'],
  template: `<div (click)="selectPony()">{{pony.name}}</div>`
})
export class OtherSelectablePonyCmp {
  pony: Pony;
  emitter: EventEmitter<Pony> = new EventEmitter<Pony>();

  selectPony() {
    this.emitter.emit(this.pony);
  }
}
```

As for the `inputs`, it is possible to use a decorator to declare an event. Again, I like to use them, but make your own choice.

```
@Component({
  selector: 'pony-cmp',
  template: `<div (click)="selectPony()">{{pony.name}}</div>`
})
export class SelectablePonyWithDecoratorCmp {
  @Input() pony: Pony;
  @Output() ponySelected: EventEmitter<Pony> = new EventEmitter<Pony>();

  selectPony() {
    this.ponySelected.emit(this.pony);
  }
}
```

12.2.4. Lifecycle

You may want your directive to react on a specific moment of its life.

This is quite advanced stuff, and you won't need it every day, so I'll go fast.

One thing is really important to understand though, and you'll save quite some time if you do: **the inputs of a component are not evaluated yet in its constructor**.

That means that the following component will not work:

```
@Directive({
  selector: '[undefinedInputs]'
})
export class UndefinedInputs {

  @Input() pony: string;

  constructor() {
    console.log(`inputs are ${this.pony}`);
    // will log "inputs are undefined", always
  }

}
```

If you want to access the value of an input, to load additional data from the server for example, you have to use a lifecycle phase.

Several phases are available, and have their own specificity:

- `ngOnChanges` will be the first to be called when the value of a bound property changes. It will receive a `changes` map, containing the current and previous values of the binding, wrapped in a `SimpleChange`. It will not be called if there is no change.
- `ngOnInit` will be called only once after the first change (whereas `ngOnChanges` is called on every changes). It makes this phase perfect for initialization work, as the name suggests.
- `ngOnDestroy` is called when the component is removed. Really useful to do some cleanup.

Other phases are available, but are for more advanced use cases:

- `ngDoCheck` is slightly different. If present it will be called at each change detection cycle, overriding the default change detection algorithm, which looks for difference between every bound property value. That means that if at least one input has changed, by default the component is considered changed by the framework, and its children will be checked and rendered. But you can override it if you know that some inputs have no effect even if they have changed. That can be useful if you want to accelerate the change detection by just checking the bare minimum and not using the default algorithm, but usually you will not use this.
- `ngAfterContentInit` is called when all the bindings of the component have been checked for the first time.
- `ngAfterContentChecked` is called when all the bindings of the component have been checked, even if they haven't changed.
- `ngAfterViewInit` is called when all the bindings of the children directives have been checked for the first time.
- `ngAfterViewChecked` is called when all the bindings of the children directives have been checked, even if they haven't changed. It can be useful if your component is waiting for something coming

from its child components. Like `ngAfterViewInit`, it only makes sense if we are in a component (a directive has no view).

Our previous sample will work better using `ngOnInit`. Angular 2 invokes the method `ngOnInit()` if it's present, so you just have to implement it in your directive. If you are using TypeScript for your app, you can leverage the available interface `OnInit` that forces you to implement the method:

```
@Directive({
  selector: '[initDirective]'
})
export class OnInitDirective implements OnInit {

  @Input() pony: string;

  ngOnInit() {
    console.log(`inputs are ${this.pony}`);
    // inputs are not undefined \o/
  }

}
```

Now we have access to our inputs!

If you want to do something every time a property changes, use `ngOnChanges`:

```
@Directive({
  selector: '[changeDirective]'
})
export class OnChangesDirective implements OnChanges {

  @Input() pony: string;

  ngOnChanges(changes: {[key: string]: SimpleChange}) {
    console.log(`changed from ${changes.pony.previousValue} to ${changes.pony
.currentValue}`);
  }

}
```

The `changes` parameter is a map, with the binding names as keys, and a `SimpleChange` object with two attributes (the previous and the current value) as value.

You can also use a setter if you want to react only on the change of one of your bindings. The following example will produce the same output as the previous one.

```
@Directive({
  selector: '[setterDirective]',
  inputs: ['pony']
})
export class SetterDirective {

  private _pony: string;

  set pony(newPony) {
    console.log(`changed from ${this._pony} to ${newPony}`);
    this._pony = newPony;
  }

}
```

`ngOnChanges` is more useful if you need to watch several bindings at the same time. It will only be invoked if at least one binding has changed and will contain only the properties that have changed.

The `ngOnDestroy` phase is perfect to clean the component - for example, to cancel background tasks. Here, the `OnDestroyDirective` is logging "hello" every second when it is created. When the component is removed from the page, you want to stop the `setInterval` to avoid a memory leak:

```
@Directive({
  selector: '[destroyDirective]'
})
export class OnDestroyDirective implements OnDestroy {

  sayHello: number;

  constructor() {
    this.sayHello = window.setInterval(() => console.log('hello'), 1000);
  }

  ngOnDestroy() {
    window.clearInterval(this.sayHello);
  }

}
```

If you don't do this, you will have the thread logging "hello" until the end or the crash...

12.2.5. Providers

We already talked about providers in the Dependency Injection chapter . This attribute allows to declare services that will be injectable in the current directive and its children.

```
@Directive({
  selector: '[providersDirective]',
  providers: [PoniesService]
})
export class ProvidersDirective {

  constructor(poniesService: PoniesService) {
    let ponies = poniesService.list();
    console.log(`ponies are: ${ponies}`);
  }
}
```

12.3. Components

A component is not really different from a directive: it just has two more optional attributes and **must** have an associated view. It does not bring a lot of new attributes compared to the directive.

12.3.1. View providers

We saw that you can specify injectables using `providers`. `viewProviders` is slightly similar but the providers will only be available for the current component, not for its children.

12.3.2. Template / Template URL

The main feature of a `@Component` is to have a template, whereas a directive does not have one. You can either declare your template inline, using `template` or use a URL to put it in a separate file with `templateURL` (but you can't do both at the same time).

As a rule of thumb, if your template is small (1-2 lines), it's perfectly fine to keep it inline. When it starts to grow, move it to its own file to avoid cluttering your component.

You can use an absolute path for your URL, a relative one or even a complete HTTP URL.

When the component is loaded, Angular 2 resolves the URL and tries fetching the template. If it succeeds, the template is the Shadow Root of the component, and its expressions are evaluated.

If I have a big component, I usually put the template in a separate file of the same folder, and use a relative URL to load it.

```
@Component({
  selector: 'templated-pony',
  templateUrl: 'components/pony/templated_pony.html'
})
export class TemplatedPony {
  @Input() pony: any;
}
```

If you use a relative URL, the URL will be resolved using the base URL of your app. The URL can be cumbersome, because if your component is in a directory `components/pony`, your template URL will be `components/pony/pony.html`.

But you can do slightly better if you package your application using CommonJS modules, by using the property `moduleId`. Its value must be `module.id`, a value that CommonJS will set at runtime. Angular 2 can then use this value and build the correct relative URL. Your template URL can now look like:

```
@Component({
  selector: 'templated-pony',
  templateUrl: './templated_pony.html',
  moduleId: module.id
})
export class ModuleIdPony {
  @Input() pony: any;
}
```

And you can locate your template in the same directory as your component!

12.3.3. Styles / Styles URL

You can also specify the styles of your component. It is particularly useful if you plan to have really isolated components. You can specify this using `styles` or `styleUrls`.

As you can see below, the `styles` attribute takes an array of CSS rules as a string. You can imagine it grows pretty quickly, so using a separate file and `styleUrls` is a good idea. As the name of the latter suggests, you can specify an array of URLs.

```
@Component({
  selector: 'styled-pony',
  template: '<div class="pony">{{pony.name}}</div>',
  styles: ['.pony{ color: red; }']
})
export class StyledPony {
  @Input() pony: any;
}
```

12.3.4. Directives

You have to declare every directive and component you are using in your component template. If you don't, your component will not be picked up in the template, and you will waste a lot of time figuring out why.

The two most common mistakes are **forgetting about the `directives` attribute** and **using the wrong selector**. If you don't understand why nothing happens, look out for these!

12.3.5. Pipes

As we previously saw, using a custom pipe in a component requires to specify it in the `pipes` attribute. See the Pipes chapter for more info.

12.4. Making a component available everywhere

Having to systematically declare the components you want to use in a template is a painful point, as you must have noticed.

If you have a component you use in a lot of templates, there is a better solution than to include it manually in the `directives` attribute of each component using it.

The common framework directives (like `NgIf`, `NgFor`, the form directives, etc...) are included for us in the `PLATFORM_DIRECTIVES` and are loaded on start. That makes these directives available everywhere.

We can leverage this to add our custom directives and components in this collection, and remove the burden of having to manually include them later in each component using them.

Let's say that the `PonyCmp` is used very often (we are writing an app about ponies racing fiercely, after all). We can add the component to the `PLATFORM_DIRECTIVES` with:

```
bootstrap([
    provide(PLATFORM_DIRECTIVES, {useValue: PonyCmp, multi: true})
]);
```

And then simplify our component `RaceCmp` which was like this:

```
@Component({
  selector: 'race-cmp',
  templateUrl: 'race/race.html',
  directives: [PonyCmp]
})
export class RaceCmp {

  @Input() race: Race;
}
```

to:

```
@Component({
  selector: 'race-cmp',
  templateUrl: 'race/race.html'
})
export class RaceCmpSimplified {

  @Input() race: Race;
}
```

We have left a few things out for now, like the queries, change detection, exports, encapsulation options, etc... As they are more advanced options, you won't need them immediately; but don't worry, we'll see them soon in an advanced chapter!

Chapter 13. Services

Angular 2 contains the concept of services: classes you can inject in an other.

A few services are provided by the framework, some by the common modules, and others can be built by you. We will see the ones provided by the common modules in dedicated chapters; right now, let's have a look at the core ones, and discover how we can build ours.

13.1. Title service

The core framework provides very few services, and those you will use in your apps are even scarcer: actually there is just one for now :).

One question that pops up frequently is how can I change the title of my page? Easy! There is a `Title` service you can inject and it offers a getter and a setter method:

```
import {Component} from 'angular2/core';
import {Title} from 'angular2/platform/browser';

@Component({
  selector: 'ponyracer-app',
  viewProviders: [Title],
  template: `<h1>PonyRacer</h1>`
})
export class PonyRacerApp {

  constructor(title: Title) {
    title.setTitle('PonyRacer - Bet on ponies');
  }

}
```

The service will automatically create the `title` element in the `head` if needed and correctly set the value for you!

13.2. Making your own service

That's really simple. Just build a class, and you're done!

```
export class RacesService {

  list() {
    return [{name: 'London' }];
  }

}
```

Just like in AngularJS 1.x, a service is a singleton, so the same, unique instance of the class will be injected everywhere. It thus makes a service a great candidate to share state between several unrelated components!

If your service has some dependencies itself, then you need to add the `@Injectable()` decorator on it. Without this decorator, the framework won't do the dependency injection.

Our `RacesService` probably fetches the races from a REST API instead of returning the same list every time. To perform an HTTP request, the framework provides the `Http` service. Don't worry, we'll soon see how it works.

Our service has a dependency on `Http` to fetch the races, so we need to add a constructor with the `Http` service as an argument, and add the `@Injectable()` decorator on the class.

```
import {Injectable} from 'angular2/core';
import {Http} from 'angular2/http';

@Injectable()
export class RacesServiceWithHttp {

  constructor(private _http: Http) {
  }

  list() {
    return this._http.get('/api/races');
  }

}
```

That's very interesting, but we now need to take a look at how to test all that!

Chapter 14. Testing your app

14.1. The problem with troubleshooting is that trouble shoots back

I love automated testing. My professional life revolves around the test progress bar going green in my IDE, patting me in the back for doing my job properly. And I hope you do care about tests too, as they are the only safety net we have when we write code. Nothing is more tedious than manually testing code.

Angular 2 does a great job to let us easily write tests. So did AngularJS 1.x, and that's partly why I loved using it. As in AngularJS 1.x, we can write two types of tests:

- unit tests
- end-to-end tests

The first ones are there to assert a small unit of code (a component, a service, a pipe…) works correctly in isolation, i.e. without considering its dependencies. Writing such a unit test requires to execute each of the component/service/pipe methods, and check that the outputs are what we expected regarding the inputs we fed it. We can also check that the dependencies used by this unit are correctly called, for example we can check that a service will do the correct HTTP request.

We can also write end-to-end tests. Their purpose is to emulate a real user interacting with your app, by starting a real instance and then driving the browser to enter values in inputs, click on buttons, etc… We'll then check that the rendered page is in the state we expect, that the URL is correct, whatever you can think of.

We're going to cover all this, but let's begin with the unit test part.

14.2. Unit test

As we saw earlier, unit tests are there to check a small unit of code in isolation. These tests can only assert a small part of your app works as intended, but they have several advantages:

- they are really fast, you can run several hundreds in a few seconds.
- they are very efficient to test (nearly) all your code, especially the tricky cases, that can be hard to manually test in the real app.

One of the core concept of unit test is the isolation concept: we don't want our test to be biased by its dependencies. So we usually use "mock" objects as dependencies. These are fake objects that we create just for testing purpose.

To do this, we are going to rely on a few tools. First we need a library to write tests. One of the most

popular (if not the most popular) is Jasmine. And for now it is the only one supported, as far as I know, so we don't really have a choice, we are going to use it :)

14.2.1. Jasmine and Karma

Jasmine gives us a few methods to declare our tests:

- `describe()` declares a test suite (a group of tests)
- `it()` declares a test
- `expect()` declares an assertion

A basic JavaScript test using Jasmine looks like:

```
class Pony {
  constructor(public name: string, public speed: number) {
  }

  isFasterThan(speed) {
    return this.speed > speed;
  }
}

describe('My first test suite', () => {
  it('should construct a Pony', () => {
    let pony: Pony = new Pony('Rainbow Dash', 10);
    expect(pony.name).toBe('Rainbow Dash');
    expect(pony.speed).not.toBe(1);
    expect(pony.isFasterThan(8)).toBe(true);
  });
});
```

The `expect()` call can be chained with a lot of methods like `toBe()`, `toBeLessThan()`, `toBeUndefined()`, etc... Every method can be negated with the `not` attribute of the object returned by `expect()`.

The test file is a separate file from the code you want to test, usually with an extension like `.spec.ts`. The test for a Pony class written in a `pony.ts` file will likely be in a file named `pony.spec.ts`. You can either put your test right next to the file you're testing, or in a dedicated directory with all your tests. I tend to put the code and test in the same directory, but both approaches are perfectly valid: pick your team.

NOTE	One cool trick is that if you use `fdescribe()` instead of `describe()` then only this test suite will run (f stands for focus). Same thing if you want to run only one test: use `fit()` instead of `it()`. If you want to exclude a test, use `xit()`, or `xdescribe()` for a suite.

You can also use the `beforeEach()` method to set up a context before each test: the **fixture**. If I have

several tests on the same pony, it makes sense to use `beforeEach()` to initialize the pony, instead of copy/pasting the same thing in every tests.

```
describe('Pony', () => {
  let pony: Pony;

  beforeEach(() => {
    pony = new Pony('Rainbow Dash', 10);
  });

  it('should have a name', () => {
    expect(pony.name).toBe('Rainbow Dash');
  });

  it('should have a speed', () => {
    expect(pony.speed).not.toBe(1);
    expect(pony.speed).toBeGreaterThan(9);
  });
});
```

There is also an `afterEach` method, but I basically never use it...

One last trick: Jasmine lets us create fake objects (mocks or spies, as you want), or even spy on a method of a real object. We can then do some assertions on these methods, like with `toHaveBeenCalled()` that checks if the method has been called, or with `toHaveBeenCalledWith()` that checks the exact parameters of the call to the spied method. You can also check how many times the method has been called, or check if it has ever been called, etc...

```
describe('My first test suite with spyOn', () => {
  let pony: Pony;

  beforeEach(() => {
    pony = new Pony('Rainbow Dash', 10);
    // define a spied method
    spyOn(pony, 'isFasterThan').and.returnValue(true);
  });

  it('should test if the Pony is fast', () => {
    let runPonyRun = pony.isFasterThan(60);
    expect(runPonyRun).toBe(true); // as the spied method always returns
    expect(pony.isFasterThan).toHaveBeenCalled();
    expect(pony.isFasterThan).toHaveBeenCalledWith(60);
  });
});
```

When you write unit tests, keep in mind that they should be small and readable. And don't forget to make them fail at first, to be sure you're testing the right thing.

The next step is to run our tests. For this, the Angular team has developed Karma, whose sole purpose is to run the tests in one or several browsers. It can also watch your files to re-run the tests on every save. As running the tests is really fast, it's actually really nice to do this and have (almost) instant feedback on your code.

I won't dive into the details on how to setup Karma, but it's a very interesting project with a lot of plugins you can use, to make it work with your favorite tools, to have a coverage report, etc... If you're writing your code in TypeScript like me, the strategy you can adopt is to let the TypeScript compiler watch your code and tests, produce the compiled files in a separate output directory, and have Karma watch this directory.

So we now know how to write a unit test in JavaScript. Let's add Angular 2 to the mix.

14.2.2. Using dependency injection

Let's say I have an Angular 2 application with a simple service like `RaceService`, containing a method returning a hard-coded races list.

```
export class RaceService {
  list() {
    let race1 = new Race('London');
    let race2 = new Race('Lyon');
    return [race1, race2];
  }
}
```

Let's write a test for this. To do this, we'll rely on the `angular2/testing` module

```
import {describe, it, expect} from 'angular2/testing';

describe('RaceService', () => {
  it('should return races when list() is called', () => {
    let raceService: RaceService = new RaceService();
    expect(raceService.list().length).toBe(2);
  });
});
```

That works great. But we can also rely on the dependency injection offered by Angular to grab the RaceService and inject it in our test. It's especially useful if our `RaceService` has some dependencies itself: instead of having to instantiate these dependencies ourselves, we could just rely on the injector to do it for us by saying: "hey, we want the RaceService, go figure out what you need to create it and give it to me".

To use the dependency injection system in our test, the framework has a utility method called `inject`.

This method can be used to wrap your test function (the second parameter of our `it` function), and inject specific dependencies inside this function.

Let's go back to our example, using `inject` this time:

```
import {describe, it, expect, inject} from 'angular2/testing';

describe('RaceService', () => {
  it('should return races when list() is called', inject([RaceService], (raceService) =>
{
    expect(raceService.list().length).toBe(2);
  }));
});
```

As you can see, `inject` takes two parameters:

- an array containing the classes we want to inject as dependencies
- a test function, whose parameters match the dependencies we declared

That won't work exactly like this, because we also need to tell the test what is available for injection, as we do in the `bootstrap` method when we start the app.

The `beforeEachProviders` function allows to declare what can be injected in the test. Try to inject only what's necessary in your test, to make them as loosely coupled to the rest of the app as possible. The method takes a function as its unique parameter, and this function should return an array of dependencies that will become available to injection.

```
import {describe, it, expect, inject, beforeEachProviders} from 'angular2/testing';

describe('RaceService', () => {

  beforeEachProviders(() => [RaceService]);

  it('should return races when list() is called', inject([RaceService], (raceService) =>
{
    expect(raceService.list().length).toBe(2);
  }));
});
```

Now that's working, great! Note that if our `RaceService` had some dependencies itself, we would have to declare them in the `beforeEachProviders` method, to make them available for injection.

As we did in the simple Jasmine example, we can maybe move the `RaceService` initialization in a

`beforeEach` method. We can also use `inject` in a `beforeEach`, so let's do it:

```
import {describe, it, expect, inject, beforeEachProviders, beforeEach} from
'angular2/testing';

describe('RaceService', () => {
  let service: RaceService;

  beforeEachProviders(() => [RaceService]);

  beforeEach(inject([RaceService], (raceService) => {
    service = raceService;
  }));

  it('should return races when list() is called', () => {
    expect(service.list().length).toBe(2);
  });
});
```

We moved the `inject` logic in a `beforeEach` and now our test is pretty clean. Be careful to always call `beforeEachProviders` (which sets up the injector), before actually using the injector with `inject` or your test will fail.

Of course, a real `RaceService` will not have a hard-coded list of races, and there is a big chance that the response will be an asynchronous one. Let's say that the `list` returns a promise. What does that change in our test? Well, now we have to set up the `expect` in the `then` callback:

```
import {describe, it, expect, injectAsync, beforeEachProviders} from 'angular2/testing';

describe('RaceService', () => {

  beforeEachProviders(() => [RaceService]);

  it('should return a promise of 2 races', injectAsync([RaceService], service => {
    return service.list().then(races => {
      expect(races.length).toBe(2);
    });
  }));
});
```

You may be thinking that this will not work, as the test will end before the promise is resolved, and our expectation will never run.

But here we use `injectAsync()` instead of the `inject()` method. And this method is really smart: it keeps track of the asynchronous calls made in the test and waits for them to resolve.

Angular 2 uses a new concept called **zones**. These **zones** are execution contexts, and, to simplify, they keep track of all the stuff going on within them (timeouts, event listeners, callbacks...). They also provide hooks that can be called when we enter or leave the zone. An Angular 2 application runs in a zone, and that's how the framework knows it has to refresh the DOM when an async action is done.

This concept is also used in the tests if your test uses `injectAsync()`: each test runs in a zone, so the framework knows when all the asynchronous actions are done, and won't complete until then.

So our asynchronous expectation will be executed. Great!

There is another way to deal with async tests in Angular 2, using `fakeAsync()` and `tick()` but that's for a more advanced chapter.

14.3. Fake dependencies

Being able to declare the dependencies in the `beforeEachProviders()` method has another use. We can without too much trouble declare a fake service as a dependency instead of a real one.

For the sake of the example, let's say that my `RaceService` uses the local storage to store the races, with a key `races`. Your colleagues have developed a service called `LocalStorageService` that deals with the JSON serialization, etc... that our `RaceService` uses. The `list()` method looks like:

```
@Injectable()
export class RaceService {
  constructor(private localStorage: LocalStorageService) {
  }

  list() {
    return this.localStorage.get('races');
  }
}
```

Now, we don't really want to test the `LocalStorageService` service, we just want to test our `RaceService`. That can easily be done by leveraging the dependency injection system to give a fake `LocalStorageService`:

```
class FakeLocalStorage {
  get(key) {
    return [new Race('Lyon'), new Race('London')];
  }
}
```

to `RaceService` in our test, using the `provide()` method:

```
import {describe, it, expect, beforeEachProviders, inject} from 'angular2/testing';
import {provide} from 'angular2/core';

describe('RaceService', () => {

  beforeEachProviders(() => [
    provide(LocalStorageService, {useClass: FakeLocalStorage}),
    RaceService
  ]);

  it('should return 2 races from localStorage', inject([RaceService], service => {
    let races = service.list();
    expect(races.length).toBe(2);
  }));
});
```

Great! But I'm not completely satisfied with this test. Creating a fake service by hand is tedious, and Jasmine can help us spy on the service and replace its implementation by a fake one. It also allows to verify that the `get()` method has been called with the proper key 'races'.

```
import {describe, it, expect, beforeEachProviders, inject} from 'angular2/testing';
import {provide} from 'angular2/core';

describe('RaceService', () => {

  beforeEachProviders(() => [
    LocalStorageService,
    RaceService
  ]);

  it('should return 2 races from localStorage', inject([RaceService, LocalStorageService
], (service, localStorage) => {
    spyOn(localStorage, 'get').and.returnValue([new Race('Lyon'), new Race('London')]);

    let races = service.list();

    expect(races.length).toBe(2);
    expect(localStorage.get).toHaveBeenCalledWith('races');
  }));
});
```

14.4. Testing components

The next step after testing a simple service is to test a component. A component test is slightly different

because we have to create the component. We can't rely on the dependency injection system to give us an instance of the component to test (you may have noticed by now that components are not injectable in other components :)).

Let's start by writing a component to test. Why not our `PonyCmp` component? It takes a pony as an input and emits an event `ponyClicked` when the component is clicked.

```
@Component({
  selector: 'pony-cmp',
  template: `<img [src]="'/images/pony-' + pony.color.toLowerCase() + '.png'" (click)=
"clickOnPony()">`
})
export class PonyCmp {

  @Input() pony: Pony;
  @Output() ponyClicked: EventEmitter<void> = new EventEmitter<void>();

  clickOnPony() {
    this.ponyClicked.emit();
  }

}
```

It comes with a fairly simple template: an image with a dynamic source depending on the pony color, and a click handler.

To test such a component, you first need to create an instance. To do this, the framework gives us `TestComponentBuilder`. This class comes with a utility method, named `createAsync`, to create a component. The method returns a promise of a `ComponentFixture`, a representation of our component.

We can then chain the creation method with our test.

```
import {
  describe,
  it,
  expect,
  inject,
  injectAsync,
  beforeEach,
  TestComponentBuilder
} from 'angular2/testing';

import {PonyCmp} from './pony_cmp';

describe('PonyCmp', () => {

  let tcb: TestComponentBuilder;

  beforeEach(inject([TestComponentBuilder], tcBuilder => {
    tcb = tcBuilder;
  }));

  it('should have an image', injectAsync([], () => {
    return tcb.createAsync(PonyCmp)
      .then(fixture => {
        // given a component instance with a pony input initialized
        let ponyCmp = fixture.componentInstance;
        ponyCmp.pony = {name: 'Rainbow Dash', color: 'BLUE'};

        // when we trigger the change detection
        fixture.detectChanges();

        // then we should have an image with the correct source attribute
        // depending of the pony color
        let element = fixture.nativeElement;
        expect(element.querySelector('img').getAttribute('src')).toBe('/images/pony-
blue.png');
      });
  }));
});
```

Here, we follow the "Given/When/Then" pattern to write the unit test. You'll find a whole literature on the subject, but it boils down to:

- a "Given" phase, where we setup the test context. We get the component instance created and add a pony. It emulates an input that would come from a parent component in the real app.
- a "When" phase, where we manually trigger the change detection, using the `detectChanges()` method. In a test, the change detection is our responsibility: it's not automatic as it is in an app.

- and a "Then" phase, containing the expectations. We can get the native element and query the DOM as you would do with the browser (using `querySelector()` for example). Here we test if the image source is the correct one.

A few matchers are added by the framework. Some are very handy:

- `toHaveText()` allows to check that a DOM element contains a given text. For example, you could check that the div text contains the pony name.
- `toHaveCssClass()` allows to check that a DOM element… has a given CSS class.

Some others are less often used (in my experience at least): `toBeInstanceOf()`, `toThrowErrorWith(message)`, etc…

You can find the list of all the additional matchers in the official documentation.

Let's have a look at another component:

```
@Component({
  selector: 'race-cmp',
  template: `<div>
    <h1>{{race.name}}</h1>
    <pony-cmp *ngFor="#pony of race.ponies" [pony]="pony"></pony-cmp>
  </div>`,
  directives: [PonyCmp]
})
export class RaceCmp {

  @Input() race: any;

}
```

and its test:

```
describe('RaceCmp', () => {

  let fixture: ComponentFixture;

  beforeEach(injectAsync([TestComponentBuilder], tcb =>
    tcb.createAsync(RaceCmp)
      .then(f => fixture = f)
  ));

  it('should have a name and a list of ponies', () => {
    // given a component instance with a race input initialized
    let raceCmp = fixture.componentInstance;
    raceCmp.race = {name: 'London', ponies: [{name: 'Rainbow Dash', color: 'BLUE'}]};

    // when we trigger the change detection
    fixture.detectChanges();

    // then we should have a name with the race name
    let element = fixture.nativeElement;
    expect(element.querySelector('h1')).toHaveText('London');

    // and a list of ponies
    let ponies = fixture.debugElement.queryAll(By.directive(PonyCmp));
    expect(ponies.length).toBe(1);
    // we can check if the pony is correctly initialized
    let rainbowDash = ponies[0].componentInstance.pony;
    expect(rainbowDash.name).toBe('Rainbow Dash');
  });
});
```

Here we query all the directives of type `PonyCmp` and test if the first pony is correctly initialized.

You can get the components inside your component with `componentViewChildren` or query them with `query()` and `queryAll()`. These methods take a predicate as argument that can be either `By.css` or `By.directive`. That's what we do to get the ponies displayed, as they are instances of `PonyCmp`. Keep in mind that this is different from a DOM query using `querySelector()`: it will only find the elements handled by Angular, and will return a `ComponentFixture`, not a DOM element (so you'll have access to the `componentInstance` of the result, for example).

14.5. Testing with fake templates, directives...

When creating a component, we sometimes want to give it a less complex template for the test.

To do this, the `TestComponentBuilder` gives an `overrideTemplate()` method, to call before the `createAsync()` one:

```
describe('RaceCmp', () => {

  let fixture: ComponentFixture;

  beforeEach(injectAsync([TestComponentBuilder], tcb => {
    // let's replace the template of the race by simpler one
    return tcb.overrideTemplate(RaceCmp, '<h2>{{race.name}}</h2>')
      .createAsync(RaceCmp)
      .then(f => fixture = f);
  }));

  it('should have a name', () => {
    // given a component instance with a race input initialized
    let raceCmp = fixture.componentInstance;
    raceCmp.race = {name: 'London'};

    // when we trigger the change detection
    fixture.detectChanges();

    // then we should have a name
    let element = fixture.nativeElement;
    expect(element.querySelector('h2')).toHaveText('London');
  });
});
```

As you can see, the method takes two arguments:

- the component you want to override
- the template

That means you can modify the template of the component you are testing, or one of its children (to replace a component with a big template by a dumb one).

`overrideTemplate()` is not the only method available, you can also use:

- `overrideProviders()` and `overrideViewProviders()` to replace the dependencies of a component
- `overrideDirective()` to replace a directive used in the template of a component
- or even `overrideView()` where you can specify another pipe, style, etc...

Now you're ready to test your app!

14.6. End-to-end tests (e2e)

End-to-end tests are the other type of tests we can run. An end-to-end test consists in really launching your app in a browser and emulating a user interacting with it (clicking on buttons, filling forms, etc...

). They have the advantage of really testing the application in a whole, but:

- they are slower (several seconds per test)
- it's hard to test the edge cases.

As you may guess, you don't have to choose between unit tests and e2e tests: you will combine both to have a great coverage and some warranties that your complete application runs as intended.

E2e tests rely on a tool called Protractor. It's identical to the tool we used in AngularJS 1.x for the same purpose. And the great news is that it works both with AngularJS 1.x and Angular 2!

You will write your test suite using Jasmine like in the unit tests, but you will use the Protractor API to interact with your app.

A simple test would look like this:

```
describe('Home', () => {

  it('should display title, tagline and logo', () => {
    browser.get('/');
    expect(element.all(by.css('img')).count()).toEqual(1);
    expect($('h1').getText()).toContain('PonyRacer');
    expect($('small').getText()).toBe('Always a pleasure to bet on ponies');
  });

});
```

Protractor gives us a `browser` object, with a few utility methods like `get()` to go to a page. Then you have `element.all()` to select all the elements matching a predicate. This predicate often relies on `by` and its various methods (`by.css()` to do a CSS query, `by.id()` to retrieve an element by id, etc...). `element.all()` will return a promise, with a special method `count()` used in the test above.

`$('h1')` is a shortcut, equivalent of writing `element(by.css('h1'))`. It will fetch the first element matching the CSS query. You can use several methods on the promise returned by `$()`, like `getText()` and `getAttribute()` to retrieve information, or methods like `click()` and `sendKeys()` to act on this element.

These tests can be quite long to write and debug (much more than unit tests), but they are really useful. You can do all sorts of great things with them, like testing several browsers, do a screenshot every time a test fails, etc...

With unit tests and e2e tests, you have the keys to build a robust and maintainable application!

Chapter 15. Forms

15.1. Forms, dear forms

Forms have always been extra polished in Angular. That's one of the features that was the most demoed in 1.x, and, as pretty much every app has forms, that won the hearts of a lot of developers.

Forms are hard: you have to validate the inputs of your user, display errors, you can have fields required or not, or depending on another field, you want to react on some field changes, etc... We also need to test these forms, and that was impossible to achieve with a unit-test in AngularJS 1.x. It was only feasible with an end-to-end test, which can be slow.

In Angular 2, the same care has been applied to forms, and the framework gives us a nice way to write our forms. In fact, it gives us several ways!

Let's go through the same use case twice, using each way, and see the differences.

We are going to write a simple form, to be able to register new users in our awesome PonyRacer app. We need a base component for each use case, let's begin with this:

```
import {Component} from 'angular2/core';

@Component({
  selector: 'register-form-cmp',
  template: `
  <h2>Sign up</h2>
  <form></form>
  `
})
export class RegisterFormCmp {

}
```

Nothing fancy: a component with a simple template containing a form. In the next minutes, we will build a form allowing to register a user with a username and a password.

We will use a bunch of directives in our form. For example, the `NgForm` directive that transforms the `form` element into its powerful Angular 2 version - think of it as the difference between Bruce Wayne and Batman. These directives are already included in the `PLATFORM_DIRECTIVES`, so we don't need to import them.

```
import {Component} from 'angular2/core';

@Component({
  selector: 'register-form-cmp',
  template: `
  <h2>Sign up</h2>
  <form></form>
  `
})
export class RegisterFormCmp {

}
```

And add the submit button:

```
import {Component} from 'angular2/core';

@Component({
  selector: 'register-form-cmp',
  template: `
  <h2>Sign up</h2>
  <form (ngSubmit)="register()">
    <button type="submit">Register</button>
  </form>
  `
})
export class RegisterFormCmp {
  register() {
    // we will have to handle the submission
  }
}
```

I have added a button, and defined an event handler for `ngSubmit` on the `form` tag. The `ngSubmit` is emitted by the `form` directive when `submit` is triggered. It calls the `register()` method of our controller, which will be implemented later.

Last thing: our template will quickly grow, so let's extract it to a dedicated file, using `templateUrl`:

```
import {Component} from 'angular2/core';

@Component({
  selector: 'register-form-cmp',
  templateUrl: './register_form_cmp.html'
})
export class RegisterFormCmp {
  register() {
    // we will have to handle the submission
  }
}
```

The template file contains:

```
<h2>Sign up</h2>
<form (ngSubmit)="register()">
  <button type="submit">Register</button>
</form>
```

Ready? Let's go!

15.2. Model-driven

In AngularJS 1.x you had to build your forms mostly in your templates. Angular 2 introduces an imperative way, which allows to construct the form programmatically rather than through a template. In AngularJS 1.x, you did not have this choice, but it was done behind the scenes anyway by the framework.

Now we can handle forms directly in our code. It has a great advantage over the template-driven forms in that it is far more easy to test: it's just code, with less templating involved.

To build a form in our component code, we need some abstractions. A form field, like an input or a select, is represented by a `Control` in Angular 2. It is the smallest part of a form, and it encapsulates the state of the field and its value.

A `Control` has several attributes:

- `valid`: if the field is valid, regarding the requirements and validations applied on it.
- `errors`: an object containing the field errors
- `dirty`: false until the user has modified its value.
- `pristine`: the opposite of dirty.
- `touched`: false until the user has entered it.

- `untouched`: the opposite of `touched`.
- `value`: the value of the field.
- `status`: either `VALID` or `INVALID`
- `valueChanges`: an Observable emitting every time there is a change on the field

It also offers some methods like `hasError()` to check if the control has a specific error.

So you can do something like this:

```
let password = new Control();
console.log(password.dirty); // false until the user enters a value
console.log(password.value); // null until the user enters a value
console.log(password.hasError('required')); // false
```

Note that you can pass an argument to the constructor, and that this argument will be the value.

```
let password = new Control('Cédric');
console.log(password.value); // logs "Cédric"
```

These controls can be grouped in a `ControlGroup` to represent a part of the form and have dedicated validation rules. The form itself is a group.

A `ControlGroup` has the same properties than a `Control`, with a few differences:

- `valid`: if all fields are valid, then the group is valid.
- `errors`: an object containing the group errors or `null` if the group is valid. Each error is a key, whose value is an array containing every control affected by this error.
- `dirty`: false until one control gets dirty too.
- `pristine`: the opposite of dirty.
- `touched`: false until one control gets touched too.
- `untouched`: the opposite of `touched`.
- `value`: the value of the group. To be more accurate, it's an object with key/values representing the controls and their values.
- `status`: either `VALID` or `INVALID`
- `valueChanges`: an Observable emitting every time there is a change on the group

It offers the same methods as `Control` like `hasError()`. It also has a method `find()` to retrieve a control in the group.

You can create one like this:

```
let form = new ControlGroup({
  username: new Control('Cédric'),
  password: new Control()
});
console.log(form.dirty); // logs false until the user enters a value
console.log(form.value); // logs Object {username: "Cédric", password: null}
console.log(form.find('username')); // logs the Control
```

With these basic elements we can build a form in our component. But instead of doing `new Control()` or `new ControlGroup()`, we will use a helper class, `FormBuilder`, that we can inject:

```
import {Component} from 'angular2/core';
import {FormBuilder} from 'angular2/common';

@Component({
  selector: 'register-form-cmp',
  viewProviders: [FormBuilder],
  templateUrl: './register_form_cmp.html'
})
export class RegisterFormCmp {

  constructor(fb: FormBuilder) {
    // we will have to build the form
  }

  register() {
    // we will have to handle the submission
  }
}
```

The `FormBuilder` is a helper class, with a handful of methods to create controls and control groups. Let's start simple, and create a small form with two controls, a username and a password.

```
import {Component} from 'angular2/core';
import {FormBuilder, ControlGroup} from 'angular2/common';

@Component({
  selector: 'register-form-cmp',
  viewProviders: [FormBuilder],
  templateUrl: './register_form_cmp.html'
})
export class RegisterFormCmp {
  userForm: ControlGroup;

  constructor(fb: FormBuilder) {
    this.userForm = fb.group({
      username: fb.control(''),
      password: fb.control('')
    });
  }

  register() {
    // we will have to handle the submission
  }
}
```

We created a `form` with two controls. You can see that each control is created using the helper method `control()`, with an empty string as parameter. It has the same effect as calling the `new Control('')` constructor and the string represents the initial value you want to display in your form. Here it is empty, so the inputs will be empty. But you can have a value here, of course, if you want to edit an existing entity for example. The helper method can also have other specific attributes, as we will see later.

We need to implement the `register` method. As we saw, the `ControlGroup` object has a `value` attribute, so we can simply log its content with:

```
import {Component} from 'angular2/core';
import {FormBuilder, ControlGroup} from 'angular2/common';

@Component({
  selector: 'register-form-cmp',
  viewProviders: [FormBuilder],
  templateUrl: './register_form_cmp.html'
})
export class RegisterFormCmp {
  userForm: ControlGroup;

  constructor(fb: FormBuilder) {
    this.userForm = fb.group({
      username: fb.control(''),
      password: fb.control('')
    });
  }

  register() {
    console.log(this.userForm.value);
  }
}
```

We now need to do some work in the template. The form needs to be bound to our `userForm` object, thanks to the `ngFormModel` directive. Each input field is bound to a control, thanks to the `ngControl` directive:

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [ngFormModel]="userForm">
  <div>>
    <label>Username</label><input ngControl="username">
  </div>
  <div>
    <label>Password</label><input type="password" ngControl="password">
  </div>
  <button type="submit">Register</button>
</form>
```

We want to bind our `userForm` object to `ngFormModel`, so we use the bracket notation `[ngFormModel]="userForm"`. Each input receives the `ngControl` directive with a string literal representing the control it is bound to. If you specify a name that does not exist, you will have an error.

And we're done: clicking on the submit button will log an object containing the username and the chosen password!

NOTE	If you have a single control in your form, and not a control group, you can use `ngFormControl` on your field instead of using `ngControl` and wrapping it in a `ngFormModel`.

If you need to, you can update the value of a `Control` from your component, using `updateValue()`:

```
import {Component} from 'angular2/core';
import {FormBuilder, ControlGroup, Control} from 'angular2/common';

@Component({
  selector: 'register-form-cmp',
  viewProviders: [FormBuilder],
  templateUrl: './register_form_cmp.html'
})
export class RegisterFormCmp {
  username: Control;
  password: Control;
  userForm: ControlGroup;

  constructor(fb: FormBuilder) {
    this.username = fb.control('');
    this.password = fb.control('');
    this.userForm = fb.group({
      username: this.username,
      password: this.password
    });
  }

  reset() {
    this.username.updateValue('');
    this.password.updateValue('');
  }

  register() {
    console.log(this.userForm.value);
  }
}
```

15.3. Template-driven

In the "template-driven" way, you write your forms pretty much like in AngularJS 1.x, with a lot of things in your template and not many in your component.

In its simplest form, you just add `ngControl` directives to your form template and that's that. The `NgControl` directive creates the `Control` for you, and the form automatically creates the `ControlGroup`.

```
<h2>Sign up</h2>
<form (ngSubmit)="register()">
  <div>
    <label>Username</label><input ngControl="username">
  </div>
  <div>
    <label>Password</label><input type="password" ngControl="password">
  </div>
  <button type="submit">Register</button>
</form>
```

Now of course we need to do something for the submission, and to get hold of the user. For this, we are going to use a local variable. Remember these from the Template chapter? Here, we are going to define a variable, `userForm`, referencing the form. We can do this because the `form` directive exports an object representing the form, with the same methods as the `ControlGroup`. We'll see the exporting part in more details when we study how to build advanced directives.

```
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input ngControl="username">
  </div>
  <div>
    <label>Password</label><input type="password" ngControl="password">
  </div>
  <button type="submit">Register</button>
</form>
```

Our `register` method is now called with the form value as argument:

```
import {Component} from 'angular2/core';

@Component({
  selector: 'register-form-cmp',
  template: `
  <h2>Sign up</h2>
  <form (ngSubmit)="register()">
    <button type="submit">Register</button>
  </form>
  `
})
export class RegisterFormCmp {
  register(user) {
    console.log(user);
  }
}
```

This is only one-way data-binding though. If you update the field, the model will be updated, but updating the model will not update your field value.

15.3.1. Two-way data-binding

If you have been using AngularJS 1.x, or even just read an article about it, you must have seen the famous example with an input and an expression displaying the input value, updated every time the user modified the input, and the field automatically updated when the model changed. The famous "Two-Way Data-Binding", something like:

```
<input type="text" ng-model="username">
<p>{{username}}</p>
```

As we saw, that's not the case by default in Angular 2: you would have to use `updateValue()` for example.

If you want a real two-way data-binding, you can use something better: `ngModel`.

You can use it in either a model-driven form or a template-driven one.

You start by defining a model of what will be filled in the form. We'll do this in a `User` class:

```
class User {
  username: string;
  password: string;
}
```

Our `RegisterFormCmp` should have a field `user` of type `User`:

```
import {Component} from 'angular2/core';

class User {
  username: string;
  password: string;
}

@Component({
  selector: 'register-form-cmp',
  templateUrl: './register_form_cmp.html'
})
export class RegisterFormCmp {
  user: User = new User();

  register() {
    console.log(this.user);
  }
}
```

As you can see this time, the `register()` method is now directly logging the `user` object.

We are ready to add the inputs of our form:

```
<h2>Sign up</h2>
<form (ngSubmit)="register()">
  <div>
    <label>Username</label><input>
  </div>
  <div>
    <label>Password</label><input type="password">
  </div>
  <button type="submit">Register</button>
</form>
```

Now we need to bind our inputs to the model we have defined. For this, there is the `ngModel` directive. `NgModel` is what you know as 'two-way data-binding' in AngularJS 1.x. It can be used on an input:

```
<h2>Sign up</h2>
<form (ngSubmit)="register()">
  <div>
    <label>Username</label><input [(ngModel)]="user.username">
  </div>
  <div>
    <label>Password</label><input type="password" [(ngModel)]="user.password">
  </div>
  <button type="submit">Register</button>
</form>
```

Wow! `[(ngModel)]`? What is this syntax? It's a syntactic sugar that has been introduced to express the same thing as:

```
<input [ngModel]="user.username" (ngModelChange)="user.username = $event">
```

The `NgModel` directive updates the related model `user.username` every time there is a change in the input, hence the `[ngModel]="user.username"` part. And it fires an event called `ngModel` every time the model is updated, where the event is the new value, hence the `(ngModel)="user.username = $event"` part.

Instead of writing the long form, we can use the new syntax `[()]`. Now, every time we type something in our input, the model and the view are updated. And if the model is updated in our component, our field will automatically display the correct value:

```
<h2>Sign up</h2>
<form (ngSubmit)="register()">
  <div>
    <label>Username</label><input [(ngModel)]="user.username">
    <small>{{ user.username }} is an awesome username!</small>
  </div>
  <div>
    <label>Password</label><input type="password" [(ngModel)]="user.password">
  </div>
  <button type="submit">Register</button>
</form>
```

If you try the example above, you will see that the two-way data-binding works. And so does our form: we can submit it, and the component will log our `user` object!

`NgModel` can be useful if you really need to have the two-way data-binding. If you don't, it does not bring much: you can easily build a form with the two previous methods. Feel free to use it or not.

15.4. Adding some validation

Validation is usually a big part of the form-building. Some fields are required, some depend on one another, some should be in a specific format, some should not have a value greater or lower than X, for example.

Let's start by adding basic validation rules: all our fields are required.

15.4.1. In a model-driven form

To specify that every field is required, we will use a `Validator`. A validator returns a map of errors or `null` if it detects no error.

A few validators are provided by the framework:

- `Validators.required` to ensure that a control is not empty
- `Validators.minLength(n)` to ensure that the value entered has at least *n* characters
- `Validators.maxLength(n)` to ensure that the value entered has at most *n* characters

Validators are composable, using `Validators.compose()`, and can be applied on a `Control` or on a `ControlGroup`. Here we want every field to be mandatory, so we can add the required validator to each control, and make sure that the username is 3 characters at least.

```
import {Component} from 'angular2/core';
import {FormBuilder, ControlGroup, Validators} from 'angular2/common';

@Component({
  selector: 'register-form-cmp',
  viewProviders: [FormBuilder],
  templateUrl: './register_form_cmp.html'
})
export class RegisterFormCmp {
  userForm: ControlGroup;

  constructor(fb: FormBuilder) {
    this.userForm = fb.group({
      username: fb.control('', Validators.compose([Validators.required, Validators
.minLength(3)])),
      password: fb.control('', Validators.required)
    });
  }

  register() {
    console.log(this.userForm.value);
  }
}
```

15.4.2. In a template-driven form

Adding a required field in a template-driven form is also really straightforward: you just have to add the `required` attribute to the inputs. `required` is a provided directive, and will automatically add the validator to this field. Same thing with `minlength` and `maxlength`.

Starting from the two-way data-binding example:

```
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input ngControl="username" required minlength="3">
  </div>
  <div>
    <label>Password</label><input type="password" ngControl="password" required>
  </div>
  <button type="submit">Register</button>
</form>
```

15.5. Errors and submission

Of course, our user should not be able to submit the form while there are still errors left, and these errors should be perfectly displayed.

If you try the examples, you will see that even if the fields are required, we can still submit our form. Maybe we can do something about that?

We know that we can easily disable a button using the `disabled` property, but we need to give it an expression reflecting the state of the current form.

15.5.1. Errors and submission in a model-driven form

For this, we are going to use the `ControlGroup` variable: `userForm`. This variable gives us a complete view of the form and field states and errors.

For example, we can disable the form submission if the form is not valid:

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [ngFormModel]="userForm">
  <div>
    <label>Username</label><input ngControl="username">
  </div>
  <div>
    <label>Password</label><input type="password" ngControl="password">
  </div>
  <button type="submit" [disabled]="!userForm.valid">Register</button>
</form>
```

As you can see on the last line, we just need to link `disabled` to the `valid` property of `userForm`.

Now we can only submit when all controls are valid. To help our user understand why the form can't be submitted, we should display error messages.

Still using the `userForm`, we can do:

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [ngFormModel]="userForm">
  <div>
    <label>Username</label><input ngControl="username">
    <div *ngIf="userForm.find('username').hasError('required')">Username is
required</div>
  </div>
  <div>
    <label>Password</label><input type="password" ngControl="password">
    <div *ngIf="userForm.find('password').hasError('required')">Password is
required</div>
  </div>
  <button type="submit" [disabled]="!userForm.valid">Register</button>
</form>
```

Cool! The errors are now displayed if the fields are empty, and they disappear when there is a value. But they are displayed right away when the form is shown. Maybe we can hide them while the user has not changed the value?

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [ngFormModel]="userForm">
  <div>
    <label>Username</label><input ngControl="username">
    <div *ngIf="userForm.find('username').dirty &&
userForm.find('username').hasError('required')">
      Username is required
    </div>
  </div>
  <div>
    <label>Password</label><input type="password" ngControl="password">
    <div *ngIf="userForm.find('password').dirty &&
userForm.find('password').hasError('required')">
      Password is required
    </div>
  </div>
  <button type="submit" [disabled]="!userForm.valid">Register</button>
</form>
```

It's a bit verbose, but you can create a reference for each control in your component:

```
@Component({
  selector: 'register-form-cmp',
  viewProviders: [FormBuilder],
  templateUrl: './register_md_form.html'
})
export class RegisterFormCmp {
  userForm: ControlGroup;
  username: Control;
  password: Control;

  constructor(fb: FormBuilder) {
    this.username = fb.control('', Validators.required);
    this.password = fb.control('', Validators.required);

    this.userForm = fb.group({
      username: this.username,
      password: this.password
    });
  }

  register() {
    console.log(this.userForm.value);
  }
}
```

And then use the references in your template:

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [ngFormModel]="userForm">
  <div>
    <label>Username</label><input ngControl="username">
    <div *ngIf="username.dirty && username.hasError('required')">Username is
required</div>
  </div>
  <div>
    <label>Password</label><input type="password" ngControl="password">
    <div *ngIf="password.dirty && password.hasError('required')">Password is
required</div>
  </div>
  <button type="submit" [disabled]="!userForm.valid">Register</button>
</form>
```

You could also abstract the error displaying in a custom directive, like `display-error`:

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [ngFormModel]="userForm">
  <div>
    <label>Username</label><input ngControl="username">
    <display-error control="username" error="required">Username is required</display-
error>
  </div>
  <div>
    <label>Password</label><input type="password" ngControl="password">
    <display-error control="password" error="required">Password is required</display-
error>
  </div>
  <button type="submit" [disabled]="!userForm.valid">Register</button>
</form>
```

Such a component would look like:

```
@Component({
  selector: 'display-error',
  template: `<div *ngIf="isDisplayed()"><ng-content></ng-content></div>`
})
export class DisplayErrorCmp implements OnInit {
  @Input('control') controlName: string;
  @Input() error: string;
  control: AbstractControl;

  // we inject the form model
  constructor(@Host() private formModel: NgFormModel) {
  }

  // we then find the control
  ngOnInit() {
    this.control = this.formModel.form.find(this.controlName);
  }

  // the div in the template will only be added if
  // the control is dirty and has the specified error
  isDisplayed() {
    return this.control.dirty && this.control.hasError(this.error);
  }
}
```

You should be able to understand what this advanced component does at this point. The only part we have not seen yet is the `@Host()` decorator. You can probably understand that it will look into the host element to find the `NgFormModel` directive and inject it. This mechanism will be explained in details

later, in the advanced chapter.

15.5.2. Errors and submission in a template-driven form

In a template-driven form, we don't have a variable referring to the `ControlGroup` but we can use a local variable to create one.

```
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input ngControl="username" required>
  </div>
  <div>
    <label>Password</label><input type="password" ngControl="password" required>
  </div>
  <button type="submit" [disabled]="!userForm.valid">Register</button>
</form>
```

Now we need to display the errors of each field.

Like the form directive, each control exports its `Control` object, so we can create a local variable to access the errors:

```
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input ngControl="username" required #username="ngForm">
    <div *ngIf="username.control?.dirty && username.control?.hasError('required')"
>Username is required</div>
  </div>
  <div>
    <label>Password</label><input type="password" ngControl="password" required #
password="ngForm">
    <div *ngIf="password.control?.dirty && password.control?.hasError('required')"
>Password is required</div>
  </div>
  <button type="submit" [disabled]="!userForm.valid">Register</button>
</form>
```

Yay!

15.6. Add some style

Whatever way you choose to create your forms, Angular 2 does another awesome job for us: it

automatically adds and removes CSS classes on each field (and on the form) to allow us to add some visual style.

For example, a field will have the class `ng-invalid` if one of its validators fails, or `ng-valid` if all the validators succeed. That means you can easily add some style, like a nice red border around the fields failing the validation:

```
<style>
  input.ng-invalid {
    border: 3px red solid;
  }
</style>
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input ngControl="username" required>
  </div>
  <div>
    <label>Password</label><input type="password" ngControl="password" required>
  </div>
  <button type="submit" [disabled]="!userForm.valid">Register</button>
</form>
```

Another useful CSS class is `ng-dirty` which will be present if the user has changed the value. Its opposite is `ng-pristine`, present if the user never changed the value. I usually display the red border only when the user has changed the value at least once:

```
<style>
  input.ng-invalid.ng-dirty {
    border: 3px red solid;
  }
</style>
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input ngControl="username" required>
  </div>
  <div>
    <label>Password</label><input type="password" ngControl="password" required>
  </div>
  <button type="submit" [disabled]="!userForm.valid">Register</button>
</form>
```

Finally, there is a last CSS class: `ng-touched`. It will be present if the user enters and leaves the field at least once (even if she/he did not changed the value). Its opposite is `ng-untouched`.

When you display a form for the first time, a field will usually have the CSS classes `ng-pristine ng-untouched ng-invalid`. Then, when the user enters and leaves the field, it switches to `ng-pristine ng-touched ng-invalid`. When the user changes the value, still for an invalid one, we'll have `ng-dirty ng-touched ng-invalid`. And finally, when the value is valid: `ng-dirty ng-touched ng-valid`.

15.7. Creating a custom validator

Pony races are an addictive game so it's only allowed to register if you are over 18. And we want the user to enter the password twice, to be sure she/he hasn't made a mistake.

How do we do this? We create a custom validator.

To do so, we just have to create a method that takes a `Control`, tests its `value` and returns an object with the errors or `null`, if the validation passes.

```
let isOldEnough = (control: Control) => {
  let birthDatePlus18 = new Date(control.value);
  birthDatePlus18.setFullYear(birthDatePlus18.getFullYear() + 18);
  return birthDatePlus18 < new Date() ? null : {tooYoung: true};
};
```

Our validation method is pretty easy: we take the value of the control, we build the date, check if the 18th birthday is before now and return an error with the key 'tooYoung' if not.

Now we need to include this validator.

15.7.1. Using a validator in a model-driven form

We need to add a new control in our form with this validator, using the `FormBuilder`:

```
import {Component} from 'angular2/core';
import {FormBuilder, Control, ControlGroup, Validators} from 'angular2/common';

@Component({
  selector: 'register-form-cmp',
  viewProviders: [FormBuilder],
  templateUrl: './register_form_cmp.html'
})
export class RegisterFormCmp {
  username: Control;
  password: Control;
  birthdate: Control;
  userForm: ControlGroup;

  constructor(fb: FormBuilder) {
    this.username = fb.control('', Validators.required);
    this.password = fb.control('', Validators.required);
    this.birthdate = fb.control('', Validators.compose([Validators.required, this
.isOldEnough]));
    this.userForm = fb.group({
      username: this.username,
      password: this.password,
      birthdate: this.birthdate
    });
  }

  register() {
    console.log(this.userForm.value);
  }

  isOldEnough(control: Control) {
    // if there is no value
    if (!control.value) {
      return null;
    }
    let birthDatePlus18 = new Date(control.value);
    birthDatePlus18.setFullYear(birthDatePlus18.getFullYear() + 18);
    return birthDatePlus18 < new Date() ? null : {tooYoung: true};
  }
}
```

As you can see, we have added a new control `birthdate`, with two validators composed. The first validator is `required` and the other is a method of our class `isOldEnough`. Of course this method could be in another class if you wanted (`required` is a static method for example).

Don’t forget to add the field and display the errors in the form:

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [ngFormModel]="userForm">
  <div>
    <label>Username</label><input ngControl="username">
    <div *ngIf="username.dirty && username.hasError('required')">Username is
required</div>
  </div>
  <div>
    <label>Password</label><input type="password" ngControl="password">
    <div *ngIf="password.dirty && password.hasError('required')">Password is
required</div>
  </div>
  <div>
    <label>Birth date</label><input type="date" ngControl="birthdate">
    <div *ngIf="birthdate.dirty">
      <div *ngIf="birthdate.hasError('required')">Birth date is required</div>
      <div *ngIf="birthdate.hasError('tooYoung')">You're way too young to be betting on
pony races</div>
    </div>
  </div>
  <button type="submit" [disabled]="!userForm.valid">Register</button>
</form>
```

Pretty easy, no?

15.7.2. Using a validator in a template-driven form

To add a custom validator in a template-driven form, we need to add it in… the template!

To do this, we are going to build a directive that we will apply on the input.

Let's create this new directive. We start by creating a class, with the `@Directive` decorator, and a selector on an attribute that we will name `is-old-enough`.

```
import {Directive} from 'angular2/core';
import {Control, Validator} from 'angular2/common';

@Directive({
  selector: '[isOldEnough]'
})
export class IsOldEnoughValidator implements Validator {

  validate(control: Control): any {
    let birthDatePlus18 = new Date(control.value);
    birthDatePlus18.setFullYear(birthDatePlus18.getFullYear() + 18);
    return birthDatePlus18 < new Date() ? null : { tooYoung: true };
  }
}
```

The class has a method `validate`.

Now we need to add something to tell the input that this directive is a validator.

To do so, we are going to use dependency injection.

When it is instantiated, a form control directive is looking for all the directives of type `NG_VALIDATORS`. So we need to bind our directive as a `NG_VALIDATORS`, using the `multi` option. This option is the only way to bind several values to the same token.

```
import {Directive, provide} from 'angular2/core';
import {Control, NG_VALIDATORS, Validator} from 'angular2/common';

@Directive({
  selector: '[isOldEnough]',
  bindings: [provide(NG_VALIDATORS, {useExisting: IsOldEnoughValidator, multi: true})]
})
export class IsOldEnoughValidator implements Validator {

  validate(control: Control): any {
    let birthDatePlus18 = new Date(control.value);
    birthDatePlus18.setFullYear(birthDatePlus18.getFullYear() + 18);
    return birthDatePlus18 < new Date() ? null : {tooYoung: true};
  }
}
```

Now if we add the directive on the input the validation will be executed:

```
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input ngControl="username" required #username="ngForm">
    <div *ngIf="username.control?.dirty && username.control?.hasError('required')"
>Username is required</div>
  </div>
  <div>
    <label>Password</label><input type="password" ngControl="password" required #
password="ngForm">
    <div *ngIf="password.control?.dirty && password.control?.hasError('required')"
>Username is required</div>
  </div>
  <div>
    <label>Birth date</label><input type="date" ngControl="birthdate" required
isOldEnough #birthdate="ngForm">
    <div *ngIf="birthdate.control?.dirty">
      <div *ngIf="birthdate.control?.hasError('required')">Birth date is required</div>
      <div *ngIf="birthdate.control?.hasError('tooYoung')">You're way too young to be
betting on pony races</div>
    </div>
  </div>
  <button type="submit" [disabled]="!userForm.valid">Register</button>
</form>
```

15.8. Grouping fields

Until now, we just had one group: the complete form. But we can declare groups inside a group. That's very useful if you want to validate a group of fields together like an address, or, like in our example, if you want to check if the password and its confirmation match.

It's super easy to set up in a model-driven form, less so in a template-driven one. So I'm only going to show you the easy way :). First, create a new group, `passwordForm` with the two fields and add it in the group `userForm`:

```
import {Component} from 'angular2/core';
import {FormBuilder, Control, ControlGroup, Validators} from 'angular2/common';

@Component({
  selector: 'register-form-cmp',
  viewProviders: [FormBuilder],
  templateUrl: './register_form_cmp.html'
})
export class RegisterFormCmp {
  passwordForm: ControlGroup;
  userForm: ControlGroup;
  username: Control;
  password: Control;
  confirm: Control;

  constructor(fb: FormBuilder) {
    this.username = fb.control('', Validators.required);
    this.password = fb.control('', Validators.required);
    this.confirm = fb.control('', Validators.required);

    this.passwordForm = fb.group({password: this.password, confirm: this.confirm},
{validator: this.passwordMatch});

    this.userForm = fb.group({username: this.username, passwordForm: this.passwordForm});
  }

  register() {
    console.log(this.userForm.value);
  }

  passwordMatch(control: {controls: {password: Control, confirm: Control}}) {
    let password = control.controls.password.value;
    let confirm = control.controls.confirm.value;
    return password !== confirm ? {matchingError: true} : null;
  }
}
```

As you can see, we have added a validator on the group, `passwordMatch`, that will be called every time one of the fields changes.

Let's update the template to reflect the new form, using the `ngControlGroup` directive:

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [ngFormModel]="userForm">
  <div>
    <label>Username</label><input ngControl="username">
    <div *ngIf="username.dirty && username.hasError('required')">Username is
required</div>
  </div>
  <div ngControlGroup="passwordForm" #passwordForm="ngForm">
    <div>
      <label>Password</label><input type="password" ngControl="password">
      <div *ngIf="password.dirty && password.hasError('required')">Password is
required</div>
    </div>
    <div>
      <label>Confirm password</label><input type="password" ngControl="confirm">
      <div *ngIf="confirm.dirty && confirm.hasError('required')">Confirm your
password</div>
    </div>
    <div *ngIf="passwordForm.dirty && passwordForm.control.hasError('matchingError')"
>Your password does not match</div>
  </div>
  <button type="submit" [disabled]="!userForm.valid">Register</button>
</form>
```

Voilà!

15.9. Reacting on changes

Last cool feature when using a model-driven form: you can easily react on value changes, using the observable `valueChanges`. Reactive programming FTW! For example, let's say we want our password field to display a strength indicator. We want to compute the strength at every change of the password value:

```
import {Component} from 'angular2/core';
import {FormBuilder, Control, ControlGroup, Validators} from 'angular2/common';

@Component({
  selector: 'register-form-cmp',
  bindings: [FormBuilder],
  templateUrl: './register_form_cmp.html'
})
export class RegisterFormCmp {
  userForm: ControlGroup;
  username: Control;
  password: Control;
  passwordStrength: number = 0;

  constructor(fb: FormBuilder) {
    this.username = fb.control('', Validators.required);
    this.password = fb.control('', Validators.required);

    this.userForm = fb.group({
      username: this.username,
      password: this.password
    });

    // we subscribe to every password change
    this.password.valueChanges.subscribe((change) => {
      this.passwordStrength = change.length;
    });
  }

  register() {
    console.log(this.userForm.value);
  }
}
```

Now we have a `passwordStrength` field in our component instance, that we can display to our user:

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [ngFormModel]="userForm">
  <div>
    <label>Username</label><input ngControl="username">
    <div *ngIf="username.dirty && username.hasError('required')">Username is
required</div>
  </div>
  <div>
    <label>Password</label><input type="password" ngControl="password">
    <div>Strength: {{passwordStrength}}</div>
    <div *ngIf="password.dirty && password.hasError('required')">Password is
required</div>
  </div>
  <button type="submit" [disabled]="!userForm.valid">Register</button>
</form>
```

15.10. Summary

Angular 2 offers two ways to build a form:

- one by setting up everything in the template. But, as you have seen, it is quickly verbose when we have several fields, forces us to have custom directives for the validation and is harder to test. This way of doing things is useful for very simple forms, with just one field for example.
- one by setting up almost everything in the component. This way allows an easier setup for validation and testing, with several levels of groups if you need them. It is your weapon of choice for building complex forms. You can even react on changes on a group, or on a field.
- `ngModel` gives you two-way data-binding if you need it. You can use it with both the template-driven forms and the model-driven ones.

NOTE It's possible to have a form NOT transformed into its superpowered Angular 2 version by using the `ng-no-form` attribute.

Chapter 16. Send and receive data with Http

That won't come as a surprise, but a lot of our job consists in asking a backend server to send data to our webapp, and then sending data back.

Usually this is done over HTTP, even though you have other alternatives nowadays, like WebSockets. Angular 2 provides an http module, but doesn't force you to use it. If you prefer, you can use your favorite HTTP library to send asynchronous requests.

One of the newcomers is the fetch API, which is currently available as a polyfill, but should become a standard in browsers. You can perfectly build your app using `fetch` or another library. In fact, that's what I used before the Http part was done in Angular 2. It works great, with no need of special calls to make the framework aware that we have received data and that it needs to run the change detection (unlike in AngularJS 1.x, where you would have to call `$scope.apply()` if you were using an external library: that's the magic of Angular 2 and its zones!).

But if you feel comfortable with the framework, you will use a small module called `http`, provided by the core team. It is an independent module, so, really, do as you like. Note that it mirrors closely enough the Fetch API proposal.

If you want to use it, you have to use the classes from the `angular2/http` module.

Why prefer this module over, say, `fetch`? The answer is simple: testing. As we will show, the Http module allows to mock your backend server and return fake responses. That's really, really useful.

Last thing before we dive into the API: the Http module heavily uses the reactive programming paradigm. So if you skipped the Reactive Programming chapter , now might be a good time to go back and read it ;).

16.1. Getting data

The Http module offers a service called `Http` that you can inject in any constructor. As the service is coming from another module, you have to manually make it available to your component or service. The easiest way is to add the `HTTP_PROVIDERS` in the `bootstrap` method:

```
bootstrap(PonyRacerApp, [HTTP_PROVIDERS]);
```

Once this is done, you can inject the `Http` service wherever you need it:

```
@Component({
  selector: 'ponyracer-app',
  template: `<h1>PonyRacer</h1>`
})
export class PonyRacerApp {

  constructor(private http: Http) {
  }

}
```

By default, the `Http` service will do AJAX request using `XMLHttpRequest`.

It offers several methods, matching the most common HTTP verbs:

- `get`
- `post`
- `put`
- `delete`
- `patch`
- `head`

All these methods are in fact just syntactic sugar, and they all call the method `request` under the hood.

If you used the `$http` service in AngularJS 1.x, you might remember that it heavily relied on Promises. In Angular 2, however, all these methods return an `Observable` object.

A few advantages come with the use of Observables for `Http` like the ability to cancel requests, to retry, to easily compose them, etc...

Let's start by fetching the races registered in PonyRacer. We'll assume that a backend is already up and running, providing a RESTful API. To fetch the races, we'll send a GET request to a URL like 'http://backend.url/api/races'.

Usually, the base URL of your HTTP calls will be stored in a variable or a service, that you can easily configure depending on your environment. Or, if the REST API is served by the same server as the Angular application, you can simply use a relative URL: '/api/races'.

Using the `Http` service, such a request is straightforward:

```
http.get(`${baseUrl}/api/races`)
```

This returns an Observable, to which you can subscribe to receive the response. The response is a `Response` object, with a few fields and handy methods. You can easily access the `status` code, `headers`, etc...

```
http.get(`${baseUrl}/api/races`)
  .subscribe(response => {
    console.log(response.status); // logs 200
    console.log(response.headers); // logs []
  });
```

Of course, the Response body is the most interesting part. But, to access it, you have to use a method:

- `text()` if you expect some text
- `json()` if you expect a JSON object, the parsing being done for you

```
http.get(`${baseUrl}/api/races`)
  .subscribe(response => {
    console.log(response.json());
    // logs the array of races
  });
```

Sending data is fairly easy too. Just call the `post()` method, with the URL and the object to post:

```
http.post(`${baseUrl}/api/races`, newRace)
```

I won't show you the other methods, I'm sure you get the idea.

16.2. Transforming data

As we are getting an Observable object as a response, don't forget you can easily transform the data.

Want a list of the races' names? Request the races and map the names!

To do so, you currently have to import the `map` operator for RxJS. The Angular team doesn't want to include the full RxJS library in our apps, so you have to explicitly import the needed operators:

```
import 'rxjs/add/operator/map';
```

You can then use it:

```
http.get(`${baseUrl}/api/races`)
  // extract json body
  .map(res => res.json())
  // transform [races] => [names]
  .map((races: Array<any>) => races.map(race => race.name))
  .subscribe(names => {
    console.log(names);
    // logs the array of the race's names
  });
```

This kind of work will usually be done in a dedicated service. I tend to create a service, like `RaceService`, where all the job is done. Then, my component just needs to subscribe to my service method, without knowing what's going on under the hood.

```
raceService.list()
  // transform [races] => [names]
  .map(races => races.map(race => race.name))
  .subscribe(names => {
    console.log(names);
    // logs the array of the race's names
  });
```

You can also leverage the power of RxJS to retry a failed request a few times, for example.

```
raceService.list()
  // if the request fails, retry 3 times
  .retry(3)
  .map(races => races.map(race => race.name))
  .subscribe(names => {
    console.log(names);
    // logs the array of the race's names
  });
```

16.3. Advanced options

Of course, you can tune your requests more finely. Every method takes a `RequestOptions` object as an optional parameter, where you can configure your request. A few options are really useful and you can override everything in the request. Some of these options have the exact same values as the ones offered by the Fetch API. The `url` option is pretty obvious and will override the URL of the request. The `method` option is the HTTP verb to use, like `RequestMethod.Get`. If you want to build a request manually, you can write:

```
let options = new RequestOptions({method: RequestMethod.Get});

http.request(`${baseUrl}/api/races/3`, options)
  .subscribe(response => {
    // will get the race with id 3
  });
```

`search` represents the URL params to add to the URL. You specify them using the `URLSearchParams` class, and the complete URL will be constructed for you.

```
let searchParams = new URLSearchParams();
searchParams.set('sort', 'ascending');

let options = new RequestOptions({search: searchParams});

http.get(`${baseUrl}/api/races`, options)
  .subscribe(response => {
    // will return the races sorted
    races = response.json();
  });
```

The `headers` option is often useful to add a few custom headers to your request. It happens to be necessary for some authentication techniques like JSON Web Token for example:

```
let headers = new Headers();
headers.append('Authorization', `Bearer ${token}`);

http.get(`${baseUrl}/api/races`, new RequestOptions({headers}))
  .subscribe(response => {
    // will return the races visible for the authenticated user
    races = response.json();
  });
```

16.4. Jsonp

To let you access their API without being blocked by the Same Origin Policy enforced by web browsers, some web services don't use CORS, but use JSONP (JSON with Padding).

The server will not return the JSON data directly, but wrap them in the function passed as a callback. The response comes back as a script, and scripts are not subject to the Same Origin Policy. Once loaded, you can access the JSON value contained in the response.

In addition to the `Http` service, the Http module also provides a `Jsonp` service that makes it easy to

interact with such APIs, and does all the dirty job for us. All you have to do is specify the URL of the service you want to call, and add `JSONP_CALLBACK` as the callback parameter value.

In the following example, we are fetching all the public repos from our Github organization using JSONP.

```
jsonp.get('https://api.github.com/orgs/Ninja-Squad/repos?callback=JSONP_CALLBACK')
  // extract json
  .map(res => res.json())
  // extract data
  .map(res => res.data)
  .subscribe(response => {
    // will return the public repos of Ninja-Squad
    repos = response;
  });
```

16.5. Tests

We now have a service calling an HTTP endpoint to fetch the races. How do we test it?

```
@Injectable()
export class RaceService {
  constructor(private http: Http) {
  }

  list() {
    return this.http.get('/api/races').map(res => res.json());
  }
}
```

In a unit test, you don't want to really call the HTTP server: that's not what we are testing. We want to "fake" the HTTP call to return fake data. To do this, we can replace the dependency to the `Http` service with a fake implementation using a class provided by the framework called `MockBackend`:

```
import {describe, it, expect, beforeEachProviders, inject} from 'angular2/testing';
import {Http, BaseRequestOptions, Response, ResponseOptions} from 'angular2/http';
import {MockBackend} from 'angular2/http/testing';

import {provide} from 'angular2/core';
import 'rxjs/add/operator/map';

describe('RaceService', () => {

  beforeEachProviders(() => [
    MockBackend,
    BaseRequestOptions,
    provide(Http, {
      useFactory: (backend, defaultOptions) => new Http(backend, defaultOptions),
      deps: [MockBackend, BaseRequestOptions]
    }),
    RaceService
  ]);

  it('should return an Observable of 2 races', inject([RaceService, MockBackend],
(service, mockBackend) => {
    // fake response
    let hardcodedRaces = [new Race('London'), new Race('Lyon')];
    let response = new Response(new ResponseOptions({body: hardcodedRaces}));
    // return the response if we have a connection to the MockBackend
    mockBackend.connections.subscribe(connection => connection.mockRespond(response));

    service.list().subscribe(races => {
      expect(races.length).toBe(2);
    });
  }));
});
```

And we're done!

Chapter 17. Router

It is fairly common to want to map a URL to a state of the application. That makes sense: you want your user to be able to bookmark a page and come back, and it provides a better experience overall.

The piece in charge of doing this job is called a router, and every framework has its own (or several ones).

The router in Angular 2 has a simple goal: allowing to have meaningful URLs reflecting the state of our app, and for each URL to know which component should be initialized and inserted in the page.

You probably know there was already a router in AngularJS 1.x, maintained by the core team, in a module called `ngRoute`. You may also know that it was a very simplistic one: OK for simple applications, but it was only allowing a single view per URL and no nesting was possible. It was a bit limited to work on bigger apps, where you often have views inside views. There was a very popular community module, called `ui-router`, that a lot of people were using and which was doing a really great job.

In Angular 2, the team has decided to bridge the gap and has written a new module called `router`. This module will hopefully fulfill all our needs!

Some new features are really interesting. So let's go!

WARNING	The router module is still a work in progress, so this chapter is only a quick introduction. We'll dive deeper into it in a future update, when the module is done.

17.1. En route

Lets's start using the router. It is an optional module, that is thus not included in the core framework. So we have to add the `ROUTER_PROVIDERS` to our injector, usually via the bootstrap method:

```
bootstrap(PonyRacerApp, [ROUTER_PROVIDERS]);
```

We also need to tell the router which component is the main one, by binding `ROUTER_PRIMARY_COMPONENT`.

```
bootstrap(PonyRacerApp, [
  ROUTER_PROVIDERS,
  provide(ROUTER_PRIMARY_COMPONENT, {useValue: PonyRacerApp})
]);
```

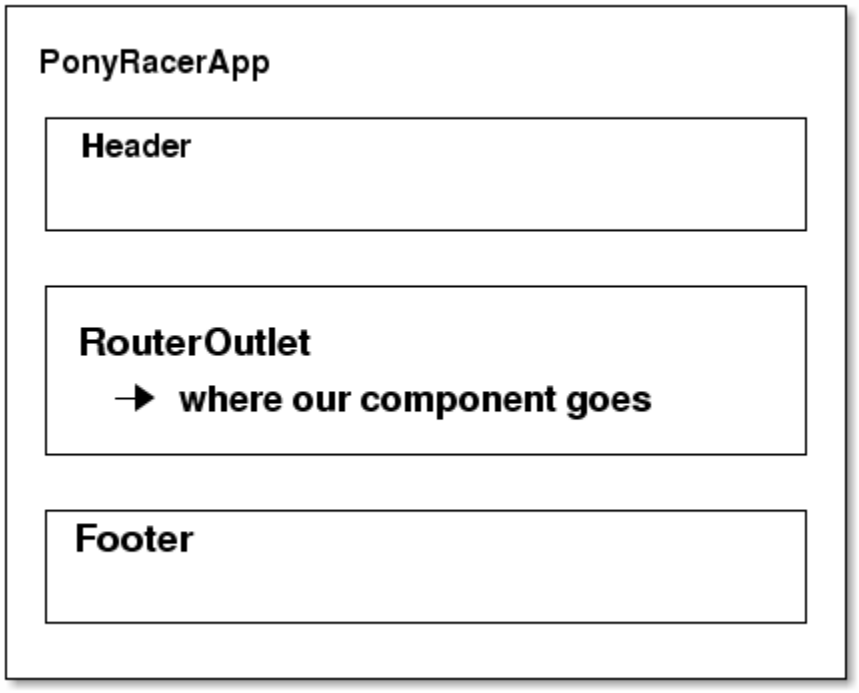
Then we need to define the mapping between URLs and components. We can do this with a decorator named `@RouteConfig` exposed by the router module, usually on top of your other decorators on the main component:

```
@RouteConfig([
  new Route({path: '/', component: HomeCmp, name: 'Home'}),
  new Route({path: '/races', component: RacesCmp, name: 'Races'}),
])
@Component({
  selector: 'ponyracer-app',
```

As you can see, the `@RouteConfig` decorator takes an array of objects, each one being a `Route` instance. A route configuration is usually a triplet of properties:

- `path`: what URL will trigger the navigation
- `component`: which component will be initialized and inserted
- `name`: what name we will use to reference this route. The name must be in CamelCase, and will be used to invoke the route.

You may be wondering where the component will be inserted in the page, and that's a good question. For a component to be included in our app, like the `RacesCmp` in the example above, we must use a special tag in the template of the primary component: `<router-outlet>`.



This is, of course, an Angular directive, whose only job is to act as a placeholder for the template of the component of the current route. Our app template would look like:

```
<header>
  <nav-bar>...</nav-bar>
</header>
<main>
  <router-outlet>
    <!-- this will be replaced by the component's template -->
  </router-outlet>
</main>
<footer>made with &lt;3 by Ninja Squad</footer>
```

When we navigate, everything will stay (the header, main and footer here) except the content of the `RouterOutlet` directive which will be replaced by the matching component. Of course, we need to add `RouterOutlet` to our component directives if we want Angular to pick it up. More conveniently, you can include `ROUTER_DIRECTIVES`: it is an array containing all the router directives.

```
@RouteConfig([
  new Route({path: '/', component: HomeCmp, name: 'Home'}),
  new Route({path: '/races', component: RacesCmp, name: 'Races'}),
])
@Component({
  selector: 'ponyracer-app',
  template: `
  <header>
    <nav-bar>...</nav-bar>
  </header>
  <main>
    <router-outlet>
      <!-- this will be replaced by the component's template -->
    </router-outlet>
  </main>
  <footer>made with &lt;3 by Ninja Squad</footer>
`,
  directives: [ROUTER_DIRECTIVES]
})
export class PonyRacerApp {

  constructor(public router: Router) {
  }

}
```

17.2. Navigation

How can we navigate between the different components? Well, you can manually type the URL and

reload the page, but that's not very convenient.

In a template, you can add a link with the directive `RouterLink` pointing to the route alias you want to go to. The `RouterLink` directive must receive an array of strings, representing the route alias and its params. For example in our `RacesCmp` template, if we want to navigate to the `HomeCmp`, we can imagine something like:

```
<a href="" [routerLink]="['/Home']">Home</a>
```

At runtime, the link `href` will be computed by the router and will point to `/`.

The `RouterLink` directive also offers a few goodies, like the CSS class `router-link-active` which is automatically added if the link points to the current route. This allows, for example, to style a menu item as selected when it points to the current page.

When the router is ready, this chapter will contain additional information:

- how to use URL parameters
- how to nest routes
- how to associate data with routes
- how to resolve data before activating a route and displaying a component

Chapter 18. This is the end

Thanks for reading!

There are some other chapters that will be added in the following releases, like the router (which was not completely done for this first release), the advanced stuff and some other goodies. They all need a little more polish, but I'm sure you'll enjoy them. And of course, we'll keep up with the framework releases, so you won't miss the new shiny features that will come out. All these future updates of the book will be available for free, of course!

If you liked what you read, tell your friends about it! And if you want to go further, check out our pro package, where you can put in practice what you've learned, and learn even more!

We have tried to give you all the keys, but Web Development looks an awful lot like:

How to draw a horse. Credit to Van Oktop.

So we also provide training, mainly in France and Europe, but all over the world really. We can also do some consulting work to help your team, or work with you to help you build your product. Just shoot us an email at hello@ninja-squad.com and we'll discuss it!

Overall, I would love hearing from you and find out what you liked, loved and hated in this ebook - whether you are writing to signal a small typo, a big mistake, or just to tell us that this book helped you find your dream job with this book (well, you never know…).

I can't finish without thanking a few people. My girlfriend, first, who has been an incredible support,

even when I was rewriting something for the tenth time, in a dreadful mood on a Sunday. My colleagues, for their tireless work and feedback, their kindness for encouraging me and giving me the time to do this crazy thing. And my friends and family, for the little words that kept me going.

And you, for buying this and reading it to the last sentence.

Stay tuned.