Become a NINJA with

ANGULAR

ninja squad

Become a ninja with Angular

Ninja Squad

Table of Contents

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 hundreds 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 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 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 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 fewer 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!

The first 6 exercises of the Pro Pack are free. The other ones are only accessible if you buy our online training. At the end of every chapter, we will link to the exercises of the Pro Pack that are related to the features explained in the chapter, mark the free ones with the following label: 🐴 , and mark the other ones with the following label: 🦄.

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.

NOTE	The ebook is using Angular version 4.3.0 for the examples.

NOTE

Angular and versioning

This book used to be named "Become a Ninja with Angular 2". Because, originally, Google named its framework Angular 2. But in October 2016, they reviewed their versioning and release policy.

We'll have a major release every six months, according to the plan. And now the framework should be called just "Angular".

Don't worry, these releases are not a complete rewrite with no backward compatibility like Angular 2 was to AngularJS 1.x.

As this ebook will be updated (for free) with all these following major releases, it is now named "Become a Ninja with Angular" (without any number).

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 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 (even if, for once, Microsoft is doing a good job with Edge). 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 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
  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 poniesInRace = 6;
```

```
poniesInRace = 7; // SyntaxError
```

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

can be simplified to

```
function createPony() {
  const name = 'Rainbow Dash';
  const 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:

```
const httpOptions = { timeout: 2000, isCache: true };
// later
const { 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:

```
const httpOptions = { timeout: 2000, isCache: true };
// later
const { 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:

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

And the same is possible with arrays:

```
const timeouts = [1000, 2000, 3000];
// later
const [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() {
  const pony = { name: 'Rainbow Dash' };
  const position = 2;
  // ...
  return { pony, position };
}

const { 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() {
  const pony = { name: 'Rainbow Dash' };
  const position = 2;
  // ...
  return { pony, position };
}

const { 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 fewer 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 fewer 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:

```
const { 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:

```
const [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:

```
const ponyPrices = [12, 3, 4];
const 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!
  }
}
const 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:

```
const 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;
  }
}
const pony = new Pony();
pony.color = 'red';
// 'set color red'
console.log(pony.color);
// 'get color'
// 'red'
```

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 {

}
const 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;
  }
}
const 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;
  }
}
const 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, 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 their 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 their 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`.

```
const 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:

```
const 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`:

```
const cedric = { id: 1, name: 'Cedric' };
const 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`:

```
const cedric = { id: 1, name: 'Cedric' };
const 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:

```
const 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:

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

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

```
const 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, 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 another file:

```
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, 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 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:

```
const pony = { name: 'Rainbow Dash', color: 'blue' };
const horse = { speed: 40, color: 'black' };
const 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, 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 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, 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 Ballmer 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 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 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 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
const fakeService = {
  list: () => {
    // returns a fake promise
  }
};
const 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, 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, 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 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:

```
const poneyNumber: number = 0;
const 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`:

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

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

```
const 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}
    const 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);
    }

    const 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 = 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`);
  }
}
    const pony = new NamedPony('Rainbow Dash', 10);
    // defines a public property name with 'Rainbow Dash'
    // and a private one speed with 10
```

Which is the same as the more verbose:

```
class NamedPonyWithoutShortcut {
  public name: string;
  private speed: number;

  constructor(name: string, speed: number) {
    this.name = name;
    this.speed = speed;
  }

  run() {
    logger.log(`pony runs at ${this.speed}m/s`);
  }
}
```

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

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 directly with NPM:

```
npm install --save-dev @types/angular
```

or download it manually. 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. So you don't even have to worry about including the interfaces in your Angular 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 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 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, 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:

```
const 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 looks like:

```
@Component({ selector: 'ns-home' })
class HomeComponent {

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

}
```

The `@Component` decorator is added to the class `Home`. When Angular 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 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 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 code. But I think it's a good thing to have an overview of what they are, because some choices in Angular 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 `<ns-pony></ns-pony>` 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('ns-pony')`:

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

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 `ns-pony` displays an image of the pony or just its name:

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

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

If you try to look at the DOM, you'll see `<ns-pony><h1>General Soda</h1></ns-pony>`. 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 `<ns-pony></ns-pony>` 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 PonyComponentProto = Object.create(HTMLElement.prototype);

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

var PonyComponent = document.registerElement('ns-pony', {prototype:
PonyComponentProto});
document.body.appendChild(new PonyComponent());
```

If you try to inspect it now you should see:

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

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 PonyComponentProto = Object.create(HTMLElement.prototype);

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

var PonyComponent = document.registerElement('ns-pony', {prototype:
PonyComponentProto});
document.body.appendChild(new PonyComponent());
```

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="ns-pony.html">`. This file, `ns-pony.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="ns-pony">
  <template>
    <h1>[[name]]</h1>
  </template>
  <script>
    Polymer({
      is: 'ns-pony',
      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="ns-pony.html">

<!-- Use element -->
<body>
  <ns-pony name="General Soda"></ns-pony>
</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.

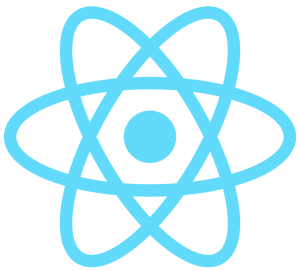
That's obviously far too short a chapter to tell you everything there is to say on Web Components, but you'll see that some of the concepts are going to pop out along your read. And you'll definitely see that the Google team designed Angular to make it easy to use Web Components along our Angular components.

Chapter 6. Grasping Angular’s philosophy

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

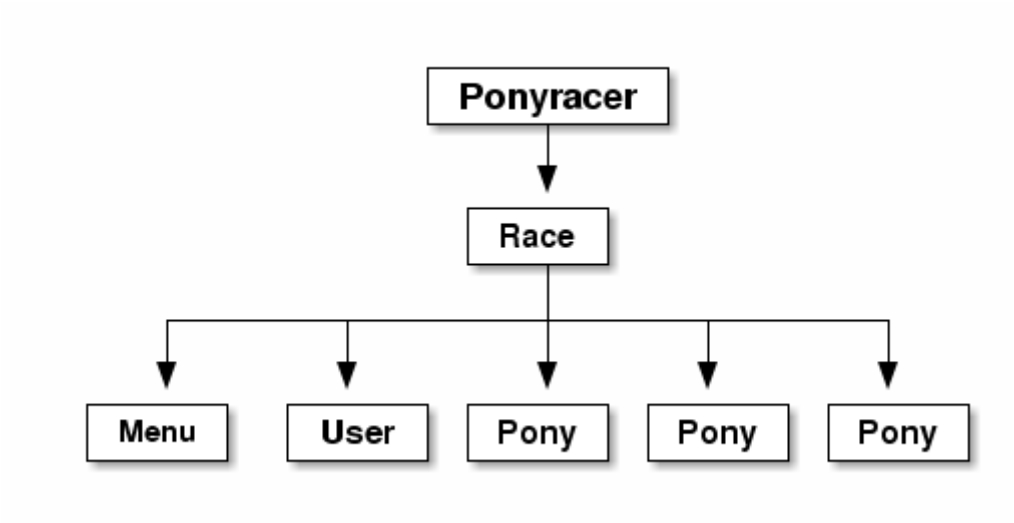
First and foremost, Angular is component-oriented. You will write tiny components and, together, they will constitute a whole application. A component is a group of HTML elements in a template, dedicated to a particular task. For this, you will usually also need to have some logic linked to that template, 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 is not alone in this, but it is among the first (it 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 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 '@angular/core';
import { RacesService } from './services';

@Component({
  selector: 'ns-race',
  templateUrl: 'race/race.html'
})
export class RaceComponent {

  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="let pony of race.ponies">
    <ns-pony [pony]="pony"></ns-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.

You will group components in one or several coherent entities, called modules (Angular Modules, not ES6 Modules).

In a perfect world, you will also take available modules from the community and just put them in your app, and be able to enjoy their features.

Such modules can offer UI components, or drag and drop capability, or validation for your forms, or whatever you can think of.

In the next chapters, we are going to explore how to get started, how to build a small component, your first module 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). It is 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 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 4.4+. 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
--lib es6,dom
```

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",
    "module": "commonjs",
    "lib": ["es6","dom"],
    "sourceMap": true,
    "strict": true,
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}
```

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.

We now need to add the Angular library and our code. For the Angular 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 3. Check which version you have with:

```
npm -v
```

If you don't have NPM version 3, you can easily update NPM:

```
npm install -g npm
```

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 Angular and its dependencies.

NOTE

The ebook is using Angular version 4.3.0 for the examples. The following command will install the most recent version, which might not be the same. If you want to use the same version as us, add `@4.3.0` to each Angular package, like `@angular/core@4.3.0`. That might save you a few headaches! Angular is really modular, so we have to install a few packages for the framework itself, and for its dependencies.

```
npm install --save @angular/core @angular/compiler @angular/common @angular/platform-
browser @angular/platform-browser-dynamic rxjs reflect-metadata zone.js
```

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

- the different `@angular` packages.
- `reflect-metadata`, as we are using decorators.
- `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).

Last thing to make the compiler happy, you have to install the typings for the things related to ES6. The easiest way is to install the typings for `core-js`:

```
npm install --save-dev @types/core-js
```

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

7.2. Our first component

Create a new file, called `app.component.ts`.

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.

When you save your file, you should see a new file `app.component.js` popping in the directory: it's the TypeScript compiler doing its job. You should see the source map file as well. 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 PonyRacerAppComponent {

}
```

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 '@angular/core';

@Component()
export class PonyRacerAppComponent {

}
```

If your IDE supports it, code completion should work as the Angular 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 '@angular/core';

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

}
```

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

NOTE

There is not a clear naming convention established yet. I tend to suffix my component classes with `Component`. A component's selector should have a dash, like `ns-pony`, 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 '@angular/core';

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

}
```

Don't forget to import the `Component` decorator. 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 `@angular/core` module, but that's not always the case. For example, when dealing with HTTP, we'll use imports from `@angular/http`; or, if we use the router, we'll import from `@angular/router`, etc.

7.3. Our first Angular Module

Like we briefly said in the previous chapter, we are going to group our components and other pieces we'll see later in coherent entities: Angular Modules.

An Angular Module is different from the ES6 Modules we crossed earlier: here we are talking about **application** modules.

Your application will always have at least one module, the **root module**. Maybe, later, when your application grows, you'll add other modules, by feature. For example, you could add a module dedicated to the Admin part of your application, containing all the components and the logic for this part. But we'll come back to this later. We'll also see that third party libraries and Angular itself expose modules, that we can use in our app.

To define an Angular Module for our little app, we have to create a class. Usually, this is done in a separate file, called `app.module.ts` for the root module.

The class has to be decorated with `@NgModule`.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports: [BrowserModule],
})
export class AppModule {

}
```

Like the `@Component` decorator, it takes a configuration object.

As we are building an app for the browser, the root module should import `BrowserModule`. This is not the only target possible for Angular, you could choose to render the app on the server for example, and therefore import another module. `BrowserModule` contains all kinds of useful stuff we will use later. A module can choose to *export* components, directives and pipes. When you import a module, you will make all the directives, components and pipes exported by the imported module usable in your module. Our root module won't be imported in another module, so we don't have `exports`, but we will have several `imports` in the end.

The terminology is not beginner friendly here. We were talking about ES6 and TS modules in the first chapters, which define imports and exports. And now we are talking about Angular modules, which also have imports and exports... I'm not a fan of having the same terms for different things, so let me explain a little bit more.

You can see an ES6 or TS import purely as a language feature, like an import statement in Java: it allows using the imported classes/functions in your source code. It also declares a dependency for the bundler or module loader (Webpack or SystemJS, for example), which knows that if `a.ts` is loaded, then `b.ts` must also be loaded since `a` imports `b`. You have to use imports and exports with ES6 and TypeScript, whether or not you're using Angular or any other framework.

On the other hand, importing an Angular module (for example `BrowserModule`) in your own Angular module (`AppModule`), has a functional meaning. It tells Angular: all the components, directives and pipes that are exported by `BrowserModule` should be made available to my Angular components/templates. It has no special meaning for the TypeScript compiler.

Back to `NgModule`: in its configuration object, we must declare the components that belong to our root module, with the `declarations` field. Let's add the component we have carefully crafted: `PonyracerAppComponent`.

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

@NgModule({
  imports: [BrowserModule],
  declarations: [PonyRacerAppComponent],
})
export class AppModule {

}
```

Since this is the root module, we need to tell Angular which component is the root component, i.e. the component that will be started when we bootstrap the app. That's what the `bootstrap` field of the configuration object is for:

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

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

}
```

Module ready, let's bootstrap the app!

7.4. Bootstrapping the app

Finally, we need to start our app, using the `bootstrapModule` method. This method is exposed on an object returned by a method called `platformBrowserDynamic`. You have to import it too, from `@angular/platform-browser-dynamic`. Now, that's a strange module! Why is it not `@angular/core`? Good question: it's because you might want to run your app somewhere else than in a browser, as Angular 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 `main.ts`, to separate the bootstrap logic:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

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, things are more complicated, but complexity comes with additional power. Angular is now bundled in modules (ES6 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.

Note that this is not the only tool that can do this job, you could use another one like Webpack if you wanted to.

Let's install SystemJS:

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

We need to load `SystemJS` statically, and to tell it where our bootstrap module is (in `main`). We also need to tell it where to find the dependencies of our application, like `@angular`. But first, we need to include `reflect-metadata` and `zone.js`:

```
<html>

<head>
  <script src="node_modules/zone.js/dist/zone.js"></script>
  <script src="node_modules/reflect-metadata/Reflect.js"></script>
  <script src="node_modules/systemjs/dist/system.js"></script>
  <script>
    System.config({
      // the app will need the following dependencies
      map: {
        '@angular/core': 'node_modules/@angular/core/bundles/core.umd.js',
        '@angular/common': 'node_modules/@angular/common/bundles/common.umd.js',
        '@angular/compiler': 'node_modules/@angular/compiler/bundles/compiler.umd.js',
        '@angular/platform-browser': 'node_modules/@angular/platform-
browser/bundles/platform-browser.umd.js',
        '@angular/platform-browser-dynamic': 'node_modules/@angular/platform-browser-
dynamic/bundles/platform-browser-dynamic.umd.js',
        'rxjs': 'node_modules/rxjs'
      },
      packages: {
        // we want to import our modules without writing '.js' at the end
        // we declare them as packages and SystemJS will add the extension for us
        '.': {}
      }
    });
    // and to finish, let's boot the app!
    System.import('main');
  </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.

7.5. 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.

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 Webpack as a build tool, 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 apps in the future, so you can give it a try.

In fact I think you should give it a try: you will have the equivalent of what we just did manually, plus a ton of cool stuff.

```
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 go have a look at `Angular CLI`!

PRACTICE	Try our exercise Getting Started 🐴! It's free and part of our Pro Pack, where you'll learn how to build a complete application step by step. The first exercise is about getting everything up and running with 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 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 '@angular/core';

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

}
```

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 PonyRacerAppComponent {

  numberOfUsers = 146;

}
```

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 PonyRacerAppComponent {

  numberOfUsers = 146;

}
```

We have a `PonyRacerAppComponent` component that will be activated every time Angular finds a `<ponyracer-app>` tag. The `PonyRacerAppComponent` 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 `PonyRacerAppComponent` class, and this instance is the evaluation context of the template's expressions. Here the `PonyRacerAppComponent` 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 PonyRacerAppComponent {

  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 PonyRacerAppComponent {

  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
PonyRacerAppComponent]
```

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. 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 PonyRacerAppComponent {

  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.component.ts
import { Component } from '@angular/core';

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

}
```

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 `PonyRacerAppComponent` template. What do we need to do?

8.2. Using other components in our templates

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

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

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

}
```

As you can see, we added the `RacesComponent` 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.

Why is that? The reason is simple: Angular doesn't know about this `RacesComponent` yet.

But the fix is simple. Do you remember that we had to add `PonyRacerAppComponent` in the `declarations` of the `@NgModule` decorator? Now, since we have a second component, it must be declared, too.

`RacesComponent` is not the root component of our application, so it must be in the declarations, but not in the `bootstrap`.

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

import { PonyRacerAppComponent } from './app.component';
// do not forget to import the component
import { RacesComponent } from './races.component';

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

Note that you will pass the class directly, so you'll have to import it. And in order to be able import it, you need to export the class `RacesComponent` in its source file `races.component.ts` (read the section about ES6 modules again if that's not clear for you). So `RacesComponent` will look like:

```
// in another file, races.component.ts
import { Component } from '@angular/core';

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

}
```

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

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

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 templating system: property binding.

In Angular, 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 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:

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

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, you can do:

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

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

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

is the same as `[property]="expression"` (which you will usually prefer):

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

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

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

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

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

All of these are equivalent, and the syntax doesn't depend on how the developer chose to design their 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:

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

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, this is simpler, no more directives to remember.

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

Reacting on an event can be done as follow:

```
<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: 'ns-races',
  template: `
    <h2>Races</h2>
    <button (click)="refreshRaces()">Refresh the races list</button>
    <p>{{ races.length }} races</p>
  `
})
export class RacesComponent {
  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>
  <ns-races>
    <h2>Races</h2>
    <button (click)="refreshRaces()">Refresh the races list</button>
    <p>0 races</p>
  </ns-races>
</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 `RacesComponent` component emits a custom event to notify the app that a new race is available.

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

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

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

```
<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 `RacesComponent` is still not displaying the races :) The "proper way" in Angular would imply to create another component `RaceComponent` 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 behavior to an element.

The structural directives provided by Angular rely on using a `ng-template` element, inspired from the `template` standard tag of the HTML specification. It even used to called `template` before version 4.0, but it's now deprecated, and you should use `ng-template`:

```
<ng-template>
  <div>Races list</div>
</ng-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 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`:

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

The framework provides a few directives, like `ngIf`. They come from the module we imported earlier: `BrowserModule`. You can also define your own directives 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 or will not display the `div` whenever the value of `races` changes: 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 '@angular/core';

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

It's also possible to use a `else` syntax since the version 4.0:

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

@Component({
  selector: 'ns-races',
  template: `
    <div *ngIf="races.length > 0; else empty"><h2>Races</h2></div>
    <ng-template #empty><h2>No races.</h2></ng-template>
  `
})
export class RacesComponent {
  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 `RacesComponent` component contains a field `races` which, as you can probably guess, is an array of races to display.

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

@Component({
  selector: 'ns-races',
  template: `<div *ngIf="races.length > 0">
  <h2>Races</h2>
  <ul>
    <li *ngFor="let race of races">{{ race.name }}</li>
  </ul>
  </div>`
})
export class RacesComponent {
  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="let race of races">{{ race.name }}</li>
</ul>
```

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

```
<ul>
  <ng-template ngFor let-race [ngForOf]="races">
    <li>{{ race.name }}</li>
  </ng-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 variable `race` allowing us to use it in the interpolation expression, and reflecting the current element.

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

```
<ul>
  <li *ngFor="let 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="let race of races; index as i">{{ 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 assign 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
- `first`, a boolean that is true if the element is the first of the collection
- `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 *ngSwitchCase="0">You have no message</p>
  <p *ngSwitchCase="1">You have a message</p>
  <p *ngSwitchDefault>You have some messages</p>
</div>
```

As you can see, `ngSwitch` takes a condition and the `*ngSwitchCase` 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:

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

or, using the canonical syntax:

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

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 `ref-`:

```
<input type="text" ref-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 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 `PoniesComponent` component, displaying a list of ponies. Each pony should be represented by a `PonyComponent` 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 '@angular/core';

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

}
```

You can add it to the `PonyRacerAppComponent` 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 `<ns-ponies></ns-ponies>` in the template.

Our new component has a list of ponies:

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

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

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

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

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

One thing is missing, the refresh button:

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

@Component({
  selector: 'ns-ponies',
  template: `<button (click)="refreshPonies()">Refresh</button>
  <ul>
    <li *ngFor="let pony of ponies">{{ pony.name }}</li>
  </ul>`
})
export class PoniesComponent {
  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 `isEven` exported variable:

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

@Component({
  selector: 'ns-ponies',
  template: `<button (click)="refreshPonies()">Refresh</button>
  <ul>
    <li *ngFor="let pony of ponies; even as isEven"
      [style.color]="isEven ? 'green' : 'black'">
      {{ pony.name }}
    </li>
  </ul>`
})
export class PoniesComponent {
  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!

PRACTICE

Try our two exercises Templates 🐴 and List of races 🐴! They are free and part of our Pro Pack, where you'll learn how to build a complete application step by step. The first one is all about building a small component, a responsive menu, and play with its template. The second guides you to build another component: the list of races.

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 create its dependencies, the idea is to let the framework create them, and provide them 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 an `ApiService` service, already written by one of your colleague. Since he's the lazy guy of the team, he just wrote a class with a method named `get` returning an empty array, but you can already guess that the service will be used to communicate with a backend API.

```
export class ApiService {
  get(path) {
    // todo: call the backend API
  }
}
```

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

Let's say we want to write a `RaceService` that would use the `ApiService`:

```
import { ApiService } from './api.service';

export class RaceService {

  constructor(private apiService: ApiService) {
  }

}
```

Angular will fetch the `ApiService` service for us and inject it into our constructor. When `RaceService` is needed, the constructor will be called, and we will have an `apiService` field referencing the `ApiService` service.

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

```
import { ApiService } from './api.service';

export class RaceService {

  constructor(private apiService: ApiService) {
  }

  list() {
    return this.apiService.get('/races');
  }

}
```

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

```
import { Injectable } from '@angular/core';
import { ApiService } from './api.service';

@Injectable()
export class RaceService {

  constructor(private apiService: ApiService) {
  }

  list() {
    return this.apiService.get('/races');
  }

}
```

As we are using the `ApiService`, we need to "register" it, so as to make it available for injection.

An easy way to do this is to use the `providers` attribute of the `@NgModule` decorator we saw earlier:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { PonyRacerAppComponent } from './app.component';
import { ApiService } from './services/api.service';

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

}
```

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

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { PonyRacerAppComponent } from './app.component';
import { RaceService } from './services/race.service';
import { ApiService } from './services/api.service';

@NgModule({
  imports: [BrowserModule],
  declarations: [PonyRacerAppComponent],
  providers: [
    RaceService,
    ApiService
  ],
  bootstrap: [PonyRacerAppComponent]
})
export class AppModule {

}
```

And we're done!

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

```
import { Component } from '@angular/core';
import { RaceService } from './services/race.service';

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

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

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

}
```

As our lazy colleague returned an empty array in the `get` method of `ApiService`, you should get nothing if you try to call the `list()` method.

Maybe we can do something about it…

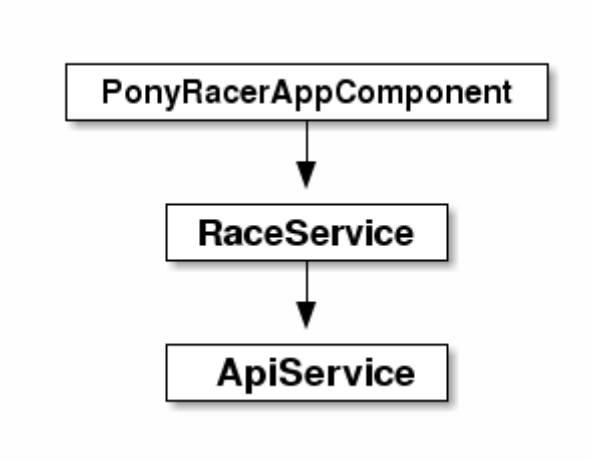
9.3. Easy to configure

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:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { PonyRacerAppComponent } from './app.component';
import { RaceService } from './services/race.service';
import { ApiService } from './services/api.service';

@NgModule({
  imports: [BrowserModule],
  declarations: [PonyRacerAppComponent],
  providers: [
    RaceService,
    ApiService
  ],
  bootstrap: [PonyRacerAppComponent]
})
export class AppModule {

}
```

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 { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { PonyRacerAppComponent } from './app.component';
import { RaceService } from './services/race.service';
import { ApiService } from './services/api.service';

@NgModule({
  imports: [BrowserModule],
  declarations: [PonyRacerAppComponent],
  bootstrap: [PonyRacerAppComponent],
  providers: [
    { provide: RaceService, useClass: RaceService },
    { provide: ApiService, useClass: ApiService }
  ]
})
export class AppModule {

}
```

We are telling the `Injector` that we want to create a link between a token (the type `RaceService`) and the class `RaceService`. The `Injector` is a service which keeps track of the injectable components by maintaining a registry and is actually injecting them when needed. The registry is a map that associates keys, called tokens, with classes. The tokens are not necessarily strings, unlike in many dependency injection frameworks. 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:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { PonyRacerAppComponent } from './app.component';
import { RaceService } from './services/race.service';
import { ApiService } from './services/api.service';

@NgModule({
  imports: [BrowserModule],
  declarations: [PonyRacerAppComponent],
  providers: [
    RaceService,
    ApiService
  ],
  bootstrap: [PonyRacerAppComponent]
})
export class AppModule {

}
```

The token has to uniquely identify the dependency.

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

```
  // in our module
  providers: [
    ApiService,
    { provide: RaceService, useClass: RaceService },
    // let's add another provider to the same class
    // with another token
    { provide: 'RaceServiceToken', useClass: RaceService }
  ]

// let's bootstrap the module
platformBrowserDynamic().bootstrapModule(AppModule)
  .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 {apiService: ApiService}"
  console.log(inj.get('RaceServiceToken'));
  // logs "RaceService {apiService: ApiService}" 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:

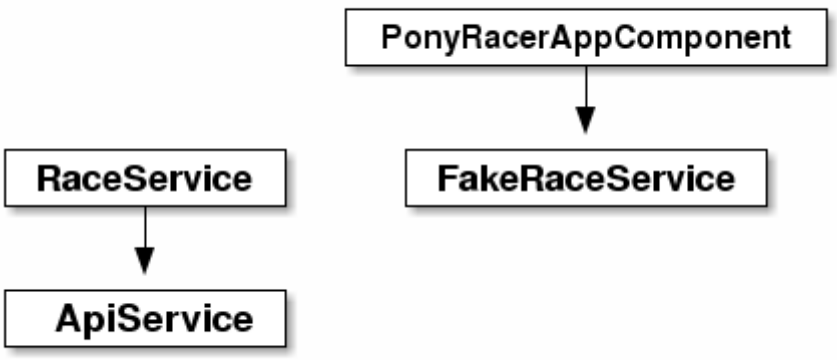
```
class FakeRaceService {
  list() {
    return [{ name: 'London' }];
  }
}
```

We can use the provider declaration to replace `RaceService` with our `FakeRaceService`:

```
// in our module
providers: [
  // we provide a fake service
  { provide: RaceService, useClass: FakeRaceService }
]
```

If you restart your app, you will see that there we have one race this time, 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`.

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

  // in our module
  providers: [
    // we provide a factory
    {
      provide: RaceService,
      useFactory: () => IS_PROD ? new RaceService(null) : 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 `ApiService` dependency instantiated! Right, if we want to make this example work, we have to pass an `ApiService` instance to the constructor call. Good news: `useFactory` can be used with another property named `deps`, where you can specify an array of dependencies:

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

  // in our module
  providers: [
    ApiService,
    // we provide a factory
    {
      provide: RaceService,
      // the apiService instance will be injected in the factory
      // so we can pass it to RaceService
      useFactory: apiService => IS_PROD ? new RaceService(apiService) : new
FakeRaceService(),
      deps: [ApiService]
    }
  ]
```

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:

```
// in our module
providers: [
  ApiService,
  { provide: RaceService, useClass: IS_PROD ? RaceService : FakeRaceService }
]
```

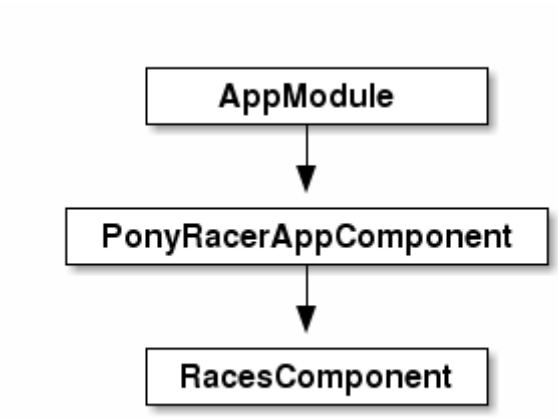
Declaring a constant for `IS_PROD` is really bothering: maybe we can use dependency injection too? I'm pushing things a bit as you can see :) You don't necessarily need to force all things in DI, but this is just to show you another provider type: `useValue`.

```
// in our module
providers: [
  ApiService,
  // we provide a factory
  { provide: 'IS_PROD', useValue: true },
  {
    provide: RaceService,
    useFactory: (IS_PROD, apiService) => IS_PROD ? new RaceService(apiService) : new
FakeRaceService(),
    deps: ['IS_PROD', ApiService]
  }
]
```

9.5. Hierarchical injectors

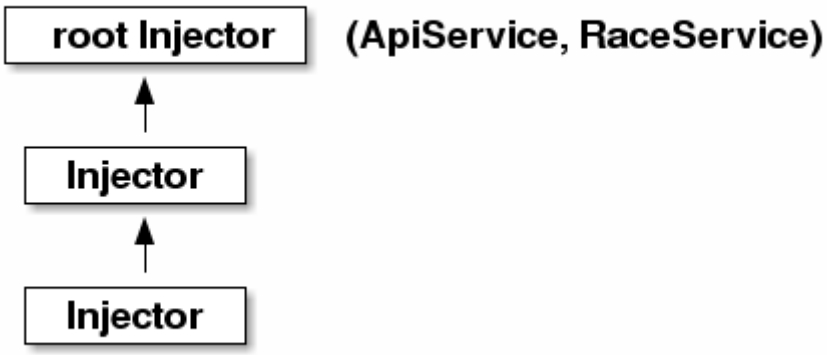
One last crucial thing to understand in Angular: there are several injectors in your app. In fact, there is one injector per component, and this injector inherits from the injector of its parent.

Let's say we have an app looking like:



We have a module `AppModule` with a root component `PonyRacerAppComponent`, with a child component `RacesComponent`.

When we bootstrap the app, we create the root injector for the module. 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 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. If 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, `ApiService` and `RaceService` can be used everywhere.
- we can declare dependencies at another level than the module. 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 `providers` attribute of `@NgModule`.

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

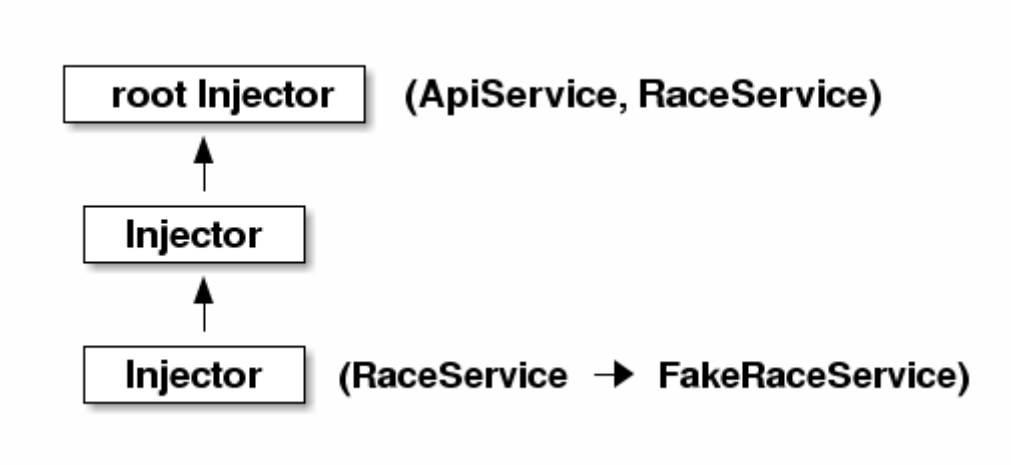
```
@Component({
  selector: 'ns-races',
  providers: [{ provide: RaceService, useClass: FakeRaceService }],
  template: `<strong>Races list: {{ list() }}</strong>`
})
export class RacesComponent {

  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

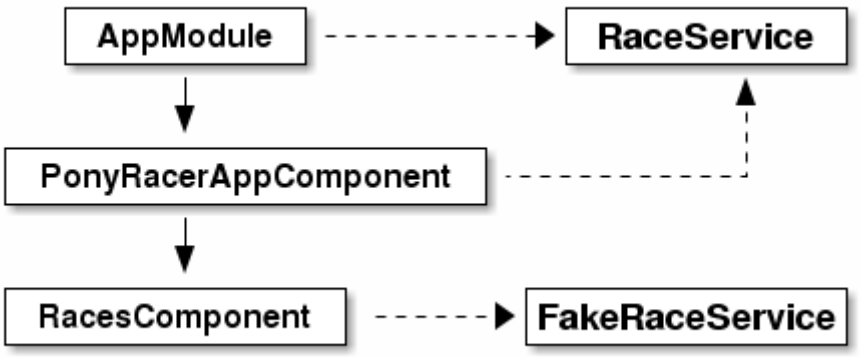
different instance of a service for a given component, or if you want to have perfectly encapsulated components that declare everything they need.

WARNING	If you declare a dependency in the module of your app and in the `providers` attribute of your component, there will be two distinct instances of this dependency created and used!

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 root module.

9.6. 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 `ApiService` service:

```
import { Injectable, Inject } from '@angular/core';
import { ApiService } from './api.service';

@Injectable()
export class RaceService {

  constructor(@Inject(ApiService) apiService) {
    this.apiService = apiService;
  }

  list() {
    return this.apiService.get('/races');
  }
}
```

To make the decorators work in this example without TypeScript, you'll have to use babel with the preset `es2015` and the plugins `angular2-annotations` and `syntax-decorators`.

Chapter 10. Services

Angular 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.

10.1. Title service

The core framework provides very few services, and those you will use in your apps are even scarcer: actually there are just two 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 '@angular/core';
import { Title } from '@angular/platform-browser';

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

  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!

10.2. Meta service

The other service is slightly similar: it allows to get or update the "meta" values of the page.

```
import { Component } from '@angular/core';
import { Meta } from '@angular/platform-browser';

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

  constructor(meta: Meta) {
    meta.addTag({ name: 'author', content: 'Ninja Squad' });
  }

}
```

10.3. 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 '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable()
export class RacesServiceWithHttp {

  constructor(private http: HttpClient) {
  }

  list() {
    return this.http.get('/api/races');
  }

}
```

Then you have to add it to the `providers` attribute of a component, or, more realistically, to the `providers` of your root module:

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

PRACTICE

Try our exercise Race service 🏇! It's free and part of our Pro Pack, where you'll learn how to build a complete application step by step. This exercise lets you build your first service!

Chapter 11. Pipes

11.1. 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.

11.2. 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 '@angular/core';
// you need to import the pipe you want to use
import { JsonPipe } from '@angular/common';

@Component({
  selector: 'ns-ponies',
  template: `<p>{{ poniesAsJson }}</p>`
})
export class PoniesComponent {
  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);
  }
}
```

But beware: the pipe must be added to the `providers` of your `@NgModule` (or `@Component`) in order to use it that way.

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

11.3. 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 '@angular/core';

@Component({
  selector: 'ns-ponies',
  template: `<div *ngFor="let pony of ponies | slice:0:2">{{ pony.name }}</div>`
})
export class PoniesComponent {
  ponies: Array<any> = [
    { name: 'Rainbow Dash' },
    { name: 'Pinkie Pie' },
    { name: 'Fluttershy' }
  ];
}
```

The component will create only two `div` elements 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 '@angular/core';

@Component({
  selector: 'ns-ponies',
  template: `<div *ngFor="let pony of ponies | slice:0:size">{{ pony.name }}</div>`
})
export class PoniesComponent {
  size = 2;
  ponies = [
    { name: 'Rainbow Dash' },
    { name: 'Pinkie Pie' },
    { name: 'Fluttershy' }
  ];
}
```

You can use this to create a dynamic display where your user chooses how many elements she/he wants to see.

Note that it's also possible to store the result of the `slice` in a variable, using the `as` syntax introduced in 4.0:

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

@Component({
  selector: 'ns-ponies',
  template: `<div *ngFor="let pony of ponies | slice:0:2 as total; index as i">
    {{ i+1 }}/{{ total.length }}: {{ pony.name }}
  </div>`
})
export class PoniesComponent {
  ponies: Array<any> = [
    { name: 'Rainbow Dash' },
    { name: 'Pinkie Pie' },
    { name: 'Fluttershy' }
  ];
}
```

11.4. 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' -->
```

11.5. 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' -->
```

11.6. titlecase

Angular 4 introduced a new `titlecase` pipe. It capitalizes the first letter of all words:

```
<p>{{ 'ninja squad' | titlecase }}</p>
<!-- will display 'Ninja Squad' -->
```

11.7. 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' -->
<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` and `currency`) 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 it forces us to use a polyfill for this API in some browsers (like Safari/iOS).

11.8. 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%' -->
```

11.9. 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 'EUR10.60' -->

<p>{{ 10.6 | currency:'USD':true }}</p>
<!-- will display '$10.60' -->

<p>{{ 10.6 | currency:'USD':true:'.3' }}</p>
<!-- will display '$10.600' -->
```

11.10. 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 'dd/MM/yyyy', 'MM-yy' or one of the predefined symbolic names available like 'short', 'longDate', etc.:

```
<p>{{ birthday | date:'dd/MM/yyyy' }}</p>
<!-- will display '16/07/1986' -->

<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:'HH:mm' }}</p>
<!-- will display '15:30' -->

<p>{{ birthday | date:'shortTime' }}</p>
<!-- will display '3:30 PM' -->
```

NOTE To learn more about internationalization in general, and, in particular, about the way you can set the language used to format numbers and dates, you can refer to the Internationalization chapter .

11.11. 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 '@angular/core';

@Component({
  selector: 'ns-greeting',
  template: `<div>{{ asyncGreeting | async }}</div>`
})
export class GreetingComponent {
  asyncGreeting = 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).

And to avoid multiple subscriptions to your Observable or calling your promise multiple times, you can store the result of the call with `as` (since 4.0):

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

@Component({
  selector: 'ns-user',
  template: `<div *ngIf="asyncUser | async as user">{{ user.name }}</div>`
})
export class UserComponent {
  asyncUser = new Promise(resolve => {
    // after 1 second, the promise will resolve
    window.setTimeout(() => resolve({ name: 'Cédric' }), 1000);
  });
}
```

11.12. 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!

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 '@angular/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: {
  '@angular': 'node_modules/@angular',
  'rxjs': 'node_modules/rxjs',
  'moment': 'node_modules/moment/moment'
}
```

The typings for Moment are already included in the NPM dependency, so the compiler should be happy without us doing anything.

```
import { PipeTransform, Pipe } from '@angular/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 '@angular/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, the last thing you need to do is to add the pipe to the `declarations` of your `@NgModule`.

```
@NgModule({
  imports: [BrowserModule],
  declarations: [PonyRacerAppComponent, RacesComponent, FromNowPipe],
  bootstrap: [PonyRacerAppComponent]
})
export class AppModule {
}
```

PRACTICE

Try our exercise Pipes 🐴! It's free and part of our Pro Pack, where you'll learn how to build a complete application step by step. This exercise lets you use your first pipe. Later, exercise Custom pipe with Moment.js 🦄 will make you build an awesome custom pipe!

Chapter 12. Reactive Programming

12.1. 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?

Well, Angular 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.

12.2. 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 a

future version, 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 relies on and lets us use.

So let's have a look.

12.3. RxJS

Every observable, just like every array, can be transformed using functions you may have already encountered:

- `take(n)` will pick the first n events (e.g. the first five).
- `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.from([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:

```
const 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:

```
const 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(x => {
    if (x % 2 === 1) {
      throw new Error('something went wrong');
    } else {
      return x;
    }
  })
  .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.

12.4. Reactive programming in Angular

Angular 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.

```
const 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.

```
const emitter = new EventEmitter();

const subscription = emitter.subscribe(
  value => console.log(value),
  error => console.log(error),
  () => console.log('done')
);

emitter.emit('hello');
subscription.unsubscribe(); // unsubscribe
emitter.emit('there');

// logs "hello" only
```

Now that we know a little bit more about reactive programming, and the `EventEmitter`, let's see how Angular uses it.

PRACTICE

Try our exercise Observables 🐴! It's free and part of our Pro Pack, where you'll learn how to build a complete application step by step. In this exercise, you will transform the `RaceService` to make it reactive!

Chapter 13. Building components and directives

13.1. 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!

13.2. 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.

13.2.1. 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 `TestComponent`:

```
@Component({
  selector: 'ns-test',
  template: '<div doNothing>Click me</div>'
})
export class TestComponent {

}
```

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-`, `let-` or `ref-`: 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.

13.2.2. 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 SimpleTextWithSetterDirective {

  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, and the official guide style also recommends to use it, so a lot of examples will use it from now on.

The examples above can be rewritten as:

```
@Directive({
  selector: '[loggable]'
})
export class InputDecoratorDirective {

  @Input('logText') text: string;
}
```

or, if the property and the binding have the same name:

```
@Directive({
  selector: '[loggable]'
})
export class SameNameInputDecoratorDirective {

  @Input() logText: string;
}
```

This will work but having a field and a setter with the same name will make the TypeScript compiler unhappy. One way to fix it if you need a setter (you don't always need it) is to add the `@Input` decorator directly on the setter.

```
@Directive({
  selector: '[loggable]'
})
export class InputDecoratorOnSetterDirective {

  @Input('logText')
  set text(value) {
    console.log(value);
  }
}
```

or, if the setter and the binding have the same name:

```
@Directive({
  selector: '[loggable]'
})
export class SameNameInputDecoratorOnSetterDirective {

  @Input()
  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:

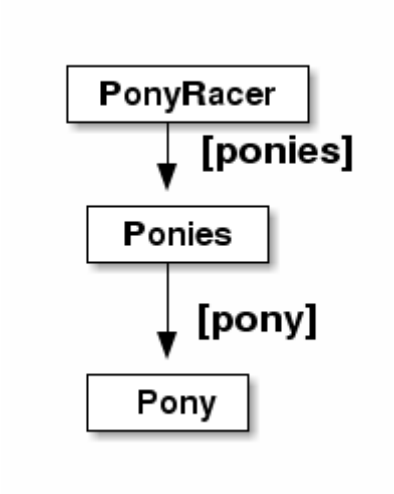
```
@Component({
  selector: 'ns-pony',
  template: `<div>{{ pony.name }}</div>`
})
export class PonyComponent {
  @Input() pony: Pony;
}

@Component({
  selector: 'ns-ponies',
  template: `<div>
    <h2>Ponies</h2>
    // the pony is handed to PonyComponent via [pony]="currentPony"
    <ns-pony *ngFor="let currentPony of ponies" [pony]="currentPony"></ns-pony>
  </div>`
})
export class PoniesComponent {
  ponies: Array<Pony> = [
    { id: 1, name: 'Rainbow Dash' },
    { id: 2, name: 'Pinkie Pie' }
  ];
}
```

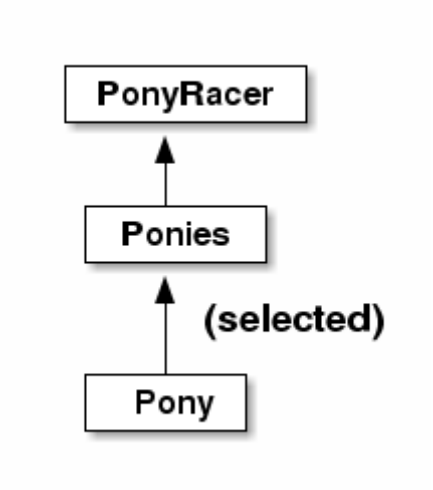
Ok, now what about passing data up? We can't use property to pass data from `PonyComponent` to `PoniesComponent`. But we can use events!

13.2.3. 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, 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. And, like the inputs, it's better to use the `@Output()` decorator.

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: 'ns-pony',
  // the method `selectPony()` will be called on click
  template: `<div (click)="selectPony()">{{ pony.name }}</div>`
})
export class SelectablePonyComponent {

  @Input() pony: Pony;

  // we declare the custom event as an output,
  // the EventEmitter is used to emit the event
  @Output() ponySelected = 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:

```
<ns-pony [pony]="pony" (ponySelected)="betOnPony($event)"></ns-pony>
```

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 parent 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: 'ns-pony',
  template: `<div (click)="selectPony()">{{ pony.name }}</div>`
})
export class OtherSelectablePonyComponent {

  @Input() pony: Pony;
  // the emitter is called `emitter`
  // and the event `ponySelected`
  @Output('ponySelected') emitter = new EventEmitter<Pony>();

  selectPony() {
    this.emitter.emit(this.pony);
  }
}
```

13.2.4. 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 UndefinedInputsDirective {

  @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 change). 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 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: SimpleChanges) {
    const ponyValue = changes['pony'];
    console.log(`changed from ${ponyValue.previousValue} to ${ponyValue.currentValue}
`);
    console.log(`is it the first change? ${ponyValue.isFirstChange()}`);
  }

}
```

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, as well as a method `isFirstChange()` to know if it is... the first change!

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]'
})
export class SetterDirective {

  private ponyModel: string;

  @Input()
  set pony(newPony) {
    console.log(`changed from ${this.ponyModel} to ${newPony}`);
    this.ponyModel = 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...

13.2.5. 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) {
    const ponies = poniesService.list();
    console.log(`ponies are: ${ponies}`);
  }
}
```

13.3. 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.

13.3.1. 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.

13.3.2. 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 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: 'ns-templated-pony',
  templateUrl: 'components/pony/templated-pony.html'
})
export class TemplatedPonyComponent {
  @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 can then use this value and build the correct relative URL. Your template URL can now look like:

```
@Component({
  selector: 'ns-templated-pony',
  templateUrl: 'templated-pony.html',
  moduleId: module.id
})
export class ModuleIdPonyComponent {
  @Input() pony: any;
}
```

And you can locate your template in the same directory as your component!

Even better: if you are using Webpack, with a bit of configuration (already done for you if you are using Angular CLI), you can even remove the `module.id` and use a relative path directly. Webpack will be able to figure out the complete URL for you!

13.3.3. 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: 'ns-styled-pony',
  template: '<div class="pony">{{ pony.name }}</div>',
  styles: ['.pony{ color: red; }']
})
export class StyledPonyComponent {
  @Input() pony: any;
}
```

13.3.4. Declarations

Remember that you have to declare every directive and component you are using in the `declarations` of your `@NgModule`. 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 to declare the directive** and **using the wrong selector**. If you don't understand why nothing happens, look out for these!

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!

PRACTICE	Try our exercises Race detail 🦄 and Pony component 🦄! These exercises will guide you to build two more advanced components, with inputs and outputs.

Chapter 14. Styling components and encapsulation

Let's stop to talk about styles and CSS for a minute. I know right? Why talking about freaking CSS?

Well because Angular is doing a lot of things for us behind the scenes.

As a Web developer, you often add CSS classes to elements. And the essence of CSS is that it will *cascade*. That's sometimes what you want (to change the font everywhere in your app for example), or sometimes not. Imagine you want to add a style on a selected element in a list: you will usually use a very narrow CSS selector in your CSS, like `li.selected`. Or an even narrower one, using conventions like BEM, because you just want to style the selected element in a specific part of your app.

That's where Angular can be useful. The styles you define in a component (either with the `styles` attribute, or in a dedicated CSS file for the component with `styleUrls`), are scoped by Angular to this component and only this one. That's called style encapsulation. How does it achieve this?

It starts with you writing some style. Then it depends on the strategy you select for the attribute `encapsulation` of the component decorator. This attribute can have three different values:

- `ViewEncapsulation.Emulated`, which is the default one
- `ViewEncapsulation.Native`, which relies on Shadow DOM
- `ViewEncapsulation.None`, which means you don't want encapsulation

Each value will induce a different behavior of course, so let's have a look. We'll take a component you're starting to know well, i.e. our `PonyComponent`. This is a really simple version of the component, only displaying the pony's name in a `div`. For the purpose of the example, we add a CSS class `red` to this `div`:

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

@Component({
  selector: 'ns-pony',
  template: `<div class="red">{{ name }}</div>`,
  styles: [`.red {color: red;}`],
  // that's the same as the default mode
  encapsulation: ViewEncapsulation.Emulated
})
export class PonyComponent {

  name = 'Rainbow Dash';

}
```

This class is then used in the styles of the component:

```
.red {
  color: red;
}
```

As you can see, we want to display the pony's name in a red font.

14.1. Native strategy

If you use the `Native` option, you're telling Angular to use the Shadow DOM of your browser to take care of the encapsulation. The Shadow DOM is a part of the rather new Web Component specification. This specification allows to create elements in a special DOM, which is perfectly encapsulated. With this strategy, if we look at the generated DOM with our browser's inspector, we'll see:

```
<ns-pony>
  #shadow-root (open)
    <style>.red {color: red}</style>
    <div class="red">Rainbow Dash</div>
</ns-pony>
```

You can spot the `#shadow-root (open)` that Chrome will display in the inspector: that's because our component has been included in a Shadow DOM element! And we can also see that the style was added at the top of our component's content.

With the `Native` strategy, you are sure that your component's styles are not "bleeding" into your child components. If we have another component inside the `PonyComponent`, it can also define its own `red` CSS class with a different style: you are sure that the correct one will be used, with no interaction between each others!

But remember, Shadow DOM is a rather new specification, so it's not available in every browser. You can check the availability on the awesome website caniuse.com. So be careful when you use it in your apps!

14.2. Emulated strategy

As said earlier, this is the default strategy. And the reason is really simple: it emulates (hence the name) the `Native` strategy, but without using the Shadow DOM. So it's safe to use everywhere, and will have the same behavior.

To achieve that, Angular will take the CSS defined for the component, and inline it inside the `<head>` element of the page (and not in each component as we saw for the `Native` strategy). But before inlining it, it's going to rewrite the CSS selector, to append a unique attribute identifier. This unique attribute is then added to all the elements of our component's template! That way the style will only apply to our component. The same example, would now give:

```
<html>
  <head>
    <style>.red[_ngcontent-dvb-3] {color: red}</style>
  </head>
  <body>
    ...
    <ns-pony _ngcontent-dvb-2="" _nghost-dvb-3="">
      <div _ngcontent-dvb-3="" class="red">Rainbow Dash</div>
    </ns-pony>
  </body>
</html>
```

The `red` class selector has been rewritten to `.red[_ngcontent-dvb-3]`, so it will only apply on elements that have both the class `red` and the attribute `_ngcontent-dvb-3`. You can see that this attribute has also been added to our `div` automatically, so that works perfectly. The `<ns-pony>` element also has a few attributes: `_ngcontent-dvb-2` which is the unique identifier generated for its parent, and `_nghost-dvb-3` which is a unique identifier for the host element itself. Yes, we can also add styles that apply on the host element, as we'll see shortly.

14.3. None strategy

This strategy is not doing any encapsulation. The styles will be inlined at the top of the page (as for the `Emulated` strategy), but not rewritten. They then behave like "normal" styles, cascading into children.

14.4. Styling the host

A special CSS selector exists to style only the host element. It is called `:host`, and it comes from the Web Component specification:

```
:host {
  display: block;
}
```

It will be kept as is for the `Native` strategy and rewritten into `[_nghost-xxx]` if you use `Emulated`.

To conclude, you don't have to do much to have perfectly encapsulated styles, because the `Emulated` strategy takes care of this business for us. You can switch the strategy to use the `Native` one if you target only specific browsers, or `None` if you don't want to encapsulate styles. This strategy can be tweaked per component, or globally for your whole app in the root module.

Chapter 15. Testing your app

15.1. 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 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.

15.2. 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 testing is **isolation**: 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, so we are going to use it!

15.2.1. 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', () => {
    const 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', () => {
    const 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 to the mix.

15.2.2. Using dependency injection

Let's say I have an Angular application with a simple service like `RaceService`, containing a method returning a hard-coded races list.

```
export class RaceService {
  list() {
    const race1 = new Race('London');
    const race2 = new Race('Lyon');
    return [race1, race2];
  }
}
```

Let's write a test for this.

```
describe('RaceService', () => {
  it('should return races when list() is called', () => {
    const 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 in `TestBed` called `get`.

This method allows to get a specific dependency from the injector inside a test function.

Let's go back to our example, using `TestBed.get` this time:

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

describe('RaceService', () => {
  it('should return races when list() is called', () => {
    const raceService = TestBed.get(RaceService);
    expect(raceService.list().length).toBe(2);
  });
});
```

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 root module when we start the app.

The `TestBed` class is here to help. Its `configureTestingModule` method allows to declare what can be injected in the test, by creating a test module containing only what we need. 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 is called in the `beforeEach` Jasmine method, and takes a module configuration, really close to what you can pass to the `@NgModule` decorator. The `providers` attribute takes an array of dependencies that will become available to injection.

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

describe('RaceService', () => {

  beforeEach(() => TestBed.configureTestingModule({
    providers: [RaceService]
  }));

  it('should return races when list() is called', () => {
    const raceService = TestBed.get(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 `providers` attribute, 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 `TestBed.get` in a `beforeEach`, so let's do it:

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

describe('RaceService', () => {
  let service: RaceService;

  beforeEach(() => TestBed.configureTestingModule({
    providers: [RaceService]
  }));

  beforeEach(() => service = TestBed.get(RaceService));

  it('should return races when list() is called', () => {
    expect(service.list().length).toBe(2);
  });
});
```

We moved the `TestBed.get` logic in a `beforeEach` and now our test is pretty clean. Be careful to always call `TestBed.configureTestingModule` (which sets up the injector), before actually using the injector with `TestBed.get` 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 { async, TestBed } from '@angular/core/testing';

describe('RaceService', () => {
  let service: RaceService;

  beforeEach(() => TestBed.configureTestingModule({
    providers: [RaceService]
  }));

  beforeEach(() => service = TestBed.get(RaceService));

  it('should return a promise of 2 races', async(() => {
    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 wrap the test with the `async()` function. And this method is really smart: it keeps track of the asynchronous calls made in the test and waits for them to resolve.

Angular 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 application runs in a zone, and that's how the framework knows it has to refresh the DOM when an asynchronous action is done.

This concept is also used in the tests if your test uses `async()`: the 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, using `fakeAsync()` and `tick()` but that's for a more advanced chapter.

15.3. Fake dependencies

Being able to declare the dependencies with the test module 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 [{ name: 'Lyon' }, { name: 'London' }];
  }
}
```

to `RaceService` in our test, using `provide`:

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

describe('RaceService', () => {

  beforeEach(() => TestBed.configureTestingModule({
    providers: [
      { provide: LocalStorageService, useClass: FakeLocalStorage },
      RaceService
    ]
  }));

  it('should return 2 races from localStorage', () => {
    const service = TestBed.get(RaceService);
    const 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 { TestBed } from '@angular/core/testing';

describe('RaceService', () => {

  const localStorage = jasmine.createSpyObj('LocalStorageService', ['get']);

  beforeEach(() => TestBed.configureTestingModule({
    providers: [
      { provide: LocalStorageService, useValue: localStorage },
      RaceService
    ]
  }));

  it('should return 2 races from localStorage', () => {
    localStorage.get.and.returnValue([{ name: 'Lyon' }, { name: 'London' }]);

    const service = TestBed.get(RaceService);
    const races = service.list();

    expect(races.length).toBe(2);
    expect(localStorage.get).toHaveBeenCalledWith('races');
  });
})
;
```

15.4. 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 `PonyComponent` component? It takes a pony as an input and emits an event `ponyClicked` when the component is clicked.

```
@Component({
  selector: 'ns-pony',
  template: `<img [src]="'/images/pony-' + pony.color.toLowerCase() + '.png'"
(click)="clickOnPony()">`
})
export class PonyComponent {

  @Input() pony: PonyModel;
  @Output() ponyClicked = new EventEmitter<PonyModel>();

  clickOnPony() {
    this.ponyClicked.emit(this.pony);
  }

}
```

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, we can also use `TestBed`. This class comes with a utility method, named `createComponent`, to create a component. The method returns a `ComponentFixture`, a representation of our component. Note that to create a component, it must be known from the test module, so we need to add it to the `declarations` attribute:

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

import { PonyComponent } from './pony_cmp';

describe('PonyComponent', () => {

  it('should have an image', () => {
    TestBed.configureTestingModule({
      declarations: [PonyComponent]
    });
    const fixture = TestBed.createComponent(PonyComponent);
    // given a component instance with a pony input initialized
    const ponyComponent = fixture.componentInstance;
    ponyComponent.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
    const 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 provide 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.

We can also test if the component really emits an event:

```
it('should emit an event on click', () => {
  TestBed.configureTestingModule({
    declarations: [PonyComponent]
  });
  const fixture = TestBed.createComponent(PonyComponent);

  // given a pony
  const ponyComponent = fixture.componentInstance;
  ponyComponent.pony = { name: 'Rainbow Dash', color: 'BLUE' };

  // we fake the event emitter with a spy
  spyOn(ponyComponent.ponyClicked, 'emit');

  // when we click on the pony
  const element = fixture.nativeElement;
  const image = element.querySelector('img');
  image.dispatchEvent(new Event('click'));

  // and we trigger the change detection
  fixture.detectChanges();

  // then the event emitter should have fired an event
  expect(ponyComponent.ponyClicked.emit).toHaveBeenCalled();
});
```

Let's have a look at another component:

```
@Component({
  selector: 'ns-race',
  template: `<div>
    <h1>{{ race.name }}</h1>
    <ns-pony *ngFor="let currentPony of race.ponies" [pony]="currentPony"></ns-pony>
  </div>`
})
export class RaceComponent {

  @Input() race: any;

}
```

and its test:

```
describe('RaceComponent', () => {

  let fixture: ComponentFixture<RaceComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [RaceComponent, PonyComponent]
    });
    fixture = TestBed.createComponent(RaceComponent);
  });

  it('should have a name and a list of ponies', () => {
    // given a component instance with a race input initialized
    const raceComponent = fixture.componentInstance;
    raceComponent.race = { name: 'London', ponies: [{ name: 'Rainbow Dash', color:
'BLUE' }] };

    // when we trigger the change detection
    fixture.detectChanges();

    // then we should have a title with the race name
    const element = fixture.nativeElement;
    expect(element.querySelector('h1').textContent).toBe('London');

    // and a list of ponies
    const ponies = fixture.debugElement.queryAll(By.directive(PonyComponent));
    expect(ponies.length).toBe(1);
    // we can check if the pony is correctly initialized
    const rainbowDash = ponies[0].componentInstance.pony;
    expect(rainbowDash.name).toBe('Rainbow Dash');
  });
});
```

Here we query all the directives of type `PonyComponent` and test if the first pony is correctly initialized.

You can get the components inside your component with `children` 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 `PonyComponent`. 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).

15.5. Testing with fake templates, providers...

When testing a component, we sometimes want to create a parent component that uses it. And if there are several use cases, we'll have to create several parent components just to try different inputs for example.

Hopefully, when we are in a test, we can modify any component to reuse it in different tests, by overriding its template.

To do this, the `TestBed` gives an `overrideComponent()` method, to call before the `createComponent()` one:

```
describe('RaceComponent', () => {

  let fixture: ComponentFixture<RaceComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [RaceComponent, PonyComponent]
    });
    TestBed.overrideComponent(RaceComponent, { set: { template: '<h2>{{ race.name
}}</h2>' } });
    fixture = TestBed.createComponent(RaceComponent);
  });

  it('should have a name', () => {
    // given a component instance with a race input initialized
    const raceComponent = fixture.componentInstance;
    raceComponent.race = { name: 'London' };

    // when we trigger the change detection
    fixture.detectChanges();

    // then we should have a name
    const element = fixture.nativeElement;
    expect(element.querySelector('h2').textContent).toBe('London');
  });
});
```

As you can see, the method takes two arguments:

- the component you want to override
- the metadata you want to set, add or remove (for example here we set 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).

`template` is not the only metadata available, you can also use:

- `providers` to replace the dependencies of a component
- `styles` to replace the styles used in the template of a component
- or any property you can set in the `@Component` decorator…

As replacing the template is the most common use-case, Angular 4 introduced an `overrideTemplate()` method:

```
describe('RaceComponent', () => {

  let fixture: ComponentFixture<RaceComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [RaceComponent, PonyComponent]
    });
    TestBed.overrideTemplate(RaceComponent, '<h2>{{ race.name }}</h2>');
    fixture = TestBed.createComponent(RaceComponent);
  });

  it('should have a name', () => {
    // given a component instance with a race input initialized
    const raceComponent = fixture.componentInstance;
    raceComponent.race = { name: 'London' };

    // when we trigger the change detection
    fixture.detectChanges();

    // then we should have a name
    const element = fixture.nativeElement;
    expect(element.querySelector('h2').textContent).toBe('London');
  });
});
```

Now you're ready to test your app!

15.6. 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 as 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!

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!

PRACTICE	All our Pro Pack exercises come with unit tests! If you want to learn more, we strongly encourage you to take a look at them: we tested every possible part of the application (100% code coverage)! In the end you'll have dozens of test examples, which you can use in your own projects.

Chapter 16. Send and receive data through HTTP

WARNING	This chapter uses the new `HttpClientModule` introduced in Angular 4.3 in the `@angular/common/http` package, which is a complete rewrite of the `HttpModule` that existed before. This chapter does *not* talk about the old `HttpModule` that was used previously from the `@angular/http` package.

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 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. 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 and its zones!).

But if you feel comfortable with the framework, you will use a small module called `HttpClientModule`, 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 `@angular/common/http` package.

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 `HttpClient` 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. To do so, import the `HttpModule` into the root module:

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

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

Once this is done, you can inject the `HttpClient` service wherever you need it:

```
@Component({
  selector: 'ns-races',
  template: `<h1>Races</h1>`
})
export class RacesComponent {

  constructor(private http: HttpClient) {
  }

}
```

By default, the `HttpClient` service will do AJAX request using `XMLHttpRequest`.

It offers several methods, matching the most common HTTP verbs:

- `get`
- `post`
- `put`
- `delete`
- `patch`
- `head`
- `jsonp`

If you used the `$http` service in AngularJS 1.x, you might remember that it heavily relied on Promises. In Angular, however, all these methods return an `Observable` object.

A few advantages come with the use of Observables for `HttpClient` 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 `HttpClient` 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 body is the most interesting part, and it is directly emitted by the Observable:

```
http.get(`${baseUrl}/api/races`)
  .subscribe((response: Array<RaceModel>) => {
    console.log(response);
    // logs the array of races
  });
```

Of course, you can also have access to the full HTTP response. The object returned is then an `HttpResponse` object, with a few fields like the `status` code, `headers`, etc.

```
http.get(`${baseUrl}/api/races`, { observe: 'response' })
  .subscribe((response: HttpResponse<Array<RaceModel>>) => {
    console.log(response.status); // logs 200
    console.log(response.headers); // logs []
  });
```

The observable will throw an error if the response status is different from 2xx or 3xx, and the error is then of type `HttpErrorResponse`.

Sending data is fairly easy too. Just call the `post()` method, with the URL and the object to post:

```
// you currently need to stringify the object you send
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

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()
  .subscribe(races => {
    // store the array of the races in the component
    this.races = races;
  });
```

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)
  .subscribe(races => {
    // store the array of the races in the component
    this.races = races;
  });
```

16.3. Advanced options

Of course, you can tune your requests more finely. Every method takes an `options` 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.

`params` represents the URL search parameters (also known as the query string) to add to the URL.

```
const params = new HttpParams()
  .set('sort', 'ascending')
  .set('page', '1');

http.get(`${baseUrl}/api/races`, { params })
// will call the URL ${baseUrl}/api/races?sort=ascending&page=1
  .subscribe(response => {
    // will return the races sorted
    this.races = response;
  });
```

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:

```
const headers = new HttpHeaders()
  .set('Authorization', `Bearer ${token}`);

http.get(`${baseUrl}/api/races`, { headers })
  .subscribe(response => {
    // will return the races visible for the authenticated user
    this.races = response;
  });
```

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 classic HTTP methods, the `HttpClient` offers a `jsonp` method. All you have to do is specify the URL of the service you want to call, and add the name of the callback you want.

In the following example, we are fetching all the public repos from our Github organization using JSONP.

```
http.jsonp('https://api.github.com/orgs/Ninja-Squad/repos', 'callback')
// extract data
  .map((res: { data: Array<any> }) => res.data)
  .subscribe(response => {
    // will return the public repos of Ninja-Squad
    this.repos = response;
  });
```

16.5. Interceptors

One of the reasons of the Http module rewrite was the introduction of interceptors. Interceptors are interesting when you want to… intercept requests or responses in your application.

For example, if you want to intercept every request to add a specific header to some of them, you can now write an interceptor like this one:

```
@Injectable()
export class GithubAPIInterceptor implements HttpInterceptor {

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    // if it is a Github API request
    if (req.url.includes('api.github.com')) {
      // we need to add an OAUTH token as a header to access the Github API
      const clone = req.clone({ setHeaders: { 'Authorization': `token ${OAUTH_TOKEN}`
} });
      return next.handle(clone);
    }
    // if it's not a Github API request, we just handle it to the next handler
    return next.handle(req);
  }

}
```

Note that you have to clone the request to update it (requests are immutable).

Then add your interceptor to the `HTTP_INTERCEPTORS` array via dependency injection:

```
providers: [
  { provide: HTTP_INTERCEPTORS, useClass: GithubAPIInterceptor, multi: true }
]
```

Now every request will go through the interceptor, and receive the custom header if needed (here the requests to the Github API).

You can also intercept the response, which can be handy to handle errors in a generic way:

```
@Injectable()
export class ErrorHandlerInterceptor implements HttpInterceptor {

  constructor(private router: Router, private errorHandler: ErrorHandler) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    return next.handle(req)
    // we catch the error
      .catch((errorResponse: HttpErrorResponse) => {
        // if the status is Unauthorized
        if (errorResponse.status === 401) {
          // we redirect to login
          this.router.navigateByUrl('/login');
        } else {
          // else we notify the user
          this.errorHandler.handle(errorResponse);
        }
        return Observable.throw(errorResponse);
      });
  }

}
```

16.6. 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: HttpClient) {
  }

  list(): Observable<Array<RaceModel>> {
    return this.http.get<Array<RaceModel>>('/api/races');
  }
}
```

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 `HttpClient` service with a fake implementation by importing the `HttpClientTestingModule`. We can then use a class provided by the framework called `HttpTestingController` to fake the HTTP responses.

And you can also add a few assertions on the underlying HTTP request:

```
import { async, TestBed } from '@angular/core/testing';
import { HttpClientTestingModule, HttpTestingController } from
'@angular/common/http/testing';
describe('RaceService', () => {

  let raceService: RaceService;
  let http: HttpTestingController;

  beforeEach(() => TestBed.configureTestingModule({
    imports: [HttpClientTestingModule],
    providers: [RaceService]
  }));

  beforeEach(() => {
    raceService = TestBed.get(RaceService);
    http = TestBed.get(HttpTestingController);
  });

  it('should return an Observable of 2 races', async(() => {
    // fake response
    const hardcodedRaces = [{ name: 'London' }, { name: 'Lyon' }];

    // call the service
    let actualRaces = [];
    raceService.list().subscribe(races => actualRaces = races);

    // check that the underlying HTTP request was correct
    http.expectOne('/api/races')
    // return the fake response when we receive a request
      .flush(hardcodedRaces);

    // check that the returned array is deserialized as expected
    expect(actualRaces.length).toBe(2);
  }));
});
```

And we're done!

PRACTICE

Try our exercise HTTP 🦄! We prepared a full REST API, ready for you to use. Let's fetch some races using the `HttpClient` service. Later you'll learn how to call a secured API with an authentication mechanism and interceptors in exercises HTTP with authentication 🦄, Bet on a pony 🦄 and Cancel a bet 🦄. Slightly related, we'll also use WebSockets 🦄.

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 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. It will execute all this without refreshing the page and without triggering a new request to our backend server: this is the whole point of having a Single Page Application.

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.

The team behind Angular decided to bridge the gap and wrote a new module called `RouterModule`. This module will hopefully fulfill all our needs!

Some new features are really interesting. So let's go!

17.1. En route

Lets's start using the router. It is an optional module, that is thus not included in the core framework.

As we saw for the other modules, you have to include it in your root module if you want to use it. But for that, we need a configuration to define the mapping between URLs and components. We can do this with a dedicated file, generally named like `app.routes.ts`, and containing an array representing the configuration:

```
import { Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { RacesComponent } from './races/races.component';

export const ROUTES: Routes = [
  { path: '', component: HomeComponent },
  { path: 'races', component: RacesComponent }
];
```

Then we need to import the router module in our root module, initialized with the proper configuration:

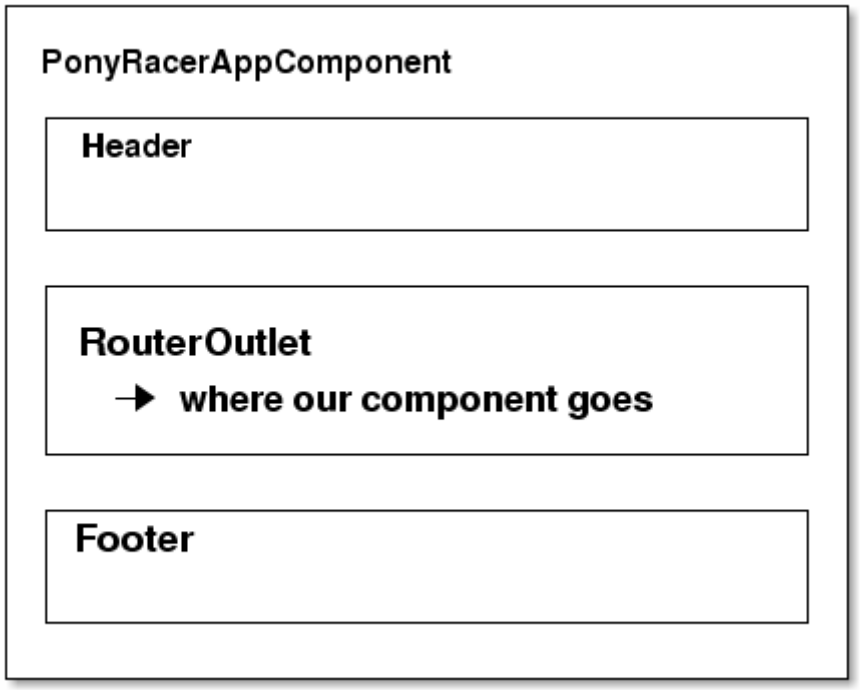
```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { RouterModule } from '@angular/router';
import { ROUTES } from './app.routes';
import { HomeComponent } from './home/home.component';
import { RacesComponent } from './races/races.component';


@NgModule({
  imports: [BrowserModule, RouterModule.forRoot(ROUTES)],
  declarations: [PonyRacerAppComponent, HomeComponent, RacesComponent],
  bootstrap: [PonyRacerAppComponent]
})
export class AppModule {
}
```

NOTE	You also need to declare all the components used by the router module in the `declarations` attribute of your root module.

As you can see, the `Routes` is an array of objects, each one being a... route. A route configuration is usually a pair of properties:

- `path`: what URL will trigger the navigation
- `component`: which component will be initialized and inserted

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 `RacesComponent` 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>...</nav>
</header>
<main>
  <router-outlet></router-outlet>
  <!-- the component's template will be inserted here-->
</main>
<footer>made with &lt;3 by Ninja Squad</footer>
```

When we navigate, everything will stay (the header, main and footer here) and the component matching the current route will be inserted just after the `RouterOutlet` directive.

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. And we don't want to use "classic" links, with `<a href=""></a>`. Indeed, clicking on that link makes the browser load the page at that URL, and restart the whole Angular application. But the goal of Angular is to avoid such page reloads: we want to create a Single Page Application. Of course, there is a solution built-in.

In a template, you can insert a link with the directive `RouterLink` pointing to the path you want to go to. We can use this directive because our root module imports the `RouterModule`, making all the exported directives of `RouterModule` available to the root module. The `RouterLink` directive can receive a constant representing the path you want to go to or an array of strings, representing the

path and its params. For example in our `RacesComponent` template, if we want to navigate to the `HomeComponent`, we can imagine something like:

```
<a href="" routerLink="/">Home</a>
<!-- same as -->
<a href="" [routerLink]="['/']">Home</a>
```

At runtime, the link `href` will be computed by the router and will point to `/`.

NOTE	The leading slash in the path is necessary. If not included, `RouterLink` build the URL relatively to the current path (that can be useful with nested components, as we'll see later). Adding a slash indicates that the URL must be computed from the application base URL.

The `RouterLink` directive can be used with the `RouterLinkActive` directive which can set a CSS class automatically 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.

```
<a href="" routerLink="/" routerLinkActive="selected-menu">Home</a>
```

We can even get a reference on this directive, to know if the route is active, and use it in the template:

```
<a href="" routerLink="/" routerLinkActive #route="routerLinkActive">Home {{
route.isActive ? '(here)' : '' }}</a>
```

It's also possible to navigate from the code, by using the `Router` service and its method `navigate()`. It's often handy when you want to redirect your user after an action:

```
export class RacesComponent {
  constructor(private router: Router) {
  }

  saveAndMoveBackToHome() {
    // ... save logic ...
    this.router.navigate(['']);
  }
}
```

The method takes an array of parameters, with the path you want to navigate to as the first element.

It is also possible to have parameters in the URL, and it's really useful to define dynamic URLs. For example, we want to display a detail page for a pony, with a meaningful URL for this page, like `ponies/id-of-the-pony-/name-of-the-pony`.

To do so, let's define a route in the configuration with one (or several) dynamic parameter.

```
export const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'races', component: RacesComponent },
  { path: 'races/:raceId/ponies/:ponyId', component: PonyComponent }
];
```

We can then define dynamic links with `routerLink`:

```
<a href="" [routerLink]="['/races', race.id, 'ponies', pony.id]">See pony</a>
```

Of course, the target component needs to access those parameters to be able to load and display the pony with the given identifier. To get the value of the parameters, the router provides a service, that you can of course inject in the component, named `ActivatedRoute`. This object can be used inside `ngOnInit`, and has a very useful field: `snapshot`. This field has all the parameters of the URL in `paramMap`!

```
export class PonyComponent implements OnInit {
  pony: any;

  constructor(private ponyService: PonyService, private route: ActivatedRoute) {
  }

  ngOnInit() {
    const id = this.route.snapshot.paramMap.get('ponyId');
    this.ponyService.get(id).subscribe(pony => this.pony = pony);
  }
}
```

This hook is also a good place to do the initialization work of the component as you can see.

As you may have spotted, we are using `snapshot`. Is there a non snapshot version? Yes there is. And it provides a way to subscribe to parameter changes, with, you guessed it, an observable. This observable is called `paramMap`.

WARNING

This is very important: the router will reuse your component if it can! Let's say our app has a "Next" button to see the next pony. The URL will change from `/ponies/1` to `/ponies/2` for example when the user clicks. The router will then reuse our component instance: that means the `ngOnInit` will not be called again! If you want your component to update for this kind of navigation, you have no other way than using the `paramMap` observable!

```
export class PonyReusableComponent implements OnInit {
  pony: any;

  constructor(private ponyService: PonyService, private route: ActivatedRoute) {
  }

  ngOnInit() {
    this.route.paramMap.subscribe((params: ParamMap) => {
      const id = params.get('ponyId');
      this.ponyService.get(id).subscribe(pony => this.pony = pony);
    });
  }
}
```

Here we subscribe to the observable offered by `ActivatedRoute`. Now, every time the URL will change from `/ponies/1` to `/ponies/2` for example, the `paramMap` observable will emit an event, and we'll fetch the correct pony to display on screen.

Note that instead of subscribing to the result of the `PonyService` inside the subscribe of the params update, we can use the more elegant `switchMap` operator:

```
export class PonySwitchMapComponent implements OnInit {
  pony: any;

  constructor(private ponyService: PonyService, private route: ActivatedRoute) {
  }

  ngOnInit() {
    this.route.paramMap
      .map((params: ParamMap) => params.get('ponyId'))
      .switchMap(id => this.ponyService.get(id))
      .subscribe(pony => this.pony = pony);
  }
}
```

PRACTICE Try our exercise Router 🦄 to learn how to configure the router, navigate between components, and test all this.

17.3. Advanced routing

What you just learnt should cover your basic routing needs. But the router goes well beyond this and offers many additional features. Covering them all in details is quite a big task, and you can feel overwhelmed when trying to learn them all.

This section will try to present most of the additional features as concisely as possible, by explaining what they're useful for. But it will not try covering every detail. If you really need an extensive coverage of the router features, there's a whole book available for that, written by the

main contributor to the Angular router, Victor Savkin.

17.3.1. Redirects

A common use-case is to have a URL simply redirect to another URL in the application. This can happen because you want, for example, the root URL of your news app to redirect to the `/breaking` news category, or an old URL to redirect to a new one after a refactoring. This is possible using

```
{ path: '', pathMatch: 'full', redirectTo: '/breaking' },
```

17.3.2. Matching strategy

In the above example illustrating a redirect, I applied a strategy for matching the route: `'full'`. The default strategy is `'prefix'`, which matches a route with a URL when the URL starts with the path of the route. If we used this default strategy here, all URLs would redirect to `/breaking`, since all URLs start with an empty string.

The matching strategy consists in finding the first route that matches the complete URL. So, for example, if you define routes like

```
{ path: 'races/:id', component: RaceComponent },
{ path: 'races/new', component: RaceCreationComponent }
```

and the URL is `races/new`, the component that the router will activate is in fact the `RaceComponent`. Indeed, `races/:id` matches with `races/new` and comes first in the list of routes. To solve this problem, change the order of the routes:

```
{ path: 'races/new', component: RaceCreationComponent },
{ path: 'races/:id', component: RaceComponent }
```

17.3.3. Hierarchical and empty-path routes

Routes can have children. This can be useful for several reasons:

- applying guards to several routes at once (see later);
- applying resolvers to several routes at once (see later);
- sharing a a common template between several routes.

As we have seen before, when the router activates a route, the component of the route is inserted in the page at the location marked by the `router-outlet` directive.

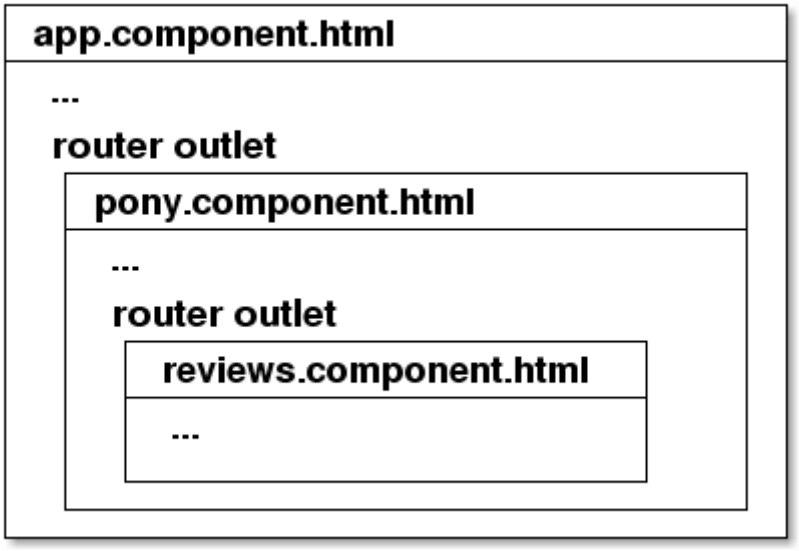
This mechanism can in fact be used in nested components, too. Suppose you have a complex page to display the profile of a pony. This page would display its name and portrait at the top, and would have several tabs at the bottom: one to display its birth certificate, one to display its track record, and one to display journalist reviews about this pony. You want to have a URL for each tab, in order

to be able to directly link to them. But you don't want to reload the pony and repeat its name and portrait on every on these three tab components.

The solution is to use a nested `router-outlet` in the tempate of the `PonyComponent`, and to define a parent pony route, this way:

```
{
  path: 'ponies/:ponyId',
  component: PonyComponent,
  children: [
    { path: 'birth-certificate', component: BirthCertificateComponent },
    { path: 'track-record', component: TrackRecordComponent },
    { path: 'reviews', component: ReviewsComponent }
  ]
}
```

When going to the URL `ponies/42/reviews`, for example, the router will insert the PonyComponent at the location indicated by the main router-outlet, in the root component. The template of PonyComponent, besides the name and the portrait of the pony, contains a second router-outlet. This is where the child `ReviewsComponent` will be inserted.



When going to the URL `ponies/42`, the pony component will be displayed, but none of the three children component will. You might want to display the birth certificate tab by default. That can be achieved using an empty-path route, redirecting to the `birth-certificate` route:

```
{
  path: 'ponies/:ponyId',
  component: PonyComponent,
  children: [
    { path: '', pathMatch: 'full', redirectTo: 'birth-certificate' },
    { path: 'birth-certificate', component: BirthCertificateComponent },
    { path: 'track-record', component: TrackRecordComponent },
    { path: 'reviews', component: ReviewsComponent }
  ]
}
```

Note that, in the above example, the redirect is relative to the `ponies/:ponyId` route, because it doesn't start with a `/`.

Instead of redirecting, you might want to display the birth certificate at the URL `ponies/42`. This can also be achieved using a child empty-path route:

```
{
  path: 'ponies/:ponyId',
  component: PonyComponent,
  children: [
    { path: '', component: BirthCertificateComponent },
    { path: 'track-record', component: TrackRecordComponent },
    { path: 'reviews', component: ReviewsComponent }
  ]
}
```

17.3.4. Guards

Some routes of the application should not be accessible to all users, depending on their permissions. Of course, you should hide or disable links pointing to these routes if the user may not access them. You should also make sure that the backend doesn't allow accessing or modifying resources that the user isn't authorized to. But that still won't prevent users to access routes that they're not allowed to, simply by entering their URL in the address bar.

That's where guards come into play. There are 4 kinds of guards:

- `CanActivate`: when set on a route, the guard can disable the activation of this route. Note that the guard can also have a side effect, like navigating elsewhere. This can be useful to show an error page, or to navigate to the login page when an unauthenticated user tries accessing a route that requires authentication;
- `CanActivateChild`: when set on a route, the guard can disable the activation of children of that route. This can be useful to disable access to many child routes at once, based on their URL;
- `CanLoad`: this guard is used on a route with a `loadChildren` attributes. This attribute allows lazy loading a whole feature bundle, containing child routes (we'll talk about lazy loading a bit later). It goes further than the `CanActivate` guard by preventing to even load the feature bundle;

- `CanDeactivate`: this guard is different from the three other ones. It's used to prevent navigation from outside of the currently activated route. This can be useful to ask for confirmation before leaving a route containing a large form, for example.

Here's how you would add a `CanActivate` guard on a route. The three other guards are added in a similar way:

```
{ path: 'races', component: RacesComponent, canActivate: [LoggedInGuard] }
```

In the above example, `LoggedInGuard` is an Angular service. As any other service, it must be provided, typically by adding it to the providers of the Angular module.

This service must implement the `CanActivate` interface. It consists in deciding whether the route can be activated or not (by checking if the user is logged in or not), and in returning a `boolean`, a `Promise<boolean>`, or an `Observable<boolean>`.

The router will navigate to the route if the returned value is `true`, or if the returned promise is resolved as `true`, or if the returned observable emits `true`.

Here's what the `LoggedInGuard` might look like:

```
@Injectable()
export class LoggedInGuard implements CanActivate {

  constructor(private router: Router, private userService: UserService) { }

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable
<boolean>|Promise<boolean>|boolean {
    const loggedIn = this.userService.isLoggedIn();
    if (!loggedIn) {
      this.router.navigate(['/login']);
    }
    return loggedIn;
  }
}
```

NOTE Instead of defining a service and passing its class in the route configuration, a function with the same signature as the service method can also be passed. I wouldn't advise it in the general case though, as it is less type-safe and doesn't allow injecting a service in the guard.

Hierarchical routes combined with empty-path routes can be very handy to apply a guard on several routes at once. For example, if you want both the races and the ponies routes to be accessible only to logged in users, instead of

```
{ path: 'ponies/:ponyId', component: PonyComponent, canActivate: [LoggedInGuard] },
{ path: 'races', component: RacesComponent, canActivate: [LoggedInGuard] }
```

you can introduce an empty-path, componentless route as a parent. This route won't consume any URL segment, and won't activate any component, but its guards will be called when navigating to any of its children:

```
{
  path: '',
  canActivate: [LoggedInGuard],
  children: [
    { path: 'ponies/:ponyId', component: PonyComponent },
    { path: 'races', component: RacesComponent }
  ]
}
```

17.3.5. Resolvers

In a good old multi-page application where the pages are generated server-side, when clicking on a link, here's what happens: a request is sent to the server, the browser typically shows a spinning icon on the tab, and when the response finally comes back from the server, the URL in the address bar changes and the content of the new page is displayed.

In an Angular single-page application, it doesn't exactly work that way. The user clicks on a link to display a pony race (for example). The router creates an instance of the `RaceComponent`, and the component sends an AJAX request to load the race. The router immediately inserts the component template at the router-outlet location and changes the URL in the address bar. At this time, immediately after the click, the user sees the new page, but without any race. When the response to the AJAX request comes back, the race is stored in the component and the DOM is updated.

This has advantages and drawbacks:

- the navigation to the new page feels faster;
- the user can be confused if loading the race is too long, because the page appears blank, which looks like a bug;
- the template must be coded carefully, in order to work fine during the small period of time when the race is `null` or `undefined`;
- the template can however provide an immediate feedback by displaying a message or a spinning animation indicating that the race is being loaded;
- if loading the race fails (because the connectivity is lost, for example), then the navigation has been made and the URL has changed, although the page can't display any race, instead of staying on the previous page.

A resolver allows making the application behave almost like a traditional multi-page application. Instead of letting the race component load the race, you apply a resolver on the route, and the

resolver loads the race on behalf of the component.

Like a guard, a resolver can return data synchronously (by returning a race) or asynchronously (by returning a promise or observable of race). The router only navigates to the route once the promise has been resolved, or once the observable has completed with at least an emitted race. Here's what a resolver for a race would look like:

```
@Injectable()
export class RaceResolver implements Resolve<RaceModel> {

  constructor(private raceService: RaceService) { }

  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable
<RaceModel> | Promise<RaceModel> | RaceModel {
    return this.raceService.get(+route.paramMap.get('raceId'));
  }
}
```

As you can see, it's a simple service, which uses the activated route snapshot passed by the router to get the value of the `raceId` parameter, and returns an `Observable<RaceModel>`.

NOTE you might wonder why we use `+route.paramMap.get('raceId')` and not simply `route.paramMap.get('raceId')`. That's because the parameter is of type string (it's a segment of a URL), and the service expects the race ID to be a number. `+` is the simplest way to convert a string into a number.

Here's how the resolver would be applied to the route:

```
{
  path: 'races/:raceId',
  component: RaceComponent,
  resolve: {
    race: RaceResolver
  }
}
```

As you can see, the `RaceResolver` is associated with an object key that I chose to name `race`. This is the key that the router will use to store the loaded race into the `data` of the activated route snapshot. So the race component can simply obtain the race the following way:

```
export class RaceComponent implements OnInit {

  race: RaceModel;

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    this.race = this.route.snapshot.data['race'];
  }
}
```

Note that, if you navigate from a route to the same route, but with different parameters (for example, if you have a *Next race* link on the page), then the guards and the resolvers applied to the route are called again. The component, in that case, will still be reused, and should still subscribe to an observable to get the race (or just store the observable in the component and use the `async` pipe in the template):

```
export class RaceComponent implements OnInit {

  race: RaceModel;

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    this.route.data.subscribe(data => this.race = data['race']);
  }
}
```

Resolvers have many advantages over loading data from the activated component:

- they make the navigation more traditional;
- they can be shared and reused by several routes;
- they make the code of the component and its template simpler: no need to load data, no need to care about the temporary undefined or null model, no need to apply somewhat complex RxJS operators to get the data from the parameters;
- if the navigation fails, the current page is preserved and the user can just click on the link again to retry it.

The only drawback I can find is that, when you know that loading the data is slow (because it requires substantial computations or external service calls by the server), then the application can feel a bit unresponsive: you click on a link, and nothing happens until the data has been loaded. This is where loading the data from the activated component and displaying a loading message or animation can be more user-friendly. Another workaround would be to rely on router events to display this loading message.

17.3.6. Router events

The router emits several events when navigating to a route. You can be notified of these events by injecting the `Router` service and subscribing to its `events` observable. The emitted events have several types that you can filter using `event instanceof NavigationStart` (for example). Here are the various types of router events:

- `NavigationStart`: emitted when a navigation is requested (when clicking on a link, for example). It can be used, for example, to start displaying a spinner;
- `NavigationEnd`: emitted when a navigation ends successfully. It can be used to stop displaying the spinner. Another use-case is to send a *hit* to an analytics service (like Google Analytics for example), which allows analyzing the browsing habits and popular pages in your application;
- `NavigationError`: emitted when a navigation fails due to an unexpected error (like a resolver returning an empty or error observable). It can be used to stop displaying a spinner, or to try sending an error log to the server;
- `NavigationCancel`: emitted when a navigation is cancelled, because a guard prevented the navigation for example. If a spinner has been shown when the navigation started, it should be hidden when this event is emitted.

There are other kinds of events for the route configuration loading (`RouteConfigLoadStart`, `RouteConfigLoadEnd`, `RoutesRecognized`) and, since version 4.3, for the resolvers (`ResolveStart`, `ResolveEnd`) and guards (`GuardsCheckStart`, `GuardsCheckEnd`).

17.3.7. Parameters and data

We've seen before that routes can have parameters. For example, the route `races/:raceId` has one parameter named `raceId`, and the value of this parameter, when navigating to `/races/42` is the string `'42'`. But this route can actually have additional parameters named *matrix parameters*.

NOTE	Matrix parameters are not an angular-specific feature. Although rarely used and thus lesser-known than query parameters, they're a standard part of URIs, that are supported by many server-side frameworks, too.

If you navigate to the URL

```
/races/42;foo=bar;baz=wiz
```

then the `params` and `paramMap` properties of the activated route will contain two additional parameters 'foo' and 'baz' having the values 'bar' and 'wiz'.

Those matrix parameters are specific to the route. So, for example, if the URL is

```
/races/42;foo=bar;baz=wiz/ponies
```

then the component associated to the `ponies` segment won't have `foo` nor `bar` in the parameters of its activated route. Only the component associated with the `races/42` segment will.

To navigate to such a URL, you would use the following code:

```
router.navigate(['/races', 42, {foo: 'bar', baz: 'wiz'}, 'ponies']);
```

or an equivalent router link:

```
<a [routerLink]="['/races', 42, {foo: 'bar', baz: 'wiz'}, 'ponies']">Link</a>
```

Query parameters, on the other hand, are shared by all the route segments. They look like this in the URL:

```
/races/42/ponies?foo=bar&baz=wiz
```

These query parameters are accessible from any route, using the `queryParams` or `queryParamMap` property.

To navigate to such a URL, you would use the following code

```
router.navigate(['/races', 42, 'ponies'], { queryParams: {foo: 'bar', 'baz': 'wiz'}
});
```

or the equivalent router link:

```
<a [routerLink]="['/races', 42, 'ponies']"  [queryParams]="{foo: 'bar', baz: 'wiz'}"
>Link</a>
```

Finally, we've seen that resolvers allowed adding properties to the `data` property of the activated route, before the route is activated. It's also possible to add additional data to a route directly from its configuration. This can be useful when the same component can be used in two different contexts for example:

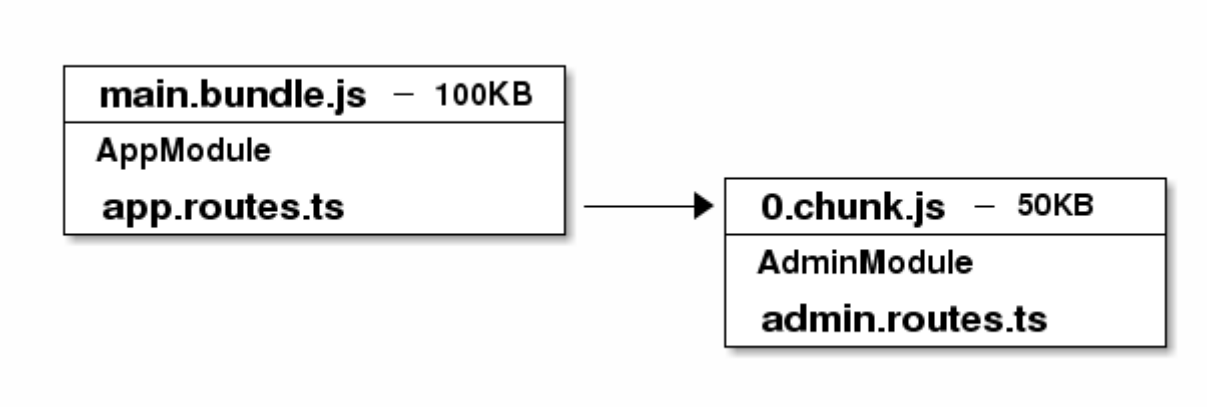
```
{
  path: 'races',
  component: RacesComponent,
  data: {
    allowDeletion: false
  }
}
```

17.3.8. Lazy loading

This section will conclude this long chapter about the Angular router.

When the application grows in size and features, loading the whole application at once can become a problem: the application bundle is too large and takes too much time to load and parse. Moreover, some parts of the application are only used by some users of the application, or are used rarely, and loading them eagerly is a waste of time and bandwidth. This is where lazy-loading is useful.

Lazy loading consists in splitting the application into multiple Angular modules, and into several JavaScript bundles. A lazy-loaded module defines its own routes, components and services, bundled into a separate JavaScript file. The main module only defines a route allowing to access the lazy-loaded module. As the name implies, the Angular router waits for the user to navigate to this route before loading the JavaScript module, and adding the child routes, module, component and services to the application.



NOTE it's actually possible to load the lazy-loading modules in the background, after the root module has been loaded and the application has started, without waiting for the user to navigate to the module, thanks to an alternative `preloadingStrategy`.

To illustrate how we can configure lazy loading, we will assume that you want to define an *admin* section in your application, that should be lazy loaded. We will also assume that you're using Angular CLI to build your application.

The first step is to define a child Angular module. The `admin.module.ts` file would look like this:

```
@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class AdminModule { }
```

There is an important difference with the root module: this child module doesn't import BrowserModule. Instead, it imports CommonModule.

The second step is to define an admin component, and at least one route for this component in a file named `admin.routes.ts`:

```
export const ADMIN_ROUTES: Routes = [
  { path: '', component: AdminComponent }
];
```

This routing configuration must be imported in the admin module, but once again, there is a subtle but important difference with the way it's done in the root module:

```
@NgModule({
  imports: [
    CommonModule,
    RouterModule.forChild(ADMIN_ROUTES)
  ],
  declarations: [AdminComponent]
})
export class AdminModule { }
```

Instead of using `RouterModule.forRoot()` we must use `RouterModule.forChild()`, because this is a child module. Since the router is a stateful singleton service, we must not provide it a second time from this child module.

NOTE	These differences between a child and a root module are not specific to lazy-loaded modules. We will explain the rationale for these differences and the subtleties of root and child modules in a future chapter.

The `AdminComponent` is declared in this child admin module, and not in the root module. Declaring it in the root module would ruin the goal we have: only loading this component when needed, lazily.

The final step is to add a route in the main `app.routes.ts` file, and tell the router to lazy-load the admin module when navigating to that route (or any child route it might have):

```
{ path: 'admin', loadChildren: './admin/admin.module#AdminModule' }
```

As you can see, this is achieved using the `loadChildren` property of the route definition.

IMPORTANT	The value of this property is a string containing the relative path of the admin module to load lazily, followed by the name of the class decorated with `NgModule`. It must **not** be a reference to the AdminModule class: that would prevent the AdminModule to be loaded lazily, since the eagerly loaded root module would have to import the admin module, which itself imports all its declared components.

When building this application, Angular CLI parses the route configurations and detects that the admin child module is lazy-loaded. Without any more work on your part, it generates an additional JavaScript bundle for the admin module (named `0.chunk.js`), and generates the necessary JavaScript to load this bundle when the router requires `'./admin/admin.module'`.

Chapter 18. Forms

WARNING	This chapter uses the new form API (from the package `@angular/forms` which appeared in `rc.2`). If you're still using the old, deprecated forms, it's time to update. This chapter will hopefully help you migrate!

18.1. 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, 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!

You can either write your form using only directives in your template: that's the "template-driven" way. From our experience, it shines when you have a simple form, with not much validation.

The other way is the "code-driven" way, where you will write a description of the form in your component, then use directives to bind this form to the inputs/textareas/selects in your template. It's more verbose, but also more powerful, especially if you want to do add custom validation, or to generate dynamic forms.

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 '@angular/core';

@Component({
  selector: 'ns-register',
  template: `
  <h2>Sign up</h2>
  <form></form>
  `
})
export class RegisterFormComponent {

}
```

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.

For both methods, Angular will create a representation of our form.

In the "template-driven" way, it's pretty much automatic: we just need to add the proper directives in the template and the framework takes care of the form representation creation.

In the "code-driven" way, we create this form representation manually, and then bind the form representation to the inputs using directives.

Behind the scenes, a form field, like an `input` or a `select`, is represented by a `FormControl` in Angular. It is the smallest part of a form, and it encapsulates the state of the field and its value.

A `FormControl` has several attributes:

- `valid`: if the field is valid, regarding the requirements and validations applied on it.
- `invalid`: if the field is invalid, 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.
- `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:

```
const password = new FormControl();
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.

```
const password = new FormControl('Cédric');
console.log(password.value); // logs "Cédric"
```

These controls can be grouped in a `FormGroup` to represent a part of the form and have dedicated validation rules. The form itself is a group.

A `FormGroup` has the same properties as a `FormControl`, with a few differences:

- `valid`: if all fields are valid, then the group is valid.
- `invalid`: if one of the fields is invalid, then the group is invalid.
- `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 of the controls gets dirty.
- `pristine`: the opposite of dirty.
- `touched`: false until one of the controls gets touched.
- `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.
- `valueChanges`: an Observable emitting every time there is a change on the group

It offers the same methods as `FormControl` like `hasError()`. It also has a method `get()` to retrieve a control in the group.

You can create one like this:

```
const form = new FormGroup({
  username: new FormControl('Cédric'),
  password: new FormControl()
});
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.get('username')); // logs the Control
```

Let's begin with a "template-driven" form!

18.2. Template-driven

With this method, we are going to use a bunch of directives in our form, and let the framework build the necessary `FormControl` and `FormGroup` instances. For example, the `NgForm` directive transforms the `form` element into its powerful Angular version - think of it as the difference between Bruce Wayne and Batman.

All the directives we need are included in the `FormsModule` module, so we need to import it in our root module.

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

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

`FormsModule` contains the directives for the "template-driven" way. We'll see later that there exists another module, `ReactiveFormsModule`, in the same package `@angular/forms`, which is needed for the "code-driven" way.

Let's add the submit button:

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

@Component({
  selector: 'ns-register',
  template: `
  <h2>Sign up</h2>
  <form (ngSubmit)="register()">
    <button type="submit">Register</button>
  </form>
  `
})
export class RegisterFormComponent {
  register() {
    // we will have to handle the submission
  }
}
```

I added a button, and defined an event handler for `ngSubmit` on the `form` tag. The `ngSubmit` event is emitted by the `NgForm` 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 '@angular/core';

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html'
})
export class RegisterFormComponent {
  register() {
    // we will have to handle the submission
  }
}
```

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 `ngModel` directives to your form template and that's all. The `NgModel` directive creates the `FormControl` for you, and the form automatically creates the `FormGroup`. Note that you have to give a name to the input, that will be used by the framework to create the `FormGroup`.

```
<h2>Sign up</h2>
<form (ngSubmit)="register()">
  <div>
    <label>Username</label><input name="username" ngModel>
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel>
  </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 name and password. To achieve that, we can define a local variable and assign it with the `NgForm` object created by Angular for the form. Remember these from the Template chapter? Here, we are going to define a variable, `userForm`, referencing the form. We can do that because the `form` directive exports the `NgForm` directive instance, which has the same methods as the `FormGroup` class. We'll see the exporting part in more details when we study how to build advanced directives.

```
<h2>Sign up</h2>
<!-- we use a local variable #userForm -->
<!-- and give its value to the register method -->
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input name="username" ngModel>
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel>
  </div>
  <button type="submit">Register</button>
</form>
```

Our `register` method is now called with the form value as the argument:

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

@Component({
  selector: 'ns-register',
  templateUrl: 'register.component.html'
})
export class RegisterFormComponent {
  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. But `ngModel` is more powerful than you think!

18.2.1. 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:

```
<!-- AngularJS 1.x code example -->
<input type="text" ng-model="username">
<p>{{ username }}</p>
```

We can do a similar thing with Angular.

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 `RegisterFormComponent` should have a field `user` of type `User`:

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

class User {
  username: string;
  password: string;
}

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html',
})
export class RegisterFormComponent {
  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. We need to bind our `input`s to the model we have defined. For this, we'll use the `ngModel` directive:

```
<h2>Sign up</h2>
<form (ngSubmit)="register()">
  <div>
    <label>Username</label><input name="username" [(ngModel)]="user.username">
  </div>
  <div>
    <label>Password</label><input type="password" name="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 name="username" [ngModel]="user.username" (ngModelChange)="user.username =
$event">
```

The `NgModel` directive updates the `input` value every time the related model `user.username` changes, hence the `[ngModel]="user.username"` part. And it emits an event from an output named `ngModelChange` every time the `input` is updated by the user, where the event is the new value, hence the `(ngModelChange)="user.username = $event"` part, which will update the model `user.username` with this new value.

Instead of writing the long form, we can use the new syntax `[()]`. If, like me, you have trouble to remember if it is `[()]` or `([])`, there is a cool mnemonic tip: it's a banana-box! Yes, look: the `[]` is a box, and, inside, there are two bananas facing each other `()`!

Now, every time we type something in our `input`, the model is 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 name="username" [(ngModel)]="user.username">
    <small>{{ user.username }} is an awesome username!</small>
  </div>
  <div>
    <label>Password</label><input type="password" name="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!

18.3. Code-driven

In AngularJS 1.x you had to build your forms mostly in your templates. Angular introduces an imperative way, which allows to construct the form programmatically rather than through a template.

Now we can handle forms directly in our code. It's more verbose but more powerful.

To build a form in our component code, we'll use the abstractions we talked about: `FormControl` and `FormGroup`.

With these basic elements we can build a form in our component. But instead of writing `new FormControl()` or `new FormGroup()`, we will use a helper class, `FormBuilder`, that we can inject:

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

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html'
})
export class RegisterFormComponent {

  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 groups. Let's start simple, and create a small form with two controls, a username and a password.

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html'
})
export class RegisterFormComponent {
  userForm: FormGroup;

  constructor(fb: FormBuilder) {
    this.userForm = fb.group({
      username: '',
      password: ''
    });
  }

  register() {
    // we will have to handle the submission
  }
}
```

We created a `form` with two controls. You can see that each control is created with the value `''`. That's the same as using the helper method `control()` of the `FormBuilder` with this string as parameter, and the same as calling the `new FormControl('')` constructor: 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 `FormGroup` object has a `value` attribute, so we can simply log its content with:

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html'
})
export class RegisterFormComponent {
  userForm: FormGroup;

  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. We are going to use other directives than those we saw for the "template-driven" forms. These directives are in the `ReactiveFormsModule` that you have to import in your root module. Their names begin with `form` instead of `ng` as it was the case for the "template-driven" forms.

The form needs to be bound to our `userForm` object, thanks to the `formGroup` directive. Each `input` field is bound to a control, thanks to the `formControlName` directive:

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [formGroup]="userForm">
  <div>
    <label>Username</label><input formControlName="username">
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
  </div>
  <button type="submit">Register</button>
</form>
```

We want to bind our component's attribute `userForm` object to `formGroup`, so we use the bracket notation `[formGroup]="userForm"`. Each `input` receives the `formControlName` 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. As we pass a value (and not an expression), we don't put the `[]` around `formControlName`.

And we're done: clicking on the submit button will log an object containing the username and the

chosen password!

If you need to, you can update the value of a `FormControl` from your component, using `setValue()`:

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, FormControl } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html',
})
export class RegisterFormComponent {
  usernameCtrl: FormControl;
  passwordCtrl: FormControl;
  userForm: FormGroup;

  constructor(fb: FormBuilder) {
    this.usernameCtrl = fb.control('');
    this.passwordCtrl = fb.control('');
    this.userForm = fb.group({
      username: this.usernameCtrl,
      password: this.passwordCtrl
    });
  }

  reset() {
    this.usernameCtrl.setValue('');
    this.passwordCtrl.setValue('');
  }

  register() {
    console.log(this.userForm.value);
  }
}
```

18.4. 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.

18.4.1. In a code-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.email()` (available since version 4.0) to ensure that the value entered is a valid email address (good luck to find the correct regular expression by yourself for this one…).
- `Validators.pattern(p)` to ensure that the value matches the regular expression *p*

You can apply several validators at once, by using an array, on a `FormControl` or on a `FormGroup`. 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 '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html',
})
export class RegisterFormComponent {
  userForm: FormGroup;

  constructor(fb: FormBuilder) {
    this.userForm = fb.group({
      username: fb.control('', [Validators.required, Validators.minLength(3)]),
      password: fb.control('', Validators.required)
    });
  }

  register() {
    console.log(this.userForm.value);
  }
}
```

18.4.2. 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 `input`s. `required` is a provided directive, and will automatically add the validator to this field. Same thing with `minlength`, `maxlength` and `email`.

Starting from the two-way data-binding example:

```
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input name="username" ngModel required minlength="3">
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel required>
  </div>
  <button type="submit">Register</button>
</form>
```

Note that this can be done in a "code-driven" form too.

18.5. 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.

18.5.1. Errors and submission in a code-driven form

We added a field `userForm`, of type `FormGroup`, to our component. This field 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()" [formGroup]="userForm">
  <div>
    <label>Username</label><input formControlName="username">
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>
```

As you can see on the last line, we just need to link `disabled` to the `invalid` 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()" [formGroup]="userForm">
  <div>
    <label>Username</label><input formControlName="username">
    <div *ngIf="userForm.get('username').hasError('required')">Username is
required</div>
    <div *ngIf="userForm.get('username').hasError('minlength')">Username should be 3
characters min</div>
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
    <div *ngIf="userForm.get('password').hasError('required')">Password is
required</div>
  </div>
  <button type="submit" [disabled]="userForm.invalid">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 until the user changes the value?

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [formGroup]="userForm">
  <div>
    <label>Username</label><input formControlName="username">
    <div *ngIf="userForm.get('username').dirty &&
userForm.get('username').hasError('required')">
      Username is required
    </div>
    <div *ngIf="userForm.get('username').dirty &&
userForm.get('username').hasError('minlength')">
      Username should be 3 characters min
    </div>
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
    <div *ngIf="userForm.get('password').dirty &&
userForm.get('password').hasError('required')">
      Password is required
    </div>
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>
```

It's a bit verbose, but you can create a reference for each control in your component:

```
@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html'
})
export class RegisterFormComponent {
  userForm: FormGroup;
  usernameCtrl: FormControl;
  passwordCtrl: FormControl;

  constructor(fb: FormBuilder) {
    this.usernameCtrl = fb.control('', Validators.required);
    this.passwordCtrl = fb.control('', Validators.required);

    this.userForm = fb.group({
      username: this.usernameCtrl,
      password: this.passwordCtrl
    });
  }

  register() {
    console.log(this.userForm.value);
  }
}
```

And then use the references in your template:

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [formGroup]="userForm">
  <div>
    <label>Username</label><input formControlName="username">
    <div *ngIf="usernameCtrl.dirty && usernameCtrl.hasError('required')">Username is
required</div>
    <div *ngIf="usernameCtrl.dirty && usernameCtrl.hasError('minlength')">Username
should be 3 characters min</div>
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
    <div *ngIf="passwordCtrl.dirty && passwordCtrl.hasError('required')">Password is
required</div>
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>
```

18.5.2. Errors and submission in a template-driven form

In a template-driven form, we don't have any field in our component referring to the `FormGroup`, but we already declared a local variable in the template, referring to the `NgForm` object exported by the form directive. Once again, this variable allows knowing the state of the form and accessing its

controls.

```
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input name="username" ngModel required>
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel required>
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>
```

Now we need to display the errors of each field.

Like the form directive, each control exports its `FormControl` 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 name="username" ngModel required #username=
"ngModel">
    <div *ngIf="username.dirty && username.hasError('required')">Username is
required</div>
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel required
#password="ngModel">
    <div *ngIf="password.dirty && password.hasError('required')">Password is
required</div>
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>
```

Yay!

18.6. Add some style

Whatever way you choose to create your forms, Angular 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 name="username" ngModel required>
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel required>
  </div>
  <button type="submit" [disabled]="userForm.invalid">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 name="username" ngModel required>
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel required>
  </div>
  <button type="submit" [disabled]="userForm.invalid">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`.

18.7. 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 `FormControl`, tests its `value` and returns an object with the errors or `null`, if the validation passes.

```
const isOldEnough = (control: FormControl) => {
  // control is a date input, so we can build the Date from the value
  const 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.

18.7.1. Using a validator in a code-driven form

We need to add a new control in our form with this validator, using the `FormBuilder`:

```
import { Component } from '@angular/core';
import { FormBuilder, FormControl, FormGroup, Validators } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html'
})
export class RegisterFormComponent {
  usernameCtrl: FormControl;
  passwordCtrl: FormControl;
  birthdateCtrl: FormControl;
  userForm: FormGroup;

  static isOldEnough(control: FormControl) {
    // control is a date input, so we can build the Date from the value
    const birthDatePlus18 = new Date(control.value);
    birthDatePlus18.setFullYear(birthDatePlus18.getFullYear() + 18);
    return birthDatePlus18 < new Date() ? null : { tooYoung: true };
  }

  constructor(fb: FormBuilder) {
    this.usernameCtrl = fb.control('', Validators.required);
    this.passwordCtrl = fb.control('', Validators.required);
    this.birthdateCtrl = fb.control('', [Validators.required, RegisterFormComponent
.isOldEnough]);
    this.userForm = fb.group({
      username: this.usernameCtrl,
      password: this.passwordCtrl,
      birthdate: this.birthdateCtrl
    });
  }

  register() {
    console.log(this.userForm.value);
  }
}
```

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 static 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()" [formGroup]="userForm">
  <div>
    <label>Username</label><input formControlName="username">
    <div *ngIf="usernameCtrl.dirty && usernameCtrl.hasError('required')">Username is
required</div>
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
    <div *ngIf="passwordCtrl.dirty && passwordCtrl.hasError('required')">Password is
required</div>
  </div>
  <div>
    <label>Birth date</label><input type="date" formControlName="birthdate">
    <div *ngIf="birthdateCtrl.dirty">
      <div *ngIf="birthdateCtrl.hasError('required')">Birth date is required</div>
      <div *ngIf="birthdateCtrl.hasError('tooYoung')">You're way too young to be
betting on pony races</div>
    </div>
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>
```

Pretty easy, no?

Note that it's also possible to create and add asynchronous validators (for example to check with the backend if a username is available).

```
@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html'
})
export class RegisterFormComponent {
  usernameCtrl: FormControl;
  userForm: FormGroup;

  constructor(fb: FormBuilder, private userService: UserService) {
    this.usernameCtrl = fb.control('', Validators.required, control => this
.isUsernameAvailable(control));
    this.userForm = fb.group({
      username: this.usernameCtrl
    });
  }

  isUsernameAvailable(control: AbstractControl) {
    const username = control.value;
    return this.userService.isUsernameAvailable(username)
      .map(available => available ? null : { alreadyUsed: true });
  }

  register() {
    console.log(this.userForm.value);
  }
}
```

This asynchronous validator is not a static method this time because it needs to access the service.

The method from the service returns an `Observable` that emits either `null` if there is no error (the username is available), or an object to represent the error (the key will be the error, as with synchronous validators).

Interesting feature, the class `ng-pending` is dynamically added to the field while the asynchronous validator is still completing its job. It allows to display a spinner for example to show that the validation is still ongoing.

18.7.2. 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, you need to build a custom directive that we will apply on the `input`, but honestly this is way easier by using a "code-driven" form…

Or you can combine the best of both world!

18.7.3. Combining template-based and code-based approaches for validation

You can do everything "template-based", except for the custom validation! We would end up with

something like this in the view:

```
<label>Username</label><input name="username" [formControl]="usernameCtrl" [(
ngModel)]="user.username" required>
<div class="error" *ngIf="usernameCtrl.dirty && usernameCtrl.hasError('notAllowed')"
>Username is not allowed</div>
```

And in the component:

```
usernameCtrl = new FormControl('', RegisterFormComponent.usernameValidator);

static usernameValidator(control: FormControl) {
  return control.value !== 'admin' ? null : { notAllowed: true };
}
```

This is a really good compromise:

- you don't need to make all your form code-driven;
- you don't need to compose the `required` and the `usernameAllowed` validators yourself: Angular does it for you;
- you still have two-way binding, that you don't have in a pure code-driven approach.

This is a solution to keep in mind: combining the template-based approach for simple things and 2-way binding, and the code-based approach when custom validation is necessary.

18.8. 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.

The solution is to use a code-driven form (that you can combine with a template-driven form if you want to, as we saw above).

First, create a new group, `passwordForm` with the two fields and add it in the group `userForm`:

```
import { Component } from '@angular/core';
import { FormBuilder, FormControl, FormGroup, Validators } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html'
})
export class RegisterFormComponent {
  passwordForm: FormGroup;
  userForm: FormGroup;
  usernameCtrl: FormControl;
  passwordCtrl: FormControl;
  confirmCtrl: FormControl;

  static passwordMatch(group: FormGroup) {
    const password = group.get('password').value;
    const confirm = group.get('confirm').value;
    return password === confirm ? null : { matchingError: true };
  }

  constructor(fb: FormBuilder) {
    this.usernameCtrl = fb.control('', Validators.required);
    this.passwordCtrl = fb.control('', Validators.required);
    this.confirmCtrl = fb.control('', Validators.required);

    this.passwordForm = fb.group(
      { password: this.passwordCtrl, confirm: this.confirmCtrl },
      { validator: RegisterFormComponent.passwordMatch }
    );

    this.userForm = fb.group({ username: this.usernameCtrl, passwordForm: this
.passwordForm });
  }

  register() {
    console.log(this.userForm.value);
  }
}
```

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 `formGroupName` directive:

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [formGroup]="userForm">
  <div>
    <label>Username</label><input formControlName="username">
    <div *ngIf="usernameCtrl.dirty && usernameCtrl.hasError('required')">Username is
required</div>
  </div>
  <div formGroupName="passwordForm">
    <div>
      <label>Password</label><input type="password" formControlName="password">
      <div *ngIf="passwordCtrl.dirty && passwordCtrl.hasError('required')">Password is
required</div>
    </div>
    <div>
      <label>Confirm password</label><input type="password" formControlName="confirm">
      <div *ngIf="confirmCtrl.dirty && confirmCtrl.hasError('required')">Confirm your
password</div>
    </div>
    <div *ngIf="passwordForm.dirty && passwordForm.hasError('matchingError')">Your
password does not match</div>
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>
```

Voilà!

18.9. Reacting on changes

Last cool feature when using a code-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 '@angular/core';
import { FormBuilder, FormControl, FormGroup, Validators } from '@angular/forms';
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/distinctUntilChanged';

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html'
})
export class RegisterFormComponent {
  userForm: FormGroup;
  usernameCtrl: FormControl;
  passwordCtrl: FormControl;
  passwordStrength = 0;

  constructor(fb: FormBuilder) {
    this.usernameCtrl = fb.control('', Validators.required);
    this.passwordCtrl = fb.control('', Validators.required);

    this.userForm = fb.group({
      username: this.usernameCtrl,
      password: this.passwordCtrl
    });

    // we subscribe to every password change
    this.passwordCtrl.valueChanges
      // only recompute when the user stops typing for 400ms
      .debounceTime(400)
      // only recompute if the new value is different than the last
      .distinctUntilChanged()
      .subscribe(newValue => this.passwordStrength = newValue.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()" [formGroup]="userForm">
  <div>
    <label>Username</label><input formControlName="username">
    <div *ngIf="usernameCtrl.dirty && usernameCtrl.hasError('required')">Username is
required</div>
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
    <div>Strength: {{ passwordStrength }}</div>
    <div *ngIf="passwordCtrl.dirty && passwordCtrl.hasError('required')">Password is
required</div>
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>
```

We are leveraging RxJS operators to add a few cool features:

- `debounceTime(400)` will only emit values if the user stops typing for 400ms. That avoids to compute the password strength on every value the user enters. That's really interesting if the computing takes a long time, or launches an HTTP request.
- `distinctUntilChanged()` will only emit values if the new value entered is different than the last one. Again that's really interesting: imagine that the user enters `'password'` then stops typing. We compute the strength. Then she enters a new character and removes it quickly (before 400ms). The next event out of `debounceTime` will again be `'password'`. It makes no sense to recompute the password strength again! This operator will not even emit the value, and saves us the recomputing.

RxJS can do tons of work for you: imagine coding yourself what we just did in two lines. It can also easily combine with HTTP work, as the `Http` service uses observables too.

18.10. Summary

Angular offers two ways to build a form:

- one by setting up everything in the template. But, as you have seen, it forces us to have custom directives for the validation and is harder to test. This way of doing things is useful for simple forms, with just one or a few fields for example, and it gives us two-way data-binding.
- 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.
- combining both of them for the reduced verbosity of "template-driven" forms, and the power of "code-driven" forms for validation.

This is maybe the most pragmatic approach: go with template-based and bidirectional binding if you like it, and as soon as you need access to form groups or form controls, for example to add custom validation or reactive behavior, then declare the ones you need in the component, and bind

the inputs and divs to them using the appropriate directives.

PRACTICE

Try our exercises Register form 🦄, Custom validators in forms 🦄 and Login form 🦄. You'll learn how to build a simple form using the code-driven way and another one with the template-driven way. You'll also learn how to write custom validators, how to test forms, and how to authenticate your users!

Chapter 19. Zones and the Angular magic

Developing with AngularJS 1.X gave a "magic" feeling, and Angular still still gives that same effect: you type some value in an input and everything is magically updating all over the place.

I love magic, but I prefer to understand what's going on with the tools I use. If you are like me, I think this part will be interesting for you too: we are going to see how Angular works under the hood!

But first, let's start with how AngularJS 1.x works, which should be interesting, even you've never used it.

All JavaScript frameworks work roughly the same way: they help the developer to react to application events, to update the application state, and to refresh the DOM accordingly. But they don't all use the same way to achieve that goal.

EmberJS, for example, asks the developers to use *setters* to change the state of the objects, in order for the framework to be able to intercept the calls to these setters. That's what allows it to know which changes have been applied to the model, and thus to update the DOM accordingly.

React, on the other hand, chose to recompute the DOM after each change. But since modifying the whole DOM is a costly operation, it starts by applying the changes to a virtual DOM, and then only applies the changes between the virtual DOM and the actual DOM.

Angular doesn't use any setter, and doesn't use any virtual DOM either. So, how does it know what to change in the DOM?

19.1. AngularJS 1.x and the digest cycle

The first step is to detect changes in the model. A change is always triggered by an event, coming either from the user directly (for example, a button click or an input in a form), or from a "system" event (an HTTP response, an asynchronous method execution after a timeout, etc.).

So, how does AngularJS 1.x know that an event has happened? That part is actually pretty simple: it forces us to use its directives, for example `ng-click` to react to a click event, or `ng-model` to observe changes to an input. It also forces us to use its services, for example `$http` for HTTP requests or `$timeout` to execute tasks asynchronously.

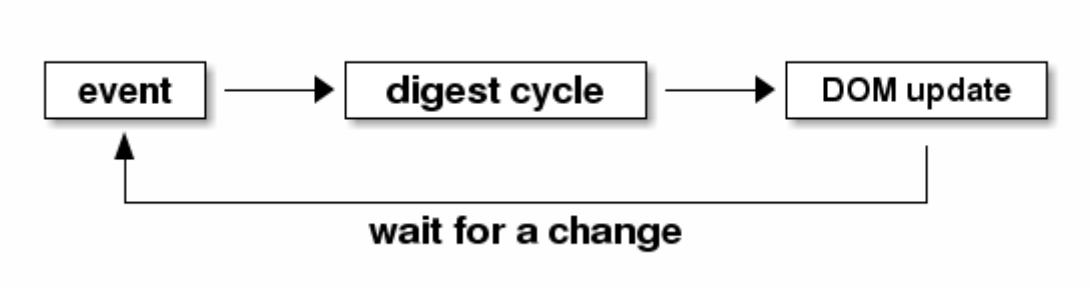
Using these directives and services allows the framework to be well informed about any event that has happened. That's the first part of the magic! And that's this first part which triggers the second one: the framework now has to analyze the changes made to the model, in order to decide which part of the DOM must be updated (and how).

To do that, in version 1.x, the framework maintains a list of *watchers*, all observing and reacting to changes in a specific part of the model. To simplify, a watcher is created for every dynamic expression used in the HTML templates. This can lead to several hundreds of watchers in a page.

These watchers are central to AngularJS 1.x: they are the memory of the framework, and are here to remember the state of the app.

Every time the framework detects an event (a user typing something in an input with an `ng-model`, an HTTP response, a timeout execution, etc.), it triggers what is called the *digest cycle*.

This *digest cycle* evaluates all the expressions stored in the watchers and compares their new value with their old value. If there is a change, then the framework knows that it has to update the DOM, so that the UI displays the new value instead of the old one. This technique is called *dirty checking*.



During this *digest cycle*, Angular is going over the whole list of watchers, and evaluates every watched expression. But there is a catch: it will execute this whole cycle again until the results are stable, i.e. until all the new values are equal to the old values.

Why does it do that? Because each time a change is detected in the value of a watched expression, a callback function is called. And this callback function can, in turn, modify the model, and thus change the value of one or several other watched expressions!

Let's take a minimalistic example: a page with two fields that the user must fill: name and password. Let's also say that the page displays a password strength indicator. A watcher is used to watch the value of the password, and recomputes its strength every time it changes.

After the first iteration of the digest cycle, once the user has entered the first letter of his/her password, we thus have this list of watchers:

```
$$watchers (expression -> value)
- "user.name" -> "Cédric"
- "user.password" -> "h"
- "passwordStrength" -> 0
```

The callback function of the watcher observing the `user.password` expression is then called, and it computes the new password strength: 3.

Angular doesn't know anything about the changes that might have been done on the model, so it starts a second digest iteration to know if the model is stable.

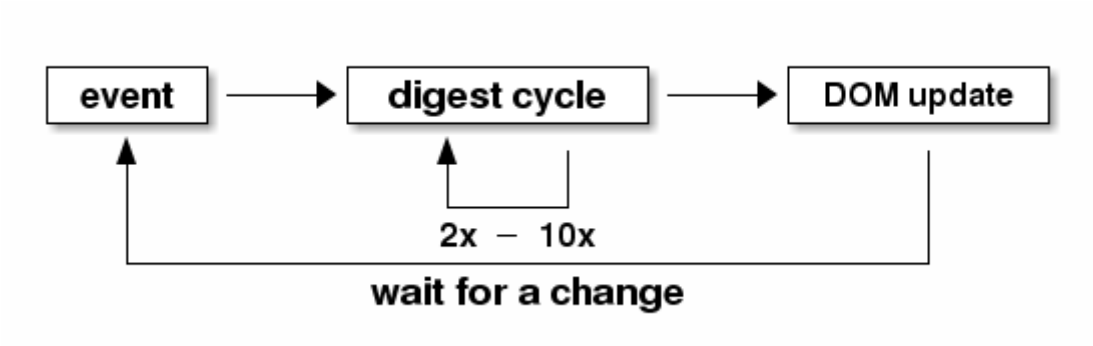
```
$$watchers
- "user.name" -> "Cédric"
- "user.password" -> "h"
- "passwordStrength" -> 3
```

The model isn't stable: the value of `passwordStrength` has changed since the first iteration. So it starts another digest iteration.

```
$$watchers
- "user.name" -> "Cédric"
- "user.password" -> "h"
- "passwordStrength" -> 3
```

This time, the model is stable. That's only at that time that AngularJS 1.x is flushing the result to the DOM. So the digest cycle happens at least 2 times for every change in your application. The digest can iterate up to 10 times, but no more: after 10 iterations, if the results are not stable, the framework considers there is an infinite loop and throws an exception.

So my little drawing earlier is much more like:



And to insist: this is going on after *each event*. That means that if your user is entering 5 characters in the password field, the digest cycle will run 5 times, with 3 iterations every time, which leads to a total of 15 iterations.

In a real application, you can have several hundred watchers, and thus thousands of expression evaluations after every event!

Even if that sounds crazy, it works fine, because modern browsers are very fast, and there are ways to optimize a few things if necessary.

So let's recap the two important points about AngularJS 1.x:

- you must use the services and directives of the framework for everything that can make a change to the model;
- modifying the model after an event that is not handled by Angular is possible, but you have to trigger the detection change mechanism explicitly (by using the famous `$scope.$apply()` method, which starts the digest cycle). For example, if you want to send an HTTP request without using the `$http` service, you need to call `$scope.$apply()` in the response callback to tell the framework: "Hey, I have stored new values in the model, would you please start the digest cycle?".

The magic in the framework can be split in two parts:

- the triggering of the change detection after each event;
- the change detection itself, thanks to the watchers and the digest cycles.

Now let's see how Angular is working, and how it differs from AngularJS.

19.2. Angular and zones

Angular is based on the same principles, but implements them in a different way. And we might even say, in a smarter way.

For the first part of the problem — triggering the change detection — the Angular team has built a small side-project named Zone.js. This project is not tied to Angular, because *zones* are a tool that can be useful in other projects. *Zones* are not a brand new concept either: they already exist in the Dart language (another Google project), for quite some time. They also have similarities with *Domains* in Node.js (now abandoned) or the *ThreadLocals* in Java.

But it's probably the first time that zones are being used in JavaScript. No worries, we'll discover them together.

19.2.1. Zones

A zone is an execution context. This context received code to execute, and this code can be synchronous or asynchronous. A zone brings some benefits:

- hooks that can be executed before and after the code to execute
- a way to intercept potential errors of the code to execute
- a way to store variables bound to this context

Let's take an example. Suppose I have the following code in my application:

```
// score computation -> synchronous
const score = computeScore();
// player score update -> synchronous
updatePlayer(player, score);
```

When we execute that code, we obtain:

```
computeScore: new score: 1000
udpatePlayer: player 1 has 1000 points
```

Now suppose I want to measure how much time is spent executing that code. I can do something like this:

```
startTimer();
const score = computeScore();
updatePlayer(player, score);
stopTimer();
```

And that would produce:

```
start
computeScore: new score: 1000
udpatePlayer: player 1 has 1000 points
stop: 12ms
```

Easy. Now what happens if `updatePlayer` is an asynchronous function? JavaScript works in a quite special way: asynchronous operations are put at the end of the execution queue, and will thus be executed after the synchronous operations.

So, my previous code:

```
    startTimer();
    const score = computeScore();
    updatePlayer(player, score); // asynchronous
    stopTimer();
```

will in fact produce :

```
start
computeScore: new score: 1000
stop: 5ms
udpatePlayer: player 1 has 1000 points
```

My execution time is not correct anymore: it only measures the time taken by the synchronous operations in my code, and not the time elapsed from the start until the end of the score update! That's where zones can be useful. We will execute the code in a dedicated execution context: a zone:

```
const scoreZone = Zone.current.fork({ name: 'scoreZone' });
scoreZone.run(() => {
  const score = computeScore();
  updatePlayer(player, score); // asynchronous
});
```

Why does this help? Well, if the zone.js library is loaded by the browser, it starts by patching all the asynchronous functions in the JavaScript runtime. So, every time we use `setTimeout()` or `setInterval()`, or we use an asynchronous API like Promises, XMLHttpRequest, WebSocket, FileReader, GeoLocation, etc., we will in fact call the patched version of zone.js. Thus, Zone.js knows when asynchronous operations are done and allows us, developers, to execute some *hooks* at that time.

A zone offers several hooks:

- `onInvoke` is called before executing the code wrapped in the zone;
- `onHasTask` is called after the code wrapped in the zone has finished executing;

- `onHandleError` is called when the code wrapped in the zone throws an exception;
- `onFork` is called when the zone is created.

We can thus use a zone and its hooks to measure the whole time spent from the start to the end of my asynchronous operation:

```
const scoreZone = Zone.current.fork({
  name: 'scoreZone',
  onInvoke(delegate, current, target, task, applyThis, applyArgs, source) {
    // start the timer
    startTimer();
    return delegate.invoke(target, task, applyThis, applyArgs, source);
  },
  onHasTask(delegate, current, target, hasTaskState) {
    delegate.hasTask(target, hasTaskState);
    if (!hasTaskState.macroTask) {
      // if the zone run is done, stop the timer
      stopTimer();
    }
  }
});
scoreZone.run(() => {
  const score = computeScore();
  updatePlayer(player, score);
});
```

And this time, the result is correct!

```
start
computeScore: new score: 1000
udpatePlayer: player 1 has 1000 points
stop: 12ms
```

Now you can guess how Angular can benefit from this mechanism. Indeed, the first problem of the framework is to know when change detection must be triggered. By using zones, and by running the code we write in a zone, the framework has a good view of what is happening. It can handle errors in a better way. But more importantly, it can trigger change detection every time an asynchronous operation is done!

To simplify, Angular does something like this :

```
const angularZone = Zone.current.fork({
  name: 'angular',
  onHasTask: triggerChangeDetection
});
angularZone.run(() => {
  // your application code
});
```

And the first problem is thus solved! That's why, in Angular, unlike AngularJS 1.x, it's not necessary to use special services to benefit from automatic change detection. You can use whatever you want, and zones will deal with it.

Note that zones are in a standardization process, and could thus become part of the official ECMAScript specification in a close future. Other interesting piece of information, the current implementation of zone.js also provides information for WTF (which does **not** mean *What The Fuck* in this context, but Web Tracing Framework). This library allows to profile your application while in development mode, and to know exactly how much time was spent in every part of the application and framework code. In short, plenty of information to analyze and troubleshoot performance if needed!

19.2.2. Change detection in Angular

The second part of the problem is change detection itself. It's one thing to know when and how it's started, but it's another to know how it works.

First of all, we have to remember that an Angular application is a tree of components. When change detection starts, the frameworks goes through all the components in the tree to know if their state has changed and if the new state affects their view. If it's the case, the DOM part of the component which is affected by the change is updated.

The tree traversal goes from the root to the leaves and, unlike AngularJS 1.x, is done only once. Because there is now a big difference: the change detection does not change the application model in Angular, whereas a watcher in AngularJS 1.x could change the model during that phase.

Another big difference is that a component may modify its state and the state of its children, but it must not modify the state of its ancestors. So cascading changes are now over!

Change detection is therefore only used for checking changes in the model and modify the DOM accordingly. It can't have side-effects on the model as it could in version 1.x, and thus doesn't need more than one pass through the tree, since the model can't be modified by the traversal!

To be accurate, the traversal, in development mode, is made twice precisely to assert that such indesirable side effects are inexistent (for example, a child component modifying the model of its parent). If the second pass detects such a change, an exception is thrown to warn the developer about the problem.

This strategy has many advantages:

- it's easier to reason about our applications, because state changes are made in one way, from

the parent to the child, instead of being made in both ways;

- the change detection can't have infinite loops anymore;
- the change detection is significantly faster.

About this last point, it's relatively easy to visualize: AngularJS 1.x did (M watchers) * (N cycles) verifications, whereas Angular only does M verifications.

But there is another parameter to take into account in the performance improvements of Angular: the time spent by the framework to make these verifications. Once again, the Google team has employed all the knowledge it has about computer science and virtual machines.

To understand how the performance has been improved, we have to examine how expressions are evaluated and values compared in both versions of the framework.

In version 1.x, the mechanism is very generic. A single generic method is called for every watcher, capable of comparing the old and the new values. The problem is that virtual machines that execute the JavaScript code in our browsers (V8 if you're using Google Chrome, for example), don't really like generic code.

If you don't mind, I'll digress a little to talk about the behavior of virtual machines. You didn't expect that in a book about a JavaScript framework, did you? Virtual machines are quite extraordinary programs: you give them a piece of code, and they transform it to machine code and execute it. Very few among us (and certainly not me), are able to produce efficient machine code, but we don't really care: we use a high-level language, and let the virtual machine deal with it. Not only do they translate the code, but they optimize it. And they're quite good at that. Some even produce code that runs faster than manually optimized code, because they benefit from runtime information that is impossible to know in advance.

To improve performance, virtual machines like the ones executing dynamic JavaScript code, use a strategy named *inline caching*. It's a very old technique, invented for SmallTalk, 40 years ago (an eternity in IT), relying on a relatively simple principle: if a program calls a function frequently, with the objects having the same shape, the VM should recall how it evaluates the properties of the object. This technique thus uses a cache, hence the name *inline caching*. When receiving an object, it looks in the cache to see if it recognizes the shape of the object. And if it does, it uses the cached optimized way of accessing the properties of the object.

This kind of cache is really beneficial if the arguments of the function have the same shape. For example, `{name: 'Cédric'}` and `{name: 'Cyril'}` have the same shape. But `{name: 'JB', skills: []}` doesn't have the same shape as the two other ones.

When the arguments always have the same shape, the cache is *monomorphic*, and thus produces very fast results. If the cache only has a few entries, it is *polymorphic*. That means that the method is called with some different kinds of objects which makes the code a bit slower. Finally, if there are too many different object shapes, the VM drops the cache completely, because it is *megamorphic*. That, of course, is the worst case in terms of performance.

Now let's come back to our change detection in AngularJS 1.x. We can understand now that this unique generic function called for all the watchers is not optimizable using inline caching. We're in a *megamorphic* state, where the code is the slowest. And even if that isn't a problem in most

situations, some pages with many watchers make us reach a limit where the performance is not sufficient anymore.

In order to benefit from the inline caching optimizations of the VM, Angular has adopted a different strategy. Instead of using a single method able to compare all kinds of objects, the Google team decided to dynamically generate a comparator for every type. At the startup of the application, the framework goes through the tree of components, and generates a set of change detection functions, specific to each component.

For example, given a component `PonyComponent` with an attribute `name` displayed in the view, the framework will generate a change detection function for the whole component, pretty much like this one (this is a dumbed down version):

```
(view) => {
  var component = view.component;
  const currVal = component.name;
  if (currVal !== view.oldValue) {
    view.oldValue = currVal;
    updateView(view, 'p', currVal);
  };
});
```

This code is similar to code that you would have written by hand, and that the VM can optimize because it's monomorphic. The result is a significantly faster code, allowing for more complex pages.

To recap, Angular needs to evaluate fewer expressions and compare fewer values than AngularJS 1.x (a single pass is enough), and those evaluations and comparisons are faster!

From the beginning, the Google team has been monitoring the performance with benchmarks comparing AngularJS 1.x, Angular and even Polymer and React on various use cases, in order to check if the new version keeps being faster.

If really needed, it's even possible to go further than those automatic optimizations provided by the framework: the *ChangeDetection* strategy can be changed from its default value, and thus be adapted and tuned to specific use-cases. But that's for another chapter.

Chapter 20. Angular compilation: Just in Time vs Ahead of Time

20.1. Code generation

In the previous chapter, we talked briefly about how the framework was *generating* a change detection function for each component.

This is a very interesting and particular point in Angular, which you don't see in other frameworks: Angular, at the start of your application, will *compile* your templates and *generate* dynamic code for each component.

The HTML you write in your templates is never read by the browser directly. Instead, Angular generates a class for each component that represents exactly what you wrote in your template. This class is generated in a file with the `ngfactory.js` extension, and you can see it in your browser if you open the `Sources` tab in the console.

Let's take an example, with our well-known `PonyComponent`. The template is mainly an image with a bound source property, and a legend with an interpolation.

```
<figure>
  <img [src]="getPonyImageUrl()">
  <legend>{{ ponyModel.name }}</legend>
</figure>
```

When Angular compiles this, it first starts by parsing the template to generate what is called an Abstract Syntax Tree (AST). An AST is a tree representing the structure of the template, commonly used by compilers to represent such things. It is the result of the *syntax analysis* step of the compilation. This AST will then be used to generate the dynamic JavaScript code, a "view definition" per component (in our case `View_PonyComponent`). A view definition contains the nodes of the templates. Nodes can have various types, for example an element (`elementDef`), with its name and attributes, or a simple text node (`textDef`).

```
function View_PonyComponent() {
  return viewDef([
    elementDef('figure', [
      elementDef('img', ['src']),
      elementDef('legend', [
        textDef()
      ])
    ])
  ]);
}
```

In fact, instead of generating the structure as a tree, the compiler generates a flat array (depth-first,

with each node indicating how many children it has), for performance reasons:

```
function View_PonyComponent() {
  return viewDef([
    elementDef(2, 'figure'),
    elementDef(0, 'img',['src']),
    elementDef(1, 'legend'),
    textDef()
  ]);
}
```

With this representation, Angular is able to generate the DOM corresponding to our `PonyComponent`: it basically appends the corresponding `HTMLElement` to the DOM, for every element of the array.

But how does it handle the change detection? The view definition has in fact a last parameter: a change detection function.

```
function View_PonyComponent() {
  return viewDef([
    elementDef(2, 'figure'),
    elementDef(0, 'img', ['src']),
    elementDef(1, 'legend'),
    textDef()
  ], (checkBinding, view) => {
    var component = view.component;
    const currVal_0 = component.getPonyImageUrl();
    checkBinding(view, 1, currVal_0);
    const currVal_1 = component.ponyModel.name;
    checkBinding(view, 4, currVal_1);
  });
}
```

This change detection function does basically what you would write by hand. It is called by the framework every time it needs to check if there is a change (see the previous chapter). It basically gets the values of the dynamic bits (the `src` attribute and the interpolation), and then calls a function of the framework with the view data, the index of the element to update and the new value. The `checkBinding` function has only one job: it compares the previous value stored for this element and if it changed, updates it, and replaces it with the new value.

This generated code is very fast and can be optimized by JS engines (see the part on monomorphism and inline caching in the previous chapter). On the other end, it takes some time to generate this code when the application starts. This is called the *Just in Time* (JiT) compilation.

To sum up, when you are using the JiT compilation: you write TypeScript code and HTML templates, you compile your TypeScript to JavaScript and send the JS and HTML to your users. At runtime, the HTML is then compiled to JS code too.

20.2. Ahead of Time compilation

But could we generate this code *before* starting the application? And indeed we can, using a compiler that the Angular team wrote: the *Ahead of Time* (AoT) compiler. You can call it manually in your project (`ngc`), or if you are using the CLI (as you should), it is a simple flag to add when you build or serve the app:

```
ng build --aot
```

The build will now use the Angular compiler to compile the templates to TypeScript files. To TypeScript? Yes, because it then allows to check that we didn't make a mistake in our templates! The generated TypeScript code and our application code will then be compiled by TypeScript (`ngc` will call the TypeScript compiler immediatly), and if you made a mistake in a template, you'll see an error:

```
<figure>
  <!-- wrong method name -->
  <img [src]="getPonyImageUr()">
  <legend>{{ ponyModel.name }}</legend>
</figure>
```

The Angular compiler then generates this TypeScript code:

```
(checkBinding, view) => {
  var component: PonyComponent = view.component;
  const currVal_0 = component.getPonyImageUr();
  checkBinding(view, 1, currVal_0);
  const currVal_1 = component.ponyModel.name;
  checkBinding(view, 4, currVal_1);
});
```

which raises an error in the TypeScript compilation:

```
Property 'getPonyImageUr' does not exist on type 'PonyComponent'.
```

This is great, because it means you can check all your templates before even running the application. Your future refactoring will be painless: if you rename a method or a property in a component, you'll know straight away that the template must be updated, too, because it breaks the build.

It also means that this compilation may throw an error whereas your code can be fine in the *Just in Time* mode. For example, if you have a private property in your component used in your template, it will be fine in JiT mode (because JavaScript has no notion of private property), but will break in AoT mode (as the TypeScript code of the ngfactory generated needs to access the property). Right now, there is no really good source of documentation on the additional rules you must respect to be

AoT-compliant, except this repository from Rangle.

Compiling your application before shipping to your users will also greatly speed up the start of the application, as the compilation will already be done!

To sum up, in AoT mode: you still write TypeScript and HTML, you compile the HTML to TypeScript and then all TypeScript code to JavaScript, and send this JS code to your users.

The downside will be the size of the JavaScript bundle you will ship to your users: the generated code is larger than the uncompiled templates. This is somewhat compensated by the fact that you don't need to ship the Angular compiler to your users, as the templates are already compiled. And the compiler is a big piece of code, so this is a nice win. On a medium or large application, this generally doesn't compensate the increase in size from the generated factories though. If you want to have all the perks of the AoT compilation AND a small bundle, you'll need to dig into the lazy-loading feature we explained in the Router chapter.

If you want to dig this topic deeper, there is a great talk on the compiler at ng-conf 2017

Chapter 21. Advanced observables

I must confess that I made a mistake: I under-estimated the value of RxJS and Observables. And that's a bit sad because I did the same with AngularJS 1.x and Promises. Promises were extremely useful in AngularJS 1.x, once you get them, you can handle any asynchronous part of your application elegantly. But it took some time to get there, and there were traps to avoid (check the blog post we wrote about Traps, anti-patterns and tips about AngularJS promises, if you want to learn more about that).

Angular relies on RxJS, and exposes Observables in a few APIs (Http, Forms, Router...). After coding for a while with RxJS, I think this ebook deserves a more "advanced" chapter about Observables, their creation, subscription, operators, possible uses with Angular, etc. I hope this will give you a few hints or spark the curiosity to dive deeper into it, because RxJS will play a big role into how you orchestrate your app, and it can do a wonderful job to simplify your life.

21.1. Subscribe, unsubscribe and async pipe

To sum up what we learned in the chapter about Reactive Programming, an Observable represents a sequence of events to which we can subscribe.

This stream of events can happen at any time, and there can be only one event or ten thousands of them. But there is a distinction to understand between two kinds of Observables: cold ones and hot ones.

Cold observables will only emit events when they are subscribed to. You can think of it as watching a Youtube video: the video will only stream when you hit the "Play" button. For example, the observables returned by the `Http` class are cold observables: they will only trigger the request when you `subscribe`.

Hot observables are slightly different: they emit events from the moment they are created. You can think of them as live television: you turn on the TV and you land in the middle of a show, that can have started minutes or hours ago. The observable representing the `valueChanges` in a `FormControl` is also a hot observable. You will not receive the values emitted before the moment you subscribed, only the value from the moment you subscribe.

When you subscribe to an observable, you can pass three possible parameters:

- a function to handle the next event
- a function to handle an error
- a function to handle the completion

The first one is pretty obvious. The observable is a stream of events and you define what to do if an event occurs.

The second one allows to handle a potential error. It's not always necessary to pass this function. If the stream of events represents a stream of clicks, there are no possible errors that can occur, even if your user broke his finger (this joke is not mine, it's from a great presentation from André Staltz on RxJS at NgEurope 2016). But in most cases, it is very useful to define an error handler. It allows

to define what to do if you get an error response from the HTTP backend, for example.

One thing to understand about this: an error is a terminal event, the observable will not emit new events after it. So if it is important for you to continue to listen, you probably want to handle this properly (we'll come back to this in a minute).

The third function you can pass allows to handle the completion: that's because an observable can finish (your Youtube video is over for example). And sometimes you want to do something if the observable is over, like warning your user, or computing a value…

In our Ponyracer app, a race is represented by an observable, emitting the positions of the ponies. When the race is over, the observable stops emitting events. Then we want to compute which pony won the race, change the UI to reflect that, etc.

But maybe the user won't stay around until the end of the race. What happens then? The component will be destroyed. But, if we subscribed to an observable in that component before the destruction, then the `next` function will continue to do its job every time an event occurs. Even if the component is no longer displayed! That can lead to memory leaks and all sorts of trouble…

So the best practice is to store the subscription returned by the `subscribe` function, and on the destruction of the component, call `unsubscribe` on this subscription (typically in the `ngOnDestroy` method). Wrap the unsubscribe in a check to test if the subscription is present (maybe the subscription is not initialize yet, and then you'll have an error because the object will be undefined when you'll call `unsubscribe`).

This general rule has a few exceptions. You don't necessarily need to do this for observables that will only emit an event then complete, like an HTTP request. And you definitely don't need to do this if you subscribe to one of the router observables, like the `params` observable: the router will clean up for you, yay!

Last, but not least, the `async` pipe. Angular provides a special pipe, called `async`. You can use it in your templates to directly subscribe to an observable.

It has a few advantages:

- you can directly store the observable in your component and don't have to subscribe manually, then store the value emitted in a component's field;
- the `async` pipe will handle the unsubscription for you on the component's destruction;
- it can be used to do some performance magic (more on that later).

It also have a downside: you have to be careful when using this syntax multiple times in a template.

Let me illustrate the last point, in a `RaceComponent`, displaying the race properties:

```
@Component({
  selector: 'ns-race',
  template: `<div>
    <h2>{{ (race | async)?.name }}</h2>
    <small>{{ (race | async)?.date }}</small>
  </div>`
})
export class RaceComponent implements OnInit {

  race: Observable<{} | RaceModel>;
  error: boolean;

  constructor(private raceService: RaceService) {
  }

  ngOnInit() {
    this.race = this.raceService.get()
    // the `catch` operator allows to handle a potential error
    // it must return an observable (here an empty one).
      .catch(error => {
        this.error = true;
        console.error(error);
        return Observable.empty();
      });
  }
}
```

This code sample assumes that the method `raceService.get()` returns an Observable emitting a `RaceModel`. We store this observable in a field named `race` (you'll sometimes see the name `race$` in blog posts, because other frameworks use that naming convention for variables of type Observable). Then we use the `race` observable in the template with the `async` pipe twice: once to display the name and once to display the date. The component's code is quite elegant: you don't have to manually subscribe to the observable.

But maybe you can spot the problem. We call twice the `async` pipe, which means that we subscribe twice to the observable. If the `raceService.get()` method performs an HTTP request to fetch the race details, this request will be performed twice!

A first solution would be to modify the observable to share it between the different subscribers. This can be achieved with the `publishReplay` operator for example:

```
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/empty';
import 'rxjs/add/operator/catch';
import 'rxjs/add/operator/publishReplay';
import { RaceService, RaceModel } from './race.service';

@Component({
  selector: 'ns-race',
  template: `<div>
    <h2>{{ (race | async)?.name }}</h2>
    <small>{{ (race | async)?.date }}</small>
  </div>`
})
export class RaceComponent implements OnInit {

  race: Observable<{} | RaceModel>;
  error: boolean;

  constructor(private raceService: RaceService) {
  }

  ngOnInit() {
    this.race = this.raceService.get()
      // the `catch` operator allows to handle a potential error
      // it must return an observable (here an empty one).
      .catch(error => {
        this.error = true;
        console.error(error);
        return Observable.empty();
      })
      // will share the subscription between the subscribers
      .publishReplay().refCount();
  }
}
```

But, even if that's now correct and does not produce two different HTTP requests, I still don't really like the template. Quite often, the component needs to access the race anyway for some presentation logic, and not only the template. And it has to handle errors, too. A subscription, or a `do()` and/or `catch()` operator in the component is often necessary anyway. So storing the race in a field is not a big burden, and makes the template code simpler.

Note that the 4.0 release introduced an `as` syntax that can solve the problem by subscribing only once and storing the result in a variable of the template:

```
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/empty';
import 'rxjs/add/operator/catch';
import { RaceService, RaceModel } from './race.service';

@Component({
  selector: 'ns-race',
  template: `<div *ngIf="race | async as raceModel">
    <h2>{{ raceModel.name }}</h2>
    <small>{{ raceModel.date }}</small>
  </div>`
})
export class RaceComponent implements OnInit {

  race: Observable<{} | RaceModel>;
  error: boolean;

  constructor(private raceService: RaceService) {
  }

  ngOnInit() {
    this.race = this.raceService.get()
      // the `catch` operator allows to handle a potential error
      // it must return an observable (here an empty one).
      .catch(error => {
        this.error = true;
        console.error(error);
        return Observable.empty();
      });
  }
}
```

This is quite elegant.

Another possible solution is to split our component in two: a smart one (responsible for the race fetching) and a dumb one (that just displays the race received in input).

That would look like:

```
@Component({
  selector: 'ns-racecontainer',
  template: `<div>
    <div *ngIf="error">An error occurred while fetching the race</div>
    <ns-race [raceModel]="race | async"></ns-race>
  </div>`
})
export class RaceContainerComponent implements OnInit {

  race: Observable<{} | RaceModel>;
  error: boolean;

  constructor(private raceService: RaceService) {
  }

  ngOnInit() {
    this.race = this.raceService.get()
    // the `catch` operator allows to handle a potential error
    // it must return an observable (here an empty one).
      .catch(error => {
        this.error = true;
        console.error(error);
        return Observable.empty();
      });
  }
}

@Component({
  selector: 'ns-race',
  template: `<div>
    <h2>{{ raceModel.name }}</h2>
    <small>{{ raceModel.date }}</small>
  </div>`
})
export class RaceComponent {

  @Input() raceModel: RaceModel;

}
```

This pattern can be useful in some parts of your application, but, like every pattern, you don't have to use it everywhere. It's sometimes not a big deal to have only one component that does both. Other times, you'll want to extract a dumb component to reuse it in another part of your application for example, so two components will make sense.

21.2. Leveraging operators

We saw a few operators until then, but I'd like to take a moment to describe a few others in a step by step example. We are going to code a typeahead input. A typeahead allows your users to enter a

text in an input, and then the application displays a few suggestions based on this text (like a Google search box).

A good typeahead has quite a few features:

- it displays the results matching the request (obviously)
- it allows to only display results if the input text is at least a few characters long
- it won't fetch the results for every keystroke from our user, but will wait for some time to make sure the user is done typing
- it won't trigger the same request twice if the user enters the same value

All this can be done by hand, but it's far from being trivial. But we're in luck, Angular and RxJS combine nicely to solve this kind of problem!

First let's see what a component like this would look like:

```
import { Component, OnInit } from '@angular/core';
import { FormControl } from '@angular/forms';
import { PonyService, PonyModel } from './pony.service';

@Component({
  selector: 'ns-typeahead',
  template: `<div>
    <input [formcontrol]="input">
    <ul>
      <li *ngFor="let pony of ponies">{{ pony.name }}</li>
    </ul>
  </div>`
})
export class PonyTypeAheadComponent implements OnInit {
  input = new FormControl();
  ponies: Array<PonyModel> = [];

  constructor(private ponyService: PonyService) {}

  ngOnInit() {
    // todo: do something with the input
  }
}
```

In the `ngOnInit` method, we can start by subscribing to the `valueChanges` observable exposed by the `FormControl` (check the chapter on forms if you need to refresh your memory).

```
this.input.valueChanges
  .subscribe(value => console.log(value));
```

Next we want to use this value to fetch the ponies matching the given input. Our `PonyService` has a

method `search` that does exactly that! We can suppose that this method does an HTTP request behind the scenes to fetch the results from the server, so it returns an `Observable<Array<PonyModel>>`, an observable that emits arrays of ponies.

Let's subscribe to this method to update the `ponies` field of our component:

```
this.input.valueChanges
  .subscribe(value => {
    this.ponyService.search(value)
      .subscribe(results => this.ponies = results);
  });
```

OK, that works. But when I see something like this, it reminds me of Promises and nested `then` calls, that can be flattened. And indeed you can do the same with Observables, with the `concatMap` operator for example:

```
this.input.valueChanges
  .concatMap(value => this.ponyService.search(value))
  .subscribe(results => this.ponies = results);
```

Wow, much more elegant! `concatMap` "flattens" our code. It replaces every event emitted by the source observable (i.e. the entered pony name) by the events emitted by the observable of ponies. But it's not the perfect operator for this situation. As our `search` method performs an HTTP request per search, we can run into some troubles if a request is too slow. Our user might query `n` then `ni`, but the result might come back really slowly for `n`, and fast for `ni`. That means our code above will display the second results only after the first displayed, even if we don't care about the first ones anymore! This could be tracked by hand, but that would be really cumbersome.

RxJS provides a super useful operator for this use-case: `switchMap`. Unlike `concatMap`, `switchMap` will only care about the last value emitted, and will discard the earlier values, so we're sure that the results corresponding to an old input won't be displayed.

```
this.input.valueChanges
  .switchMap(value => this.ponyService.search(value))
  .subscribe(results => this.ponies = results);
```

OK, now let's discard the queries that are less than three characters. Easy: we just have to use a `filter` operator!

```
this.input.valueChanges
  .filter(query => query.length >= 3)
  .switchMap(value => this.ponyService.search(value))
  .subscribe(results => this.ponies = results);
```

We also don't want to search immediately after a keystroke: we'd like to search only after the user

stops typing for 400ms for example. Yep, you guessed it: there's an operator for that, and it's called `debounceTime`:

```
this.input.valueChanges
  .filter(query => query.length >= 3)
  .debounceTime(400)
  .switchMap(value => this.ponyService.search(value))
  .subscribe(results => this.ponies = results);
```

So now a user can enter a value, delete some character, add others and the query will only fire when 400ms have passed since the last keystroke. But what if the user enters "Rainbow", waits for 400ms (which will thus send a request), then enters "Rainbow Dash" and immediately removed the "Dash" to get back to "Rainbow"? That would send two subsequent requests for "Rainbow"! Maybe we can only trigger a request if the query is different than the last one? Of course we can, with `distinctUntilChanged`:

```
this.input.valueChanges
  .filter(query => query.length >= 3)
  .debounceTime(400)
  .distinctUntilChanged()
  .switchMap(value => this.ponyService.search(value))
  .subscribe(results => this.ponies = results);
```

Last thing: we need to properly handle the errors. We know that the `valueChanges` will not emit any error, but our `ponyService.search()` observable might throw as it is dependent on the network. And the problem with observables is that an error will completely break the stream: so if one request blows, the whole typeahead will be down… We don't want that, so let's catch potential errors:

```
this.input.valueChanges
  .filter(query => query.length >= 3)
  .debounceTime(400)
  .distinctUntilChanged()
  .switchMap(value => this.ponyService.search(value).catch(error => Observable.
of([])))
  .subscribe(results => this.ponies = results);
```

Quite nice, don't you think? We now only trigger a search when the user enters a text with more then 3 characters and waits at least 400ms. We guarantee that we don't trigger the same request twice, and that the results are always in sync with the request! All that in 5 lines of code. Good luck doing the same by hand without adding any issue…

This is of course a really good use-case for RxJS, but the point is that it provides a lot of operators, with some gems hidden in it. It takes time to understand it, but it's worth the trouble as it can be tremendously useful in your application.

21.3. Building your own Observable

Sometimes, sadly, you need to use libraries that produce events but not using an Observable. All hope is not lost, because you can of course create your own Observables, using, for example, the `Observable.create(observer ⇒ {})` method.

The method passed as a parameter is called the `subscribe` function: it will be responsible for emitting events and errors, and to complete when done.

For example, if you want to create an Observable which emits 1, then 2, then completes, you could do:

```
const numbers = Observable.create(observer => {
  observer.next(1);
  observer.next(2);
  observer.complete();
});
```

Now we could subscribe to such an observable:

```
numbers.subscribe(
  number => console.log(number),
  error => console.log(error),
  () => console.log('Complete!')
);
// Will log:
// 1
// 2
// Complete!
```

Now, let's say I want to emit 'hello' every 2 seconds, and never complete. We could easily do this with some built-in operators, but we can try by hand, as a small example:

```
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/create';


export class HelloService {
  get(): Observable<string> {
    return Observable.create(observer => {
      const interval = setInterval(() => observer.next('hello'), 2000);
    });
  }
}
```

The callback function passed to `Observable.create()` can also return a function that will be called on the unsubscription. That's really useful if you have some cleanup to do. Like us with our

`HelloService`, because we'll need to stop the `setInterval` when the observable will be unsubscribed.

```
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/create';


export class HelloService {
  get(): Observable<string> {
    return Observable.create(observer => {
      const interval = setInterval(() => observer.next('hello'), 2000);
      return () => clearInterval(interval);
    });
  }
}
```

The interval will not be created until the subscription, so we just created a cold observable.

I hope you enjoyed this small chapter on observables. They can also be use to sequence your HTTP requests, or to communicate between components (more on this soon). But you have now a good overview of what's possible!

PRACTICE

We have several exercises that leverage RxJS and let you discover a lot of operators:

- Display the user 🦄
- Logged home 🦄
- Remember me 🦄
- Logout 🦄
- Observable tips and tricks 🦄
- Boost a pony 🦄

Chapter 22. Advanced components and directives

22.1. View queries

In the Template chapter, we talked about a nice feature called "local variables", allowing to get a reference to a DOM element in the template. For example, you can give the focus to an input with a button easily:

```
<input #myInput>
<button (click)="myInput.focus()">Focus</button>
```

We also saw this same feature in the Forms chapter for example, when we wanted to grab a reference to a specific directive:

```
<input name="login" [(ngModel)]="user.login" required #loginCtrl="ngModel">
<div *ngIf="loginCtrl.dirty && loginCtrl.hasError('required')">
  The login field is required
</div>
```

What if we need to have these references in our component code, and not only in the template? That where "view queries" enter the scene and can save the day!

For example, you may want to focus an input as soon as your component is displayed. To do so, we need to grab a reference to the input, using the `ViewChild` decorator.

```
@Component({
  selector: 'ns-login',
  template: `
    <input #loginInput name="login" [(ngModel)]="credentials.login" required>
  `
})
export class LoginComponent implements AfterViewInit {

  credentials = { login: '' };

  @ViewChild('loginInput') private loginInput: ElementRef;

  ngAfterViewInit() {
    this.loginInput.nativeElement.focus();
  }

}
```

We declare a field called `loginInput`, and we decorate it with `ViewChild`. This decorator needs a

selector as parameter: here we use the local variable declared in our template. The decorator indicates to the framework that it needs to query the template to find an element with this local variable name. The field will be initialized with this element, of type `ElementRef`. This type has only one field, `nativeElement`, which is a reference on the underlying DOM element.

The example also showcases a nice use of the lifecycle method `ngAfterViewInit`. This method is called as soon as the view is created, so you are sure that the element you are waiting for is indeed present. If you try to do the same in `ngOnInit`, it won't work. There is also another method called `ngAfterViewChecked`, which is called every time the view is checked (after each change detection).

If you wanted to have this feature in a lot of different components, you could create a directive for this, instead of duplicating the code in each component.

```
@Directive({
  selector: '[nsFocus]'
})
export class FocusDirective implements AfterViewInit {

  ngAfterViewInit(): void {
  }

}
```

This directive does nothing yet, but it would be used like this in a template:

```
<input nsFocus>
```

The directive needs to access its host element to give it the focus. That's where `ElementRef` is interesting, as it can be injected in our directive:

```
@Directive({
  selector: '[nsFocus]'
})
export class FocusDirective implements AfterViewInit {

  constructor(private element: ElementRef) { }

  ngAfterViewInit(): void {
    this.element.nativeElement.focus();
  }

}
```

And we're done: using this directive will give the focus to its host element!

NOTE	Accessing directly the `nativeElement` like in these examples won't work if you decide to run your application in another platform than the browser, like in Web Workers or on the server.

Let's go back to our `ViewChild` decorator: it can also accept a type as a selector.

For example, in the Forms chapter, we saw that to submit a form in the template-driven way, you can use two-way binding, or you can grab a reference to the form and pass its value to the submit method:

```
<form (ngSubmit)="authenticate(form.value)" #form="ngForm">
  <!-- ... -->
</form>
```

But we can also use a `ViewChild` for this:

```
@Component({
  selector: 'ns-login',
  template: `
    <form (ngSubmit)="authenticate()">
      <!-- ... -->
    </form>
  `
})
export class LoginFormComponent {

  @ViewChild(NgForm) credentialsForm: NgForm;

  authenticate() {
    console.log(this.credentialsForm.value);
  }

}
```

The cool thing with `ViewChild` is that it is a dynamic query: it will always be up to date with the template. If the element queried is destroyed, the field will be `undefined`.

This decorator also has a twin called `ViewChildren`. Unlike `ViewChild` which will get a reference to one field matching the selector (the first if there are several ones), `ViewChildren` will get a reference to all the fields. It returns a `QueryList`, a type with a few useful attributes:

- `first` returns the first element matching the query
- `last` returns the last element matching the query
- `length` returns how many elements are matching the query
- `changes` returns an observable that will emit the new `QueryList` every time an element matching the query is added, removed or moved.

Let's say we have a `RaceComponent` displaying a list of `PonyComponent`, we can easily be notified every time a `PonyComponent` is added or removed:

```
@Component({
  selector: 'ns-race',
  templateUrl: './race.component.html'
})
export class RaceComponent implements AfterViewInit {

  @Input() raceModel: RaceModel;

  @ViewChildren(PonyComponent) ponies: QueryList<PonyComponent>;

  ngAfterViewInit() {
    this.ponies.changes
    // this will log how many ponies are displayed in the component
      .subscribe(newList => console.log(newList.length));
  }

}
```

`QueryList` has also a lot of methods available like `toArray()`, `map()`, `filter()`, `find()`...

22.2. Content

Another common thing that we usually need as developers is the ability to build UI components whose content will be dynamic.

For example, let's say you want to build a "card" component using the Bootstrap 4 CSS framework. The template of such a card looks like this:

```
<div class="card">
  <div class="card-block">
    <h4 class="card-title">Card title</h4>
    <p class="card-text">Some quick example text</p>
  </div>
</div>
```

You can of course duplicate this HTML every time you need it in your application. But at this point of your read, you are probably thinking about building a component. Two parts are dynamic in the card (the title and the content), so this is probably what you will come up with:

```
@Component({
  selector: 'ns-card',
  template: `<div class="card">
    <div class="card-block">
      <h4 class="card-title">{{ title }}</h4>
      <p class="card-text">{{ text }}</p>
    </div>
  </div>`
})
export class CardComponent {

  @Input() title: string;
  @Input() text: string;

}
```

And then use it like this:

```
<ns-card title="Card title" text="Some quick example text"></ns-card>
```

This works perfectly. But looking more closely to your need, you realize that the content of the card can also be complex HTML, and not just text, which is supported by Bootstrap!

Of course, Angular has your back, and it's easy to "pass" HTML to a child component, thanks to `<ng-content>`. That's what we used to call "transclusion" in AngularJS 1.x.

`ng-content` is a special tag you can use in your templates to include HTML provided by the parent component:

```
<div class="card">
  <div class="card-block">
    <h4 class="card-title">{{ title }}</h4>
    <p class="card-text">
      <ng-content></ng-content>
    </p>
  </div>
</div>
```

And you can now use the component like this:

```
<ns-card title="Card title">
  Some quick <strong>example</strong> text
</ns-card>
```

Later, you realize that the title can also be some complex HTML. Of course, there is a way to pass multiple contents to the card component, using multiple `ng-content` with a selector.

```
<div class="card">
  <div class="card-block">
    <h4 class="card-title">
      <ng-content select=".title"></ng-content>
    </h4>
    <p class="card-text">
      <ng-content select=".content"></ng-content>
    </p>
  </div>
</div>
```

and use it like this:

```
<ns-card>
  <p class="title">Card <strong>title</strong></p>
  <p class="content">Some quick <strong>example</strong> text</p>
</ns-card>
```

This will produce the following result:

```
<div class="card">
  <div class="card-block">
    <h4 class="card-title">
      <p class="title">Card <strong>title</strong></p>
    </h4>
    <p class="card-text">
      <p class="content">Some quick <strong>example</strong> text</p>
    </p>
  </div>
</div>
```

22.3. Content queries

When you are using these `ng-content` tags, the projected content will not be queried by `ViewChild` or `ViewChildren`. For these contents, you have to use two other decorators: `ContentChild` and `ContentChildren`.

Let's say you are building another UI component based on Bootstrap 4, this time a "tabs" component. The HTML must look like this according to the docs:

```
<ul class="nav nav-tabs">
  <li class="nav-item">
    <a class="nav-link">Races</a>
  </li>
  <li class="nav-item">
    <a class="nav-link">About</a>
  </li>
</ul>
```

But we would like to offer a nice component to our team, something like:

```
<ns-tabs>
  <ns-tab title="Races"></ns-tab>
  <ns-tab title="About"></ns-tab>
<ns-tabs>
```

We need an outer Tabs component, which must find out how many `ns-tab` directives are embedded inside the component template, iterate through each of them and generate the appropriate markup.

To do so, let's start by creating a directive `TabDirective`:

```
@Directive({
  selector: 'ns-tab'
})
export class TabDirective {
  @Input() title: string;
}
```

The directive doesn't do much: it only has an input to get the tab title. Note that we are using an element as the selector, `ns-tab`.

Now we need to build the `TabsComponent`:

```
@Component({
  selector: 'ns-tabs',
  template: `
  <ul class="nav nav-tabs">
    <li class="nav-item" *ngFor="let tab of tabs">
      <a class="nav-link">{{ tab.title }}</a>
    </li>
  </ul>`
})
export class TabsComponent {
  @ContentChildren(TabDirective) tabs: QueryList<TabDirective>;
}
```

As you can see, the template iterates through an array of tabs to generate an `li` element for each of them. But where does this `tabs` array come from? How can the component know about the two `ns-tab` directives embedded inside the ns-tabs component? That's what the `ContentChildren` directive allows doing.

To grab the list of the tabs, we need to use `ContentChildren`, here with `TabDirective`. This gives us an iterable list of tabs, that you can use in an `NgFor`. As each element of this list is a `TabDirective`, we can then access the public property `title`, and display the tab's title!

Note that if, for whatever reason, you had a template like this one:

```
<ns-tabs>
    <div>
      <ns-tab title="Races"></ns-tab>
    </div>
    <ns-tabgroup>
      <ns-tab title="About"></ns-tab>
    </ns-tabgroup>
  <ns-tabs>
```

Then the `QueryList` in `TabComponent` will only contain the first `TabDirective`. `ContentChild` and `ContentChildren` are indeed only looking for direct descendants, and will stop at the `ns-tabgroup` component.

If we want our component to still work with this, there is an option you can add to your decorator:

```
@Component({
  selector: 'ns-tabs',
  template: `
  <ul class="nav nav-tabs">
    <li class="nav-item" *ngFor="let tab of tabs">
      <a class="nav-link">{{ tab.title }}</a>
    </li>
  </ul>`
})
export class TabsWithDescendantsComponent {

  @ContentChildren(TabDirective, { descendants: true })
  tabs: QueryList<TabDirective>;

}
```

Now it finds all the `TabDirective` again!

The `tabs` field is a `QueryList`, so you can also subscribe to the changes, like we saw for `ViewChildren`. Once again, the `QueryList` is not accessible in the component constructor or even in `ngOnInit`. To be sure that the content can be queried (i.e. that the `QueryList` is properly populated), use the `ngAfterContentInit` lifecycle hook. You can also use the `ngAfterContentChecked` hook, which is called

every time the content is checked.

22.4. Host listener

When writing a directive, it can be fairly common to interact with the host element.

Let's take a simple example: our customer wants to easily clear the content of some text inputs by double-clicking on them. This is the kind of behavior that you can encapsulate in a custom directive, let's say `InputClearDirective`. Its selector will be an attribute, let's say `nsInputClear`. When this attribute is added to an element, we want to listen for a double-click on this host element.

NOTE	This example is simple, but not very realistic. More realistic (but also more complex) applications of this feature would be, for example, to display a tooltip or a popover when an element is being hovered or clicked

Let's create the directive:

```
@Directive({
  selector: '[nsInputClear]'
})
export class InputClearDirective {

  constructor(private element: ElementRef) {}

}
```

And use our directive like this:

```
<input nsInputClear>
```

We now need to react on a 'dblclick' event on our host element (here, the `input`) to clear the value.

That's where we can use the `HostListener` decorator! This decorator can be added to a method of a directive, to indicate to the framework that this method needs to be called if the event given as parameter to the decorator is triggered on the host element.

In our case, we can write:

```
@Directive({
  selector: '[nsInputClear]'
})
export class InputClearDirective {

  constructor(private element: ElementRef) {}

  @HostListener('dblclick')
  clearContent() {
    this.element.nativeElement.value = '';
  }

}
```

Now, every time a 'dblclick' event is triggered on the host element, the directive will clear the input value.

Note that it is also possible to listen to global events, like `window:resize` for example:

```
@Directive({
  selector: '[nsWindowResize]'
})
export class WindowResizeDirective {

  @HostListener('window:resize', ['$event'])
  resize(event) {
    console.log(`The screen is being resized to ${event.target.innerWidth}`);
  }

}
```

Also note that you can access the event in the method, by specifying `['$event']` as a second parameter of the decorator.

22.5. Host binding

Another decorator available to create advanced directives and components is `HostBinding`. Whereas `HostListener` allows to interact with the events on the host element, `HostBinding` allows to interact with the properties of the host element.

Let's say we want to add a specific CSS class (`is-required`) to an input if this input has a specific validation error (`required`). Maybe this class adds a nice border around the input, or a small asterisk, that's not really important. This will not validate the field in any way, it will simply grab the result of the built-in Angular form validation, and use this result to style the input.

This is, again, a task that perfectly fits a directive:

```
@Directive({
  selector: '[nsAddClassIfRequired]'
})
export class AddClassIfRequiredDirective {

}
```

We will use it in a code-driven Angular form:

```
<input formControlName="firstName" nsAddClassIfRequired>
```

or in a template driven one:

```
<input [(ngModel)]="user.name" required nsAddClassIfRequired>
```

We then need to grab a reference to the status of the input in our directive. Angular does automatically validate the fields, and will add the `required` error if the field is required and not filled. That's where the powerful dependency injection system can help us! You can indeed ask Angular to inject into a directive another directive applied to the same host, or to one of its ancestors.

As we want our directive to work with `FormControlName` or `NgModel`, we could ask Angular to inject these two. But it would break as only one of the two will be available (as you generally either use one or the other on a given input), and Angular breaks if a dependency can't be provided. There is a trick that allows Angular to continue even if a dependency can't be provided: the `Optional` decorator.

So something like this could work:

```
constructor(
  @Optional() private formControl: FormControlName,
  @Optional() private ngModel: NgModel
) {}
```

But that's not the best we can do. Indeed, both directives inherits the same base class: `NgControl`. So instead of injecting one or the other, we can simply ask Angular to provide us the common `NgControl`!

```
@Directive({
  selector: '[nsAddClassIfRequired]'
})
export class AddClassIfRequiredDirective {

  constructor(private control: NgControl) {}

}
```

Now that we have a reference to the `NgControl`, it's easy to know if the field has the `required` error, by using its `hasError()` method. The last step requires us to add the class `is-required` to our host element if that's the case. That's where the `HostBinding` decorator enters the scene! This decorator allows to bind a property of the host element to a field of our directive. So it is generally used like this:

```
@HostBinding('value') innerValue;
```

And that would automatically update the host's value, every time the `innerValue` property of the directive would change.

In our case, we don't have a field to bind to. But we can define a getter that returns true or false, depending on the field's error, and decorate this getter with `HostBinding` to automatically add or remove the class `is-required`:

```
@Directive({
  selector: '[nsAddClassIfRequired]'
})
export class AddClassIfRequiredDirective {

  constructor(private control: NgControl) {}

  @HostBinding('class.is-required')
  get isRequired() {
    return this.control.hasError('required');
  }

}
```

These few lines of code are really powerful: every time the directive is used in a form, Angular will automatically add or remove our custom class depending on the new value entered by the user!

It can be used to bind other kinds of properties, not just CSS classes. For example, some component libraries use it to add accessibility attributes (`aria.xxx`) to a host element.

Note that the directive we built has a custom selector, but if you decide that you want to apply these directives to every input, you can change their selector to `input`, and they will automatically be applied on every input of your application.

Chapter 23. Internationalization

Alors comme ça, tu veux internationaliser ton application?

OK, don't worry if you didn't understand anything to this French introduction. Your role as a developer, fortunately, is not to translate your application in French, Spanish, or whatever other language. What you can do, though, is to allow this to happen. This chapter explains how to achieve that.

23.1. The locale

We already mentioned internationalization before, in the chapter about pipes. Three of the built-in Angular pipes deal with internationalization, and use the standard JavaScript Internationalization API, which is supposed to be provided by the browser. Those are the pipes `number`, `currency` and `date`.

What we don't know yet is how these three pipes decide how to format the numbers and dates. Should they use a dot or a comma as decimal separator? Should they use *January* or *Janvier* for the first month of the year? You might think that this is decided based on the preferred language configured in the browser, but actually, it's not. This depends on an injectable value named `LOCALE_ID`. And the default value of `LOCALE_ID` is `'en-US'`.

Here is an example showing how to get the value of `LOCALE_ID`. As you can see, it's a simple string value. To inject it into your components or services, you can't just rely on its type. You need to tell Angular which token identifies the value, using `@Inject(LOCALE_ID)`. This can be useful if your logic needs to know which locale the application is using.

```
@Component({
  selector: 'ns-locale',
  template: `
    <p>The locale is {{ locale }}</p>
    <!-- will display 'en-US' -->

    <p>{{ 1234.56 | number }}</p>
    <!-- will display '1,234.56' -->
  `
})
class DefaultLocaleComponent {
  constructor(@Inject(LOCALE_ID) public locale: string) { }
}
```

This is good. But how can we change the locale? Actually, you can't. The locale is a constant, that you can't change once the application has started. But that doesn't mean you can't set it to another value *before* the application starts. This is possible, simply by providing another value for the `LOCALE_ID` token in your root Angular module.

Here's an example showing how to do that, and its effect on our component:

```
@NgModule({
  imports: [BrowserModule],
  declarations: [CustomLocaleComponent], // and other components
  providers: [
    { provide: LOCALE_ID, useValue: 'fr-FR'}
  ]
  // ...
})
export class AppModule { }

@Component({
  selector: 'ns-locale',
  template: `
    <p>The locale is {{ locale }}</p>
    <!-- will display 'fr-FR' -->

    <p>{{ 1234.56 | number }}</p>
    <!-- will display '1 234,56' -->
  `
})
class CustomLocaleComponent {
  constructor(@Inject(LOCALE_ID) public locale: string) { }
}
```

If you want to create an application that uses only one locale, but different from `'en-US'`, then setting the locale as explained above is all you need to do. But often, this is not enough, and you want to really internationalize your application.

23.2. Translating text

If you have used AngularJS 1.x before, and have internationalized your AngularJS application, you probably know that there is nothing built-in to display translated text based on the preferred language of the user.

One of the popular libraries to achieve that with AngularJS is angular-translate. The strategy it uses is fairly common: you use a directive or a pipe to translate a key (for example `'home.welcome'`). This key identifies a message, and you provide the translations for all the languages you want to support (for example: 'Welcome' and 'Bienvenue'). At runtime, the directive or pipe uses the preferred language to get the appropriate translation, and updates the DOM with the translated message. You can change the preferred language at runtime, and all the messages on the page are immediately translated to the new language.

Internationalization is now provided by Angular directly (although it was hardly usable before version 4.0). No need for an external dependency anymore. And it uses basically the same strategy based on keys, but with a big difference: this happens at compile-time rather than happening at runtime. When the application starts, or, if you use the Angular AOT compiler, when you build your app, Angular parses all the HTML templates of your components, and transforms them to JavaScript code that, basically, analyses the changes in the model and modifies the DOM

accordingly. This is when the translation happens. That has important consequences:

- you can't change the locale and the translated messages at runtime. The whole application needs to be reloaded and restarted to do that;
- once started, the application is faster, since it doesn't have to dynamically translate the keys again and again;
- if you use the AOT compiler (and you should, at least in production), you must build and serve as many applications as locales that you want to support.

23.3. Process and tooling

In the remaining parts of this chapter, we will assume that you use Angular CLI to build your application. The tools are actually available and usable outside of Angular CLI, because they're part of the Angular `ngc` compiler. But since they're well integrated and simple to use in Angular CLI, and since it's the recommended way to build your applications anyway, we will use that.

We will also use the AOT compiler to build and serve internationalized versions of our application. This is indeed the simplest way to do that. If you want to test the internationalization in JIT mode (i.e. without precompiling the templates at build time), it requires a substantial modification of the code used to bootstrap the application. Please refer to the internationalization cookbook in the official documentation if you really want to do that.

That said, how do we proceed? You already know how to create components and write their templates. Will you have to rewrite everything to internationalize them? Thankfully, no. The process is the following one:

1. you mark the parts of the templates that need to be translated using the `i18n` attribute;
2. You run a tool to extract all those marked parts into a file, for example `messages.xlf`. Two file formats, both xml-based and industry-standard, are supported;
3. You ask a competent translator to create a translated version of this file, for example `messages.fr.xlf`
4. You build your application by providing the locale ID (`'fr'` for example) and the file containing the translations (`messages.fr.xlf`). The angular compiler and the CLI replace all the i18n-marked parts of the templates by the translations found in the file, and configures your application to use the provided locale ID.

Let's examine each of those steps in more details.

23.3.1. Marking text with i18n and extracting

Let's start with an example template:

```
<h1>Welcome to Ponyracer</h1>
<p>Welcome to Ponyracer {{ user.firstName }} {{ user.lastName }}!</p>
<img src="/img/pony.gif" alt="running pony" title="Ponies are cool, aren't they?" />
Let's start playing.
```

There are 5 text snippets that need to be translated in this template. Of course, you could imagine translating everything at once, but in a more realistic example, that would expose a lot of HTML boilerplate to the translators, and you don't want them to translate everything again when the HTML structure changes. So you should translate the 5 snippets separately.

One of them, the body of the `h1` element, is purely static text. One of them is a text containing two interpolated expressions. Two are attributes of an HTML element. The last one is static text that is not embedded in any element.

Here's how you would mark them. Let's start with the first, simplest one:

```
<h1 i18n>Welcome to Ponyracer</h1>
```

Now let's extract our very first messages file, using the xi18n command provided by Angular CLI:

```
ng xi18n --output-path src/locale
```

This will create the file `messages.xlf` in the `src/locale` directory. Here's what it contains:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file source-language="en" datatype="plaintext" original="ng2.template">
    <body>
      <trans-unit id="5e3335d7f1a430ef14a91507531838c57138b7f2" datatype="html">
        <source>Welcome to Ponyracer</source>
        <target/>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app.component.ts</context>
          <context context-type="linenumber">2</context>
        </context-group>
      </trans-unit>
    </body>
  </file>
</xliff>
```

As you can see, it generates a `trans-unit` containing, as the source, our static text. The role of the French translator will be to provide a `messages.fr.xlf` file looking like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file source-language="en" datatype="plaintext" original="ng2.template">
    <body>
      <trans-unit id="5e3335d7f1a430ef14a91507531838c57138b7f2" datatype="html">
        <source>Welcome to Ponyracer</source>
        <target>Bienvenue dans Ponyracer</target>
      </trans-unit>
    </body>
  </file>
</xliff>
```

This is easy enough, because the source message is easy to understand. You don't need too much context to know what it is about, and how to translate it. But this way of doing things has a big disadvantage. If you change the source code of the template and introduce meaningless white spaces for example, or a dot at the end of the title, here's what happens when extracting the file again:

```
<h1 i18n>
  Welcome to Ponyracer.
</h1>
```

```
      <trans-unit id="6e37e34598c734b3649ba478cac5f2d29e67c331" datatype="html">
        <source>
  Welcome to Ponyracer.
</source>
        <target/>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app.component.ts</context>
          <context context-type="linenumber">5</context>
        </context-group>
      </trans-unit>
```

Not only does the source change, which is expected, but the id also does. That really makes maintaining the translated messages files more difficult than it should. Fortunately, there's a better way. You can provide a unique ID by yourself:

```
<h1 i18n="@@home.title">Welcome to Ponyracer</h1>
```

Which generates:

```
<trans-unit id="home.title" datatype="html">
  <source>Welcome to Ponyracer</source>
  <target/>
  <context-group purpose="location">
    <context context-type="sourcefile">src/app.component.ts</context>
    <context context-type="linenumber">10</context>
  </context-group>
</trans-unit>
```

And in fact, in order to provide more context to your translators, you can provide a meaning and a description in addition to the message ID:

```
<h1 i18n="welcome title|the title of the home page@@home.fullTitle">Welcome to
Ponyracer</h1>
```

```
<trans-unit id="home.fullTitle" datatype="html">
  <source>Welcome to Ponyracer</source>
  <target/>
  <context-group purpose="location">
    <context context-type="sourcefile">src/app.component.ts</context>
    <context context-type="linenumber">13</context>
  </context-group>
  <note priority="1" from="description">the title of the home page</note>
  <note priority="1" from="meaning">welcome title</note>
</trans-unit>
```

Let's move to the second snippet now:

```
<p i18n="@@home.welcome">Welcome to Ponyracer {{ user.firstName }} {{ user.lastName
}}!</p>
```

Here is what it generates:

```
<trans-unit id="home.welcome" datatype="html">
  <source>Welcome to Ponyracer <x id="INTERPOLATION"/> <x id="INTERPOLATION_1"/>
!</source>
  <target/>
  <context-group purpose="location">
    <context context-type="sourcefile">src/app.component.ts</context>
    <context context-type="linenumber">16</context>
  </context-group>
</trans-unit>
```

As you can see, this format has several interesting features:

- there is no way a translator could mess up with the content of the angular expressions, since they are not present in the message;
- it's not clear what these two interpolations are. So it might be a good idea to explain that in the description of the message;
- if, in some language, the last name should come before the first name, the translator is free to reorder the interpolations;
- if the developer chooses to rename the attributes of the component or of the user, nothing will have to be re-translated.

Let's proceed with the two attributes in the `img` element. The syntax to translate attributes is the following:

```
<img src="/img/pony.gif"
     alt="running pony" i18n-alt="@@home.ponyImage.alt"
     title="Ponies are cool, aren't they?" i18n-title="@@home.ponyImage.title"/>
```

That generates the following translation units:

```
<trans-unit id="home.ponyImage.alt" datatype="html">
  <source>running pony</source>
  <target/>
  <context-group purpose="location">
    <context context-type="sourcefile">src/app.component.ts</context>
    <context context-type="linenumber">20</context>
  </context-group>
</trans-unit>
<trans-unit id="home.ponyImage.title" datatype="html">
  <source>Ponies are cool, aren&amp;apos;t they?</source>
  <target/>
  <context-group purpose="location">
    <context context-type="sourcefile">src/app.component.ts</context>
    <context context-type="linenumber">21</context>
  </context-group>
</trans-unit>
```

Finally, how to translate the last snippet? There is no element where we could place an `i18n` attribute. There are two ways to solve the problem: use a comment, or use an `ng-container` element, which won't be rendered in the DOM at runtime:

```
<!--i18n: @@home.startMessage -->Let's start playing.<!--/i18n-->

or

<ng-container i18n="@@home.startMessage">Let's start playing.</ng-container>
```

My preference goes for the second one, which is more consistent.

23.3.2. Translating, building and deploying the application

Now that you generated a complete `messages.xlf` file, someone needs to translate it.

CAUTION	a common mistake is to just replace the original source text (in English in our case) by its translation. That won't work. The translation must be written inside the `<target>` element of each translation unit. The `<source>` element should be kept untouched: it provides the original message that must be translated. For example:

```
<trans-unit id="home.welcome" datatype="html">
  <source>Welcome to Ponyracer <x id="INTERPOLATION"/> <x id="INTERPOLATION_1"/>
!</source>
  <target>Bienvenue dans Ponyracer <x id="INTERPOLATION"/> <x id="INTERPOLATION_1"
/> !</target>
</trans-unit>
```

To run or build the application in French, you need to specify the locale, and the location of the messages file, to `ng serve` or `ng build`:

```
# to run:
ng serve --aot --locale fr --i18n-file src/locale/messages.fr.xlf

#to build:
ng build --aot --locale fr --i18n-file src/locale/messages.fr.xlf
```

The AOT compiler will locate all the i18n-marked snippets in the templates, find the corresponding translations in the XLF file, and transform the snippets in the template using the translations. It will then proceed as usual to generate JavaScript code from the templates and bundle your application.

If you want to support English, French and Spanish, for example, you'll have to build your app three times (once for each locale), and deploy the 3 built applications to your production web server. You will also need to decide which application to serve to which user. This can be done at server-side, by detecting the preferred locale from the request header and serving the appropriate `index.html` page. Or by getting the preferred locale of the authenticated user from the database and serving the appropriate index.html page. You could also do it at client-side, by serving your three applications on three different URLs (`ponyracer.com`, `ponyracer.fr` and `ponyracer.es`, or `ponyracer.com/en`, `ponyracer.com/fr` and `ponyracer.com/es`), and by redirecting from `ponyracer.com` to the correct URL based on the browser locale.

23.4. Translating messages in the code

Sometimes, the text you need to translate is not in the templates, but is in the TypeScript code itself. For example, the three states of a pony race PENDING, RUNNING and FINISHED should be

translated somehow. The current plan is to be able to do something like the following.

```
const RaceStatus = {
  PENDING: __('pending'),
  RUNNING: __('running'),
  FINISHED: __('finished')
}
```

Unfortunately, this is not implemented yet. A temporary solution can be, for example, to load JSON translation files from the server based on the LOCALE_ID.

23.5. Pluralization

Sometimes, the message you want to display depends on the number of elements in a collection, or on a count of elements.

For example, let's say our home page displayed the number of planned races for the day. You could simply show *"Number of planned race(s): 4"*. But a friendlier message would be *"No race is planned"* if there is none, or *"Only one race is planned"* if there is just one, or *"N races are planned"* in the other cases.

Angular actually has a special template syntax to do that. It's hard to read, except maybe for LISP programmers, but it does the job and is fairly easy to understand with the following example. Suppose our component has a property `racesPlanned`, containing the number of races that are planned for today. You can display it as:

```
<p>Hello, {racesPlanned, plural, =0 {no race is planned}
                                 =1 {only one race is planned}
                                 other {{{ racesPlanned }} races are planned}}.</p>
```

To internationalize this message, you would just use the i18n attribute as usual:

```
<p i18n="@@home.racesPlanned">
  Hello, {racesPlanned, plural, =0 {no race is planned}
                                =1 {only one race is planned}
                                other {{{ racesPlanned }} races are planned}}.
</p>
```

Extracting this generates two translation units, one for the message itself, and one for the expression bundled in the message:

```
      <trans-unit id="home.racesPlanned" datatype="html">
        <source>
  Hello, <x id="ICU"/>.
</source>
        <target/>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app.component.ts</context>
          <context context-type="linenumber">30</context>
        </context-group>
      </trans-unit>
      <trans-unit id="963d1c0d7d9c63f9e7ba6c048709604b484b79ac" datatype="html">
        <source>{VAR_PLURAL, plural, =0 {no race is planned} =1 {only one race is
planned} other {<x id="INTERPOLATION"/> races are planned} }</source>
        <target/>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app.component.ts</context>
          <context context-type="linenumber">31</context>
        </context-group>
      </trans-unit>
```

Unfortunately, the second translation unit has an auto-generated ID, and the pluralization syntax must be understood and respected by the translator. It's possible to translate those two translation units, though, and the translation works as expected:

```
      <trans-unit id="home.racesPlanned" datatype="html">
        <source>
  Hello, <x id="ICU"/>.
</source>
        <target>Bonjour, <x id="ICU"/>.</target>
      </trans-unit>
      <trans-unit id="963d1c0d7d9c63f9e7ba6c048709604b484b79ac" datatype="html">
        <source/>
        <target>{VAR_PLURAL, plural, =0 {aucune course n'est planifiée}
                                     =1 {seule une course est planifiée}
                                     other {<x id="INTERPOLATION"/> courses sont
planifiées}}</target>
      </trans-unit>
```

23.6. Best practices

These best practices, acquired in years of development of i18ned applications, are not necessarily related to Angular, but to i18n in general.

Always specify an explicit unique ID for your messages. If you choose a meaningful ID, you often don't need to specify a meaning and a description for your message, because the ID is sufficient. Prefixing the IDs with the name of the component where they are used (like I did in all the example with the `home.` prefix) allows knowing where they are used, and finding them in the code easily.

Relying on auto-generated IDs doesn’t allow having different translations for two identical messages in your source language. It also makes it very hard to figure out what needs to be changed between two releases of your application.

Even if the translators are not always developers, store your messages files in your version control system. This makes sure a branch can have its own additions, that can be merged to the main branch when ready. It makes it easy to spot differences between branches and releases.

Duplication isn’t necessarily bad. You might think that two pages sharing a label "Save" or "OK" should use the same key, but maybe you will want to label them "OK, I’ll do it" and "OK, I accept" later. Or maybe they need to be differentiated in some foreign language. It’s even more important to use two separate keys for words that are identical in English, but not in other languages, like "free", which can mean "free as in beer" or "free as in speech". Other languages use different words for these two concepts.

Don’t confuse languages with countries. Don’t use country flags to represent languages. Some languages (like English) are spoken in several countries. And some countries use several languages (like Belgium, which uses French, Dutch and German).

Avoid concatenation to translate text with parameters. For example, to translate *"Hello, my name is X and I’m Y years old"*, don’t use a first key for *"Hello, my name is "*, a second key for *" and I’m "* and a third key for *" years old"*. Use a single key, containing interpolated expressions.

You should now be ready to conquer the world with your shiny i18ned Angular application.

Chapter 24. 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 don't already own it, you should know that there is also a *pro* package of this ebook. This package gives access to a whole set of exercises to build a real application, step by step, starting from scratch. For each step we provide a full unit tests suite covering 100% of your code, detailed instructions (which are not a basic copy-paste, but will push you to understand what you are doing), and a solution if you need (which might be the most beautiful one, or at least one consistent with the latest best practices) A home-brewed tool analyzes your code and computes a score for each exercise, and your progression is visible on a dashboard. If you're looking for actual code samples, always up-to-date, which might save you hours of work, our Pro Pack is waiting for you! You can even try the first exercises for free. And as you are already the proud owner of this ebook, we want to thank you for your historic support with a generous discount that you can grab here!

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 (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.

Appendix A: Changelog

Here are all the major changes since the first version. It should help you to see what changed since your last read!

By buying this ebook, you'll get all the following updates for free. Go to https://books.ninja-squad.com/claim to obtain the latest version of this ebook.

A.1. v1.8 - 2017-07-16

Global

- Bump to ng `4.3.0` (2017-07-16)
- Bump to ng `4.2.3` (2017-06-17)

Forms

- Remove min/max validators mention, as they have been removed temporarily in ng 4.2.3 (2017-06-17)

Send and receive data with Http

- Updates the chapter to use the new `HttpClientModule` introduced in ng 4.3.0. (2017-07-16)

Router

- List the new router events introduced in 4.3.0 (2017-07-16)

Advanced components and directives

- Add a section about `HostBinding` (2017-06-29)
- Add a section about `HostListener` (2017-06-29)
- New chapter on advanced components, with `ViewChild`, `ContentChild` and `ng-content`! (2017-06-29)

A.2. v1.7 - 2017-06-09

Global

- Bump to ng `4.2.0` (2017-06-09)
- Bump to ng `4.1.0` (2017-04-28)

Forms

- Introduce the `min` and `max` validators from version 4.2.0 (2017-06-09)

Router

- New chapter on advanced router usage: protected routes with guards, nested routes, resolvers and lazy-loading! (2017-04-28)

Angular compiler

- Adds a chapter about the Angular compiler and the differences between JiT and AoT. (2017-05-02)

A.3. v1.6 - 2017-03-24

Global

- 🎉 Bump to stable release `4.0.0` 🎉 (2017-03-24)
- Bump to `4.0.0-rc.6` (2017-03-23)
- Bump to `4.0.0-rc.5` (2017-03-23)
- Bump to `4.0.0-rc.4` (2017-03-23)
- Bump to `4.0.0-rc.3` (2017-03-23)
- Bump to `4.0.0-rc.1` (2017-03-23)
- Bump to `4.0.0-beta.8` (2017-03-23)
- Bump to ng `4.0.0-beta.7` and TS 2.1+ is now required (2017-03-23)
- Bump to `4.0.0-beta.5` (2017-03-23)
- Bump to `4.0.0-beta.0` (2017-03-23)
- Each chapter now has a link to the corresponding exercise of our Pro Pack Chapters are slightly re-ordered to match the exercises order. (2017-03-22)

The templating syntax

- Use `as`, introduced in 4.0.0, instead of `let` for variables in templates (2017-03-23)
- The `template` tag is now deprecated in favor of `ng-template` in 4.0 (2017-03-23)
- Introduces the else syntax from version 4.0.0 (2017-03-23)

Dependency Injection

- Fix the Babel 6 config for dependency injection without TypeScript (2017-02-17)

Pipes

- Introduce the `as` syntax to store a `NgIf` or `NgFor` result, which can be useful with some pipes like `slice` or `async`. (2017-03-23)
- Adds `titlecase` pipe introduced in 4.0.0 (2017-03-23)

Services

- New `Meta` service in 4.0.0 to get/set meta tags (2017-03-23)

Testing your app

- `overrideTemplate` has been added in 4.0.0 (2017-03-23)

Forms

- Introduce the `email` validator from version 4.0.0 (2017-03-23)

Send and receive data with Http

- Use `params` instead of the deprecated `search` in 4.0.0 (2017-03-23)

Router

- Use `paramMap` introduced in 4.0 instead of `params` (2017-03-23)

Advanced observables

- Shows the `as` syntax introduced in 4.0.0 as an alternative for the mulitple async pipe subscriptions problem (2017-03-23)

Internationalization

- Add a new chapter on internationalization (i18n) (2017-03-23)

A.4. v1.5 - 2017-01-25

Global

- Bump to `2.4.4` (2017-01-25)
- The big rename: "Angular 2" is now known as "Angular" (2017-01-13)
- Bump to `2.4.0` (2016-12-21)

Forms

- Fix the `NgModel` explanation (2017-01-09)
- `Validators.compose()` is no longer necessary, we can apply several validators by just passing an array. (2016-12-01)

A.5. v1.4 - 2016-11-18

Global

- Bump to `2.2.0` (2016-11-18)
- Bump to `2.1.0` (2016-10-17)
- Remove typings and use `npm install @types/⋯` (2016-10-17)
- Use `const` instead of `let` and TypeScript type inference whenever possible (2016-10-01)
- Bump to `2.0.1` (2016-09-24)

Testing your app

- Use `TestBed.get` instead of `inject` in tests (2016-09-30)

Forms

- Add an async validator example (2016-11-18)
- Remove the useless (2.2+) `.control` in templates like `username.control.hasError('required')`. (2016-11-18)

Router

- `routerLinkActive` can be exported (2.2+). (2016-11-18)
- We don't need to unsubscribe from the router params in the `ngOnDestroy` method. (2016-10-07)

Advanced observables

- New chapter on Advanced Observables! (2016-11-03)

A.6. v1.3 - 2016-09-15

Global

- 🎉 Bump to stable release `2.0.0` 🎉 (2016-09-15)
- Bump to `rc.7` (2016-09-14)
- Bump to `rc.6` (2016-09-05)

From zero to something

- Update the SystemJS config for `rc.6` and bump the RxJS version (2016-09-05)

Pipes

- Remove the section about the replace pipe, removed in rc.6 (2016-09-05)

A.7. v1.2 - 2016-08-25

Global

- Bump to `rc.5` (2016-08-23)
- Bump to `rc.4` (2016-07-08)
- Bump to `rc.3` (2016-06-28)
- Bump to `rc.2` (2016-06-16)
- Bump to `rc.1` (2016-06-08)
- Code examples now follow the official style guide (2016-06-08)

From zero to something

- Small introduction to NgModule when you start your app from scratch (2016-08-12)

The templating syntax

- Replace the deprecated `ngSwitchWhen` with `ngSwitchCase` (2016-06-16)

Dependency Injection

- Introduce modules and their role in DI. Changed the example to use a custom service instead of Http. (2016-08-15)
- Remove deprecated `provide()` method and use `{provide: ⋯}` instead (2016-06-09)

Pipes

- Date pipe is now fixed in `rc.2`, no more problem with Intl API (2016-06-16)

Styling components and encapsulation

- New chapter on styling components and the different encapsulation strategies! (2016-06-08)

Services

- Add the service to the module's providers (2016-08-21)

Testing your app

- Tests now use the TestBed API instead of the deprecated TestComponentBuilder one. (2016-08-15)
- Angular 2 does not provide Jasmine wrappers and custom matchers for unit tests in `rc.4` anymore (2016-07-08)

Forms

- Forms now use the new form API (FormsModule and ReactiveFormsModule). (2016-08-22)
- Warn about forms module being rewritten (and deprecated) (2016-06-16)

Send and receive data with Http

- Add the HttpModule import (2016-08-21)
- `http.post()` now autodetects the body type, removing the need of using `JSON.stringify` and setting the `ContentType` (2016-06-16)

Router

- Introduce RouterModule (2016-08-21)
- Update the router to the API v3! (2016-07-08)
- Warn about router module being rewritten (and deprecated) (2016-06-16)

Changelog

- Mention free updates and web page for obtaining latest version (2016-07-25)

A.8. v1.1 - 2016-05-11

Global

- Bump to `rc.0`. All packages have changed! (2016-05-03)
- Bump to `beta.17` (2016-05-03)
- Bump to `beta.15` (2016-04-16)
- Bump to `beta.14` (2016-04-11)
- Bump to `beta.11` (2016-03-20)
- Bump to `beta.9` (2016-03-11)
- Bump to `beta.8` (2016-03-10)
- Bump to `beta.7` (2016-03-04)
- Display the Angular 2 version used in the intro and in the chapter "Zero to something". (2016-03-04)
- Bump to `beta.6` (`beta.4` and `beta.5` were broken) (2016-03-04)
- Bump to `beta.3` (2016-03-04)
- Bump to `beta.2` (2016-03-04)

Diving into TypeScript

- Use `typings` instead of `tsd`. (2016-03-04)

The templating syntax

- `*ngFor` now uses `let` instead of to declare a variable `*ngFor="let pony of ponies"` [small](2016-05-03)#
- `*ngFor` now also exports a `first` variable (2016-04-16)

Dependency Injection

- Better explanation of hierarchical injectors (2016-03-04)

Pipes

- A `replace` pipe has been introduced (2016-04-16)

Reactive Programming

- Observables are not scheduled for ES7 anymore (2016-03-04)

Building components and directives

- Explain how to remove the compilation warning when using `@Input` and a setter at the same time (2016-03-04)

- Add an explanation on `isFirstChange` for `ngOnChanges` (2016-03-04)

Testing your app

- `injectAsync` is now deprecated and replaced by `async` (2016-05-03)
- Add an example on how to test an event emitter (2016-03-04)

Forms

- A pattern validator has been introduced to make sure that the input matches a regexp (2016-04-16)
- Add a mnemonic tip to rememeber the `[()]` syntax: the banana box! (2016-03-04)
- Examples use `module.id` to have a relative `templateUrl` (2016-03-04)
- Fix error `ng-no-form` → `ngNoForm` (2016-03-04)
- Fix errors `(ngModel)` → `(ngModelChange)`, `is-old-enough` → `isOldEnough` (2016-03-04)

Send and receive data with Http

- Use `JSON.stringify` before sending data with a POST (2016-03-04)
- Add a mention to `JSONP_PROVIDERS` (2016-03-04)

Router

- Introduce the new router (previous one is deprecated), and how to use parameters in URLs! (2016-05-06)
- `RouterOutlet` inserts the template of the component just after itself and not inside itself (2016-03-04)

Zones and the Angular magic

- New chapter! Let's talk about how Angular 2 works under the hood! First part is about how AngularJS 1.x used to work, and then we'll see how Angular 2 differs, and uses a new concept called zones. (2016-05-03)