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.

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 deep 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.

The ebook is using Angular version 20.0.1 for the examples.

*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 now have a major release every six months. And 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 is updated (for free) with all the future major releases, it is now

> named "Become a Ninja with Angular" (without any number).

# Chapter 2. A gentle introduction to ECMAScript 2015+

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 version 5, that has been used these last years.

But, in 2015, a new version of the spec was released, called ECMAScript 2015, ES2015, or sometimes ES6, as it was the sixth version of the specification. And since then, we have had a yearly release of the specification (ES2016, ES2017, etc.), with a few new features every year. From now on, I'll mainly say ES2015, as it is the most popular way to reference it, or ES2015+ to reference ES2015, ES2016, ES2017, etc. 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 ES2015+. So we're going to spend some time in this chapter to get a grip on what ES2015+ 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 ES2015+, 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

The sixth version of the specification reached its final state in 2015. So it's now supported by modern browsers, but there are still browsers in the wild that don't support it yet, or only support it partially. And of course, now that we have a new specification every year (ES2016, ES2017, etc.), some browsers will always be late. You might be thinking: what's the point 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 ES2015+ wants to write ES2015+ apps, the community has found a solution: a transpiler.

A transpiler takes ES2015+ source code and generates ES5 code that can run in every browser. It even generates the source map files, which allows you to debug directly the ES2015+ source code from the browser. Back in 2015, there were two main alternatives to transpile ES2015+ code:

- Traceur, a Google project, historically the first one but now unmaintained.
- 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.

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 nowadays, so I would advise you to use it. It is now the de-facto standard in this area.

So if you want to play with ES2015+, 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 ES2015+ 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 language, 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;
}
```

ES2015 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. ES2015 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. Shorthands in object creation

Not a new keyword, but it can also catch your attention when reading ES2015 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 };
}
```

Similarly, when you want to define a method in the object:

```
function createPony() {
  return {
    run: () => {
      console.log('Run!');
    }
  };
}
```

you can simplify it to:

```
function createPony() {
  return {
    run() {
      console.log('Run!');
    }
  };
}
```

## 2.5. Destructuring assignment

This new feature can also catch your attention when reading ES2015 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 ES2015, 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 in 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 assigns 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. ES2015 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.

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);
}
```

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

ES2015 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 this:

```
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?

ES2015 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 ES2015. It makes sure that you iterate over the collection values, and not also over 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: ES2015 introduces classes to JavaScript! You can now easily use classes and inheritance in JavaScript. You always could, using prototypal inheritance, but that 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 ES2015, 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 onto 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 ES2015.

```
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 ES2015 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. Async/await

We were talking about promises earlier, and it's worth knowing that another keyword was introduced to handle them more synchronously: `await`.

This is not a feature introduced in ECMAScript 2015 but in ECMAScript 2017, and to use `await`, your function must be marked as `async`. When you use the `await` keyword in front of a Promise, you pause the execution of your `async` function, wait for the Promise to resolve, and then resume the execution of the `async` function. The returned value will be the resolved value.

So we can write our previous example using `async/await` like this:

```
async function getUserRightsAndUpdateMenu() {
  // getUser is a promise
  const user = await getUser(login);
  // getRights is a promise
  const rights = await getRights(user);
  updateMenu(rights);
}
await getUserRightsAndUpdateMenu();
```

And your code now looks like it is synchronous! Another cool feature of `async/await` is that you can use a simple `try/catch` to handle errors:

```
async function getUserRightsAndUpdateMenu() {
  try {
    // getUser is a promise
    const user = await getUser(login);
    // getRights is a promise
    const rights = await getRights(user);
    updateMenu(rights);
  } catch (e) {
    // will be called if getUser, getRights or updateMenu fails
    console.log(e);
  }
}
await getUserRightsAndUpdateMenu();
```

Note that async/await is still asynchronous, although it looks like synchronous. The function execution is paused and resumed, but just like with callbacks, this doesn't block the thread: other JavaScript events can be handled while the execution is paused.

## 2.12. Sets and Maps

This is a short one: you now have proper collections in ES2015. 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 in order to iterate over a collection in a template.

## 2.13. 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>`;
```

One last feature is the ability to tag them. You can define a function, and apply it to a template string. Here `askQuestion` adds an interrogation point at the end of the string:

```
const askQuestion = strings => strings + '?';
const template = askQuestion`Hello there`;
```

So what's the difference with a simple function? The tag function in fact receives several arguments:

- an array of the static parts of the string
- the values resulting from the evaluation of the expressions

For example if we have a template string containing expressions:

```
const person1 = 'Cedric';
const person2 = 'Agnes';
const template = `Hello ${person1}! Where is ${person2}?`;
```

then the tag function will receive the various static and dynamic parts. Here we have a tag function to uppercase the names of the protagonists:

```
const uppercaseNames = (strings, ...values) => {
  // `strings` is an array with the static parts ['Hello ', '! Where is ', '?']
  // `values` is an array with the evaluated expressions ['Cedric', 'Agnes']
  const names = values.map(name => name.toUpperCase());
  // `names` now has ['CEDRIC', 'AGNES']
  // let's merge the `strings` and `names` arrays
  return strings.map((string, i) => `${string}${names[i] ? names[i] : ''}`).join('');
};
const result = uppercaseNames`Hello ${person1}! Where is ${person2}?`;
// returns 'Hello CEDRIC! Where is AGNES?'
```

Let's now talk about one of the big changes introduced: modules.

## 2.14. Modules

A standard way to organize functions in namespaces and to dynamically load code in JS has always been lacking. Node.js 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.

ES2015 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 ES2015 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, which 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, and you can even give it an alias:

```
import { start as startRace } from './races-service';
```

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

And if you want to use all the exported symbols (functions, constants, classes etc.) 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 of importing 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 choose an alias that allows you 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.

## 2.15. Conclusion

That ends our gentle introduction to ES2015+. We skipped some other parts, but if you're comfortable with this chapter, you will have no problem writing your apps in ES2015+. 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 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 ES2015+

## 3.1. Dynamic, static and optional types

You may have heard that Angular apps can be written in 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 other reasons 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 to write 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 ES2015+ 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 about 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 up 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 of maintaining 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, you'll probably want to code every application with it. 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, 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!

That's why we're going to spend some time learning TypeScript (TS). Angular is clearly built to leverage this language, 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 ES2015+, 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 ponyNumber: number = 0;
const ponyName: 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 come 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
}
```

Since TypeScript 2.4, you can even specify a string value:

```
enum Position {
  First = 'First',
  Second = 'Second',
  Other = 'Other'
}
```

To be honest though, we don't use enums a lot in our projects: we use union types. They are simpler and cover roughly the same use-cases:

```
let color: 'blue' | 'red' | 'green';
// we can only give one of these values to `color`
color = 'blue';
```

TypeScript even allows you to create your own types, so you could do something like:

```
type Color = 'blue' | 'red' | 'green';
const ponyColor: Color = 'blue';
```

## 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, which is 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;
}
```

You'll see that we often use interfaces throughout the book to represent our entities. We use interfaces for our models in our other projects as well. We usually append a `Model` suffix to make it clear. It's then very easy to create a new entity:

```
interface PonyModel {
  name: string;
  speed: number;
}
const pony: PonyModel = { name: 'Light Shoe', speed: 56 };
```

## 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. The interface definition will be:

```
interface CanRun {
  run(meters: number): void;
}
```

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

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

## 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: number): void {
    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: number): void {
    logger.log(`pony runs ${meters}m`);
  }

  eat(): void {
    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 ES2015+ feature. It is only possible in TypeScript.

```
class SpeedyPony {
  speed = 10;

  run(): void {
    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(): void {
    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(): void {
    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 type definitions of an NPM library if they are packaged with the library itself. 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 feature was added in TypeScript 1.5, notably to help support 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 that was 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. 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. You are 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: number) {
    // call API
  }
}
```

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

```
const Log = () => {
  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 prototype of the class 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', template: 'home' })
class Home {
  constructor() {
    logger.log('Home');
  }
}
```

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 throughout the book.

So my advice would be to give TypeScript a try! All my examples from here will be in TypeScript, as Angular and all the tooling around are really designed for it.

# Chapter 5. Advanced TypeScript

If you're just starting to learn TypeScript, you can safely skip this chapter for now and come back later. This chapter is here to showcase some more advanced usages of TypeScript. They'll only make sense if you already have some familiarity with the language

## 5.1. readonly

You can use the `readonly` keyword to mark the property of a class or interface as... read only! That way, the compiler will refuse to compile any code trying to assign a new value to the property:

```
interface Config {
  readonly timeout: number;
}

const config: Config = { timeout: 2000 };
// `config.timeout` is now readonly and can't be reassigned
```

## 5.2. keyof

The `keyof` keyword can be used to get a type representing the union of the names of the properties of another type. For example, you have a `PonyModel` interface:

```
interface PonyModel {
  name: string;
  color: string;
  speed: number;
}
```

You want to build a function that returns the value of a property. You could implement a naive version:

```
function getProperty(obj: any, key: string): any {
  return obj[key];
}

const pony: PonyModel = {
  name: 'Rainbow Dash',
  color: 'blue',
  speed: 45
};
const nameValue = getProperty(pony, 'name');
```

Two problems here:

- you can give any value to the `key` parameter, even keys that don't exist on `PonyModel`.
- the return type being `any`, you are losing a lot of type information.

This is where `keyof` can shine. `keyof` allows you to list all the keys of a type:

```
type PonyModelKey = keyof PonyModel;
// this is the same as `'name'|'speed'|'color'`
let property: PonyModelKey = 'name'; // works
property = 'speed'; // works
// key = 'other' would not compile
```

So we can use this type to make `getProperty` safer, by declaring that:

- the first parameter is of type `T`
- the second parameter is of type `K`, which is a key of `T`

```
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}

const pony: PonyModel = {
  name: 'Rainbow Dash',
  color: 'blue',
  speed: 45
};
// TypeScript infers that `nameValue` is of type `string`!
const nameValue = getProperty(pony, 'name');
```

We killed two birds with one stone here:

- `key` can now only be an existing property of `PonyModel`
- the return value will be inferred by TypeScript (which is pretty awesome!)

Now let's see how we can leverage `keyof` to do even more.

## 5.3. Mapped type

Let's say you want to create a type that has exactly the same properties as `PonyModel`, but you want every property to be optional. You can of course define it manually:

```
interface PartialPonyModel {
  name?: string;
  color?: string;
  speed?: number;
}
```

```
const pony: PartialPonyModel = {
  name: 'Rainbow Dash'
};
```

But you can do something more generic with a mapped type:

```
type Partial<T> = {
  [P in keyof T]?: T[P];
};

const pony: Partial<PonyModel> = {
  name: 'Rainbow Dash'
};
```

The `Partial` type is a transformation that applies the `?` modifier to every property of a type! In fact, you don't have to define the type `Partial` yourself, because since version 2.1, it's part of the language itself, and it's declared exactly like in the above example.

TypeScript offers other mapped types out of the box.

### 5.3.1. Readonly

`Readonly` makes all the properties of an object `readonly`:

```
const pony: Readonly<PonyModel> = {
  name: 'Rainbow Dash',
  color: 'blue',
  speed: 45
};
// all properties are `readonly`
```

### 5.3.2. Pick

`Pick` helps you build a type with only some of the original properties:

```
const pony: Pick<PonyModel, 'name' | 'color'> = {
  name: 'Rainbow Dash',
  color: 'blue'
};
// `pony` can't have a `speed` property
```

### 5.3.3. Record

`Record` helps you build a type with the same properties as another type, but with a different type:

```
interface FormValue {
```

```
  value: string;
  valid: boolean;
}

const pony: Record<keyof PonyModel, FormValue> = {
  name: { value: 'Rainbow Dash', valid: true },
  color: { value: 'blue', valid: true },
  speed: { value: '45', valid: true }
};
```

There are even more than that, but these are the most useful.

## 5.4. Union types and type guards

Union types are really handy. Let's say your application has authenticated users and anonymous users, and sometimes you need to do a different action depending on that. You can model this as:

```
interface User {
  type: 'authenticated' | 'anonymous';
  name: string;
  // other fields
}

interface AuthenticatedUser extends User {
  type: 'authenticated';
  loggedSince: number;
}

interface AnonymousUser extends User {
  type: 'anonymous';
  visitingSince: number;
}

function onWebsiteSince(user: User): number {
  if (user.type === 'authenticated') {
    // this is a LoggedUser
    return (user as AuthenticatedUser).loggedSince;
  } else if (user.type === 'anonymous') {
    // this is an AnonymousUser
    return (user as AnonymousUser).visitingSince;
  }
  // TS doesn't know every possibility was covered
  // so we have to return something here
  return 0;
}
```

I don't know about you, but I don't like these `as ⋯` explicit casts. Maybe we can do better?

One possibility is to use a type guard, a special function whose sole purpose is to help the

TypeScript compiler.

```
function isAuthenticated(user: User): user is AuthenticatedUser {
  return user.type === 'authenticated';
}

function isAnonymous(user: User): user is AnonymousUser {
  return user.type === 'anonymous';
}

function onWebsiteSince(user: User): number {
  if (isAuthenticated(user)) {
    // this is inferred as a LoggedUser
    return user.loggedSince;
  } else if (isAnonymous(user)) {
    // this is inferred as an AnonymousUser
    return user.visitingSince;
  }
  // TS still doesn't know every possibility was covered
  // so we have to return something here
  return 0;
}
```

This is better! But we still need to return a default value, even if we covered all the possibilities.

We can slightly improve the situation if we drop the type guards and use a union type instead.

```
interface BaseUser {
  name: string;
  // other fields
}

interface AuthenticatedUser extends BaseUser {
  type: 'authenticated';
  loggedSince: number;
}

interface AnonymousUser extends BaseUser {
  type: 'anonymous';
  visitingSince: number;
}

type User = AuthenticatedUser | AnonymousUser;

function onWebsiteSince(user: User): number {
  if (user.type === 'authenticated') {
    // this is inferred as a LoggedUser
    return user.loggedSince;
  } else {
```

```
    // this is narrowed as an AnonymousUser
    // without even testing the type!
    return user.visitingSince;
  }
  // no need to return a default value
  // as TS knows that we covered every possibility!
}
```

This is even better, as TypeScript automatically narrows the type in the `else` branch.

Sometimes you know that the model will grow in the future, and that more cases will need to be handled. For example if you introduce an `AdminUser`. In that case, you can use a `switch`. A `switch` statement will break if one of the cases is not handled. So introducing our `AdminUser`, or another type of user later, would automatically add compilation errors in every place you need to handle it!

```
interface AdminUser extends BaseUser {
  type: 'admin';
  adminSince: number;
}

type User = AuthenticatedUser | AnonymousUser | AdminUser;

function onWebsiteSince(user: User): number {
  switch (user.type) {
    case 'authenticated':
      return user.loggedSince;
    case 'anonymous':
      return user.visitingSince;
    case 'admin':
      // without this case, we could not even compile the code
      // as TS would complain that all possible paths are not returning a value
      return user.adminSince;
  }
}
```

I hope these patterns will help you. Now let's focus on Web Components.

# Chapter 6. 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.

## 6.1. 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 we'll have to wait a few years to use them fully, or even if the concept never takes off.

This emerging standard is defined in 3 specifications:

- Custom elements
- Shadow DOM
- Template

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

## 6.2. 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 using `customElements.define`:

```
class Pony extends HTMLElement {
  constructor() {
    super();
    console.log("I'm a pony!");
  }

}

customElements.define('ns-pony', Pony);
```

And you can then use it:

```
<ns-pony></ns-pony>
```

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:

```
class Pony extends HTMLElement {
  constructor() {
    super();
    console.log("I'm a pony!");
  }


  /**
   * This is called when the component is inserted
   */
  connectedCallback() {
    this.innerHTML = '<h1>General Soda</h1>';
  }

}
```

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.

## 6.3. 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 nothing leaks from the component to the app, or vice-versa.

Going back to our previous example:

```
class Pony extends HTMLElement {
  constructor() {
    super();
    const shadow = this.attachShadow({ mode: 'open' });
    const title = document.createElement('h1');
    title.textContent = 'General Soda';
    shadow.appendChild(title);
  }
}
```

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.

## 6.4. 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 methods like `getElementById()` and it can be safely placed anywhere in your page.

To use a template, it needs to be cloned:

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

```
class Pony extends HTMLElement {
  constructor() {
    super();
    const template = document.querySelector('#pony-template');
    const clonedTemplate = document.importNode(template.content, true);
    const shadow = this.attachShadow({ mode: 'open' });
    shadow.appendChild(clonedTemplate);
  }
}
```

## 6.5. Frameworks on top of Web Components

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, the first attempt from Google
- LitElement, a more recent project from the Polymer team ;
- X-tag from Mozilla and Microsoft
- Stencil.

I won't go into the details, but you can easily use an already existing 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 -->
<script src="google-map.js"></script>
```

```
<!-- 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://www.webcomponents.org/.

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

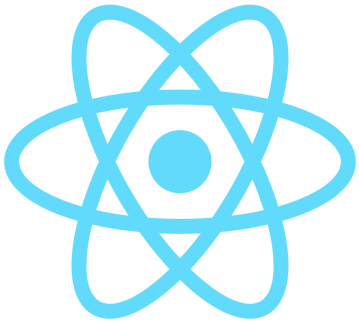
That's obviously far too short a chapter to tell you everything there is to say on Web Components, but you'll see that some of the concepts are going to pop out along your read. And you'll definitely see that the Google team designed Angular to make it easy to use Web Components with our Angular components.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

# Chapter 7. Grasping Angular's philosophy

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

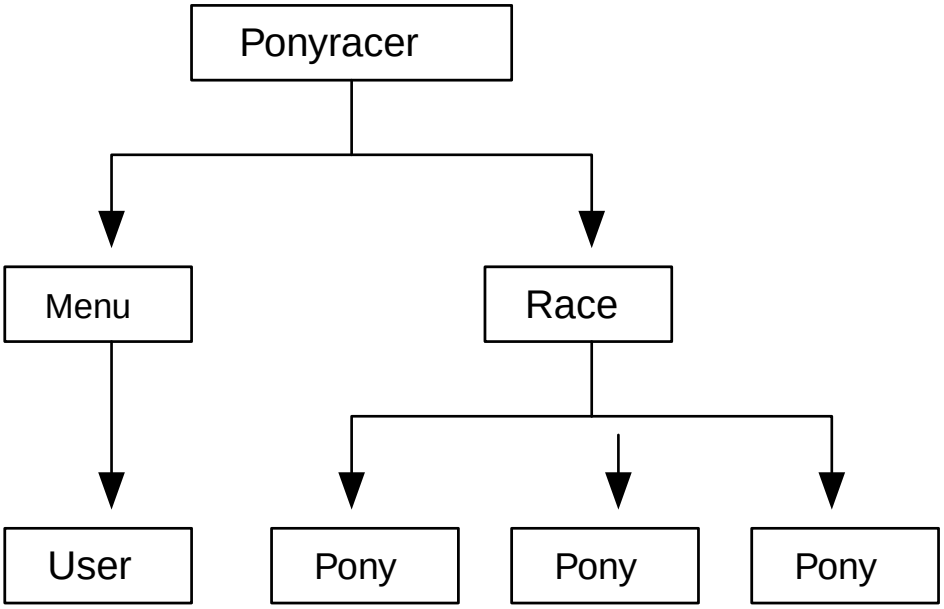
First and foremost, Angular is component-oriented. You will write tiny components and, together, they will constitute a whole application. A component is a group of HTML elements in a template, dedicated to a particular task. For this, you will usually also need to have some logic linked to that template, to populate data, and react to events for example.

This component orientation is something that is becoming widely shared across front-end frameworks: React, the cool kid from Facebook, has been doing it that way from the beginning; Ember has its own way of doing something similar; and others like Svelte 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`), displaying a menu (`Menu`) with the logged in user (`User`) and a child view (`Race`), displaying, 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: ES2015+. So you can write your components in ES5 (but that's not very cool) or in ES2015+ (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 us 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, inject, Signal, signal } from '@angular/core';
import { RaceModel, RaceService } from './services';
import { Pony } from './components';

@Component({
  selector: 'ns-race',
  templateUrl: './race.html',
  imports: [Pony]
})
export class Race {
  private readonly raceService = inject(RaceService);
  protected readonly race = signal(this.raceService.get());
}
```

And the template looks like this:

```
<div>
```

```
  <h2>{{ race().name }}</h2>
  <div>{{ race().status }}</div>
  @for (pony of race().ponies; track pony.id) {
    <div>
      <ns-pony [ponyModel]="pony" />
    </div>
  }
</div>
```

The template shouldn't be too hard to understand. It's HTML with expressions in curly braces `{{ }}`, which will be evaluated and replaced by the corresponding value. 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 ES2015 Modules), or learn how to avoid them by making your components *standalone*.

You can also take available libraries of components from the community and just use them in your app, and be able to enjoy their features.

Such libraries 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 application 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 a 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 8. From zero to something

Let's start by creating our first Angular app and our first component, with a minimum of tooling.

## 8.1. Node.js and NPM

Pretty much all the modern JavaScript tools are built for Node.js and NPM these days. 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`).

## 8.2. Angular CLI

You *could* setup everything by yourself, starting with a TypeScript project, then install every dependency needed, etc.

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

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

But it's a bit cumbersome to setup everything yourself, especially when there are sooooo many tools to learn first.

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 were a few attempts for Angular from the community.

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 (the popular kid these years), 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 under continuous development, with a dedicated Google team working on it and making it better and better. It is now the recommended and *de facto* standard way of creating and building Angular apps. So let's give it a try, and discover the ton of cool stuff packed into it!

If you want, you can follow our online 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, and goes further than what we see in the chapter.

First let's install Angular CLI, and generate a new application with the `ng new` command. If you want to use exactly the same CLI version than we are (`20.0.1`), you can use `npm install -g @angular/cli@20.0.1` instead.

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

```
ng new ponyracer --defaults --no-routing --prefix ns
```

This will create a project skeleton in a new directory called `ponyracer`. From this directory, you can start your app with:

```
ng serve
```

This will start a small HTTP server locally, with a hot reload configuration. It means that every time you modify and save a file, the server will rebuild the app, and the browser will reload it immediately.

Tada! You have your first application up and running! 🎉

In Angular 15, the framework introduced a new feature called *standalone components*, which is now the default since Angular v17. Until then, components had to be declared in Angular modules, which are quite a complex concept to grasp and use correctly, especially when starting with Angular. Using standalone components allows us to avoid having to create Angular modules and makes many things simpler, especially for beginners. Standalone components are the way of the future, so we chose to use that option. We will explain Angular modules in a later chapter though, because you will still have to understand their purpose and use existing modules in your day-to-day work, but for now, you don't need to worry too much about them.

## 8.3. Application structure

Let's dive for a few seconds into the generated code.

Open the project in your preferred IDE. You can use pretty much anything you want, but you should activate the TypeScript support for maximum comfort. Pick your favorite: Webstorm, Visual Studio Code… All of them have great support for TypeScript.

If your IDE supports it, code completion should work as the Angular dependencies have their own `d.ts` files in the `node_modules` directory, and TypeScript is able to detect them. 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. 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.

You should see a bunch of configuration files in the root directory: welcome to Modern JavaScript!

The first one you may recognize is the `package.json` file: that's where the dependencies of the application are defined. You can have a look inside, it should now contain the following dependencies:

- the different `@angular` packages.
- `rxjs`, a really cool library for reactive programming. We have a dedicated chapter on this topic and about RxJS in particular.
- `zone.js`, doing the heavy lifting for detecting the changes (we'll dive into this later also).
- some dependencies for developing the application, like the CLI, TypeScript, some test librairies, some typings…

TypeScript itself has a configuration file `tsconfig.json` (and another one called `tsconfig.app.json`), which stores the compilation options. As we saw in the previous chapters, we are using TypeScript with decorators (hence the two options about decorators). The `sourceMap` option allows you to generate source maps, i.e. files that contain a mapping between the generated JavaScript code and the original TypeScript code. Those source maps are used by the browser to let you debug the JavaScript code it executes by stepping through the original TypeScript code that you have written.

TypeScript projects often also use ESLint (that we need to add to a project, as we show in the exercise `Getting Started` of our online course), a linter used to check your code against the best practices. ESLint has its own options, stored in `.eslintrc.json`, where you add/remove some of its rules.

Angular CLI itself has a configuration file `angular.json` if you want to override some of its defaults.

This ebook is using Angular version `20.0.1` for the examples. Angular CLI will probably install the most recent version, which might not be exactly the same. If you want to use the same version as we are, replace the version in the `package.json` by `20.0.1` for each Angular package. That might save you a few headaches! Or, even better, follow our free online exercise Getting Started 🐴! which is always up-to-date and battle-tested!

Now that we have been over the configuration, let's see the application code.

## 8.4. Our first standalone component

As we saw in the previous section, a component is a combination of a view (the template) and some logic (our TS class). The CLI has already created one for us: `src/app/app.ts`. Let's check it out:

*app.ts*

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

@Component({
  selector: 'ns-root',
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
  protected title = 'ponyracer';
}
```

Our application itself is a simple component. To tell Angular that it is a component, we use the `@Component` decorator. And to be able to use this decorator, we have to import it as you can see at the top of the file.

When you write new components, 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/common/http`; or, if we use the router, we'll import from `@angular/router`, etc.

*app.ts*

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

@Component({
})
export class App {
  protected title = 'ponyracer';
}
```

The `@Component` decorator is expecting a configuration object. We'll see later in detail what you can configure here, but for now let's start with the `selector` property. It will tell Angular what to look for in our HTML pages. Every time Angular finds an element in our HTML which matches the selector of the component, it will create an instance of the component, and replace the content of the element by the template of the component.

*app.ts*

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

```
@Component({
  selector: 'ns-root',
})
export class App {
  protected title = 'ponyracer';
}
```

So, here, every time our HTML contains an element like `<ns-root />`, Angular will instantiate a new instance of our `App` class.

There is a clear naming convention established, and applied by Angular CLI. Angular recommends using a prefix in component selectors, to avoid name clashes with external components. For example, since our company is named **N**inja **S**quad, we chose to use the prefix `ns`. Our pony component selector is thus named `ns-pony`. You can configure Angular CLI so that it prepends this prefix to every generated component. If you remember, when we created the project with `ng new`, we passed the option `--prefix ns`. That's what this option does: it configures the project to have components generated with the `ns` prefix.

A component must also have a template. We can have an inline template or externalize it in another file, like the CLI does:

*app.ts*

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

@Component({
  selector: 'ns-root',
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
  protected title = 'ponyracer';
}
```

The corresponding HTML is defined in `app.html`, with a bunch of static elements, except the first `h1`:

*app.html*

```
<h1>Hello, {{ title }}</h1>
<p>Congratulations! Your app is running. 🎉</p>
```

Finally, you can see that the component has an additional option: `imports`.

*app.ts*

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

```
@Component({
  selector: 'ns-root',
  imports: [],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
  protected title = 'ponyracer';
}
```

Our component is standalone (the `standalone: true` property is by default since Angular v19). This means that the component doesn't need to be declared in an Angular module in order to be usable.

The property `imports: []` is not always strictly required. Its goal is to tell Angular which other components, pipes and directives can be used inside the template of our component. Most of the components that we create use pipes and directives provided by Angular. These very commonly used pipes and directives are declared in an Angular module named `CommonModule`. Using `imports: [CommonModule]` in the configuration of our component thus makes it possible to use them all in its template, or you can explicitly import only the ones you need.

We were talking about ES2015+ and TS modules in the first chapters, which define imports and exports. The TypeScript compiler, when it sees an interface such as `Race` being used in the TypeScript source code, must know where to find the definition of this interface `Race`. That's why you need to import it.

The `imports` property of the `@Component` decorator serves a similar purpose: if Angular were to find an element `<ns-race>` in the template of `App`, it would need to know where and how the corresponding `Race` is defined. To let it know where it can find its definition, you would need to add the `Race` to the imports of the `App` decorator.

## 8.5. Bootstrapping the app

Finally, we need to start our app, using the `bootstrapApplication` function. Angular CLI will by default generate a separate file containing this bootstrap logic: `main.ts`:

*main.ts*

```
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { App } from './app/app';

bootstrapApplication(App, appConfig)
  .catch((err) => console.error(err));
```

As you can see, what it expects is the root component of the application: `App`.

Yay! But wait a second. We need an HTML file to serve to our users, right?

The CLI created an `index.html` file for us, which is the single page of our application. You might

wonder how it could possibly work, since it doesn't contain any `script` element.

When you run `ng serve`, the CLI calls the TypeScript compiler. The compiler outputs JavaScript files. The CLI then bundles them and adds the necessary `script` elements to the `index.html` file (using Vite behind the scenes).

Hopefully, you now have a better understanding of the various parts of this first Angular application. It's not really a dynamic app yet, 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 signals and templating.

# Chapter 9. Signals: the building blocks of the application state

Traditional web applications use JavaScript to modify what the HTML page displays by modifying the DOM.

An Angular application does the same thing, but the framework is designed so that you almost never have to query or modify the DOM in your code. Instead, you write HTML templates which display data contained in the component properties. The magic of Angular is that modifying the data in the components is sufficient to update the DOM. Angular detects the changes and applies the modifications to the DOM for you.

This mechanism is called *Change Detection*. We'll explore it in more detail later in the book.

Angular has always allowed developers to use plain old JavaScript objects and arrays to store the data that the templates display. Angular is able to detect the modifications or replacements done to these plain old JavaScript objects. This mechanism, however, is quite brute-force, has a non-negligible cost in terms of performance, and comes with many drawbacks.

Since Angular 16, a revolution has started. While still supporting this mechanism, Angular now promotes a new way of handling the state. This new way eliminates the drawbacks of the brute-force approach. It makes things more efficient. But it requires the developers to use a special type of objects to store the data of their application: **signals**.

## 9.1. What is a signal?

A signal is a box which always contains a value. You can get the value inside the box by *reading* the signal. And when the signal is writable, you can put another value inside the box by *setting* or *updating* the signal. The value inside the box can be anything: a number, a string, an object, an array, or even `null` or `undefined`.

Those boxes come with super-powers. When a component template reads a signal to display its value, Angular knows that the template depends on that signal. And thus, every time you update that signal with a new value, Angular knows that it must update the DOM generated by this template.

This is the principle of the new change detection mechanism used by Angular. If all components use signals to store the data displayed by their template, then the brute-force mechanism is not necessary anymore.

This revolution that started in Angular 16 has not ended yet. But all the signal-related features that were introduced since then are now stable enough. We can use them in our applications.

This book will thus describe how to use Angular in the new, modern way, based on signals. We will mention how it was done before though, because there's a good chance that you'll find code in your applications that still uses the legacy way of writing components. Both ways can coexist in the same application, but we advise you to start adopting the newest best practices.

## 9.2. Creating, reading and writing signals

Let's now start by showing how you can create a *writable* signal, read its value, and update it. We'll talk about the other kinds of signals and about their other capabilities in later chapters of the book, when we will need them.

To create a writable signal, call the `signal()` function (from `@angular/core`) with the value that it must contain initially. All the following examples create instances of `WritableSignal` which, as you can see, is a generic type specifying the type of the value contained in the signal.

```
// this signal can contain a number, and initially contains 42
const count = signal(42);

// this signal can contain a PonyModel, and initially contains Rainbow Dash
const rainbowDash = signal<PonyModel>({
  name: 'Rainbow Dash',
  color: 'blue'
});

// this signal can contain a PonyModel, and initially contains Pinkie Pie
const pinkiePie: WritableSignal<PonyModel> = signal({
  name: 'Pinkie Pie',
  color: 'red'
});

// this signal can contain a PonyModel or undefined, and initially contains undefined
const unknownPony = signal<PonyModel | undefined>(undefined);
```

A signal is both an object and a function. To get the value contained in the signal, you *call* the signal.

```
// let's read the value of the rainbowDash signal
const value: PonyModel = rainbowDash();
// and print it in the console as JSON
console.log(JSON.stringify(value));
```

To change the value of the signal (and thus notify Angular that this signal has a new value), you call its `set()` method:

```
rainbowDash.set({
  name: 'Rainbow Dash',
  color: 'yellow'
});
```

Now let's say you only want to change the color of Rainbow Dash. You might be tempted to read the value of the signal and then mutate it. Or even to mutate it and then set it back into the signal again. That's something you should never do :

```
// ❌ Don't do this!
rainbowDash().color = 'yellow';
// ❌ Don't do this!
const pony = rainbowDash();
pony.color = 'yellow';
rainbowDash.set(pony);
```

If you mutate the object contained in a signal, or if you set a signal with the same object it already contains, then Angular will consider that the value has not changed, which will lead to bugs.

Instead, treat the value as immutable, and replace the old value with a new one. The `update()` method, along with the destructuring operator, can be used to create a new value from the existing one:

```
// ✅ Do this!
rainbowDash.update(pony => ({ ...pony, color: 'yellow' }));
```

But it's just a different way of doing:

```
rainbowDash.set({ ...rainbowDash(), color: 'yellow' });
```

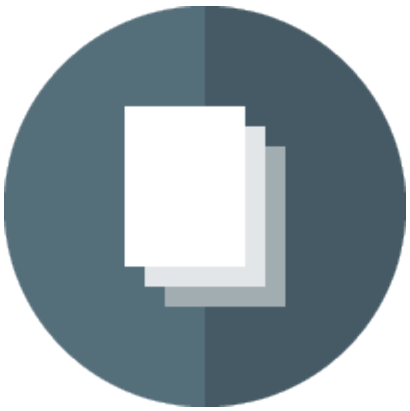
Let's now see how we can display the state of our components in a template!

Angular is not the first framework to use signals. Knockout.js, for example, already used signals a long time ago. More recently, Solid.js, Vue and Svelte also adopted signals as the building blocks of their state management. Signals are getting so popular, that the teams of these frameworks are even trying to standardize them into the JavaScript language itself!

# Chapter 10. 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, by modifying our first component:

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

@Component({
  selector: 'ns-root',
  template: '<h1>PonyRacer</h1>'
})
export class App {}
```

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: 'ns-root',
  template: '<h1>PonyRacer</h1>'
})
export class App {
  protected readonly numberOfUsers = 146;
}
```

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

## 10.1. Interpolation

Interpolation is a big word for a simple concept.

Quick example:

```
@Component({
  selector: 'ns-root',
  template: `
    <h1>PonyRacer</h1>
    <h2>{{ numberOfUsers }} users</h2>
  `
})
export class App {
  protected readonly numberOfUsers = 146;
}
```

We have an `App` component that will be activated every time Angular finds a `<ns-root>` tag. The `App` 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:

```
<ns-root>
  <h1>PonyRacer</h1>
  <h2>146 users</h2>
</ns-root>
```

as `{{ numberOfUsers }}` will be replaced by its value. When Angular detects a `<ns-root>` element in the page, it creates an instance of the `App` class, and this instance is the evaluation context of the template's expressions. Here the `App` 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 page will be automatically updated! Or rather, that's how it works now, with the "brute-force" change detection mechanism that Angular currently uses. But we've seen in the previous chapter that Angular now promotes using signals to help it detect changes in a more clinical way. If we wanted the number of users to change over time, we should thus store it in a signal, and call the signal in the template:

```
@Component({
  selector: 'ns-root',
  template: `
    <h1>PonyRacer</h1>
    <h2>{{ numberOfUsers() }} users</h2>
  `
})
export class App {
  protected readonly numberOfUsers = signal(146);
}
```

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

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

```
@Component({
  selector: 'ns-root',
  template: `
    <h1>PonyRacer</h1>
    <h2>Welcome {{ user().name }}</h2>
  `
})
export class App {
  protected readonly user = signal({ name: 'Cédric' });
}
```

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

```
<ns-root>
  <h1>PonyRacer</h1>
  <h2>Welcome Cédric</h2>
</ns-root>
```

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

```
@Component({
  selector: 'ns-root',
  // typo: users is not user!
  template: `
    <h1>PonyRacer</h1>
    <h2>Welcome {{ users().name }}</h2>
  `
})
export class App {
  protected readonly user = signal({ name: 'Cédric' });
}
```

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

```
error TS2339: Property 'users' does not exist on type 'App'
```

That's great, because now you are quite sure that your templates are correct.

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: 'ns-root',
  // the value of the user signal is undefined
  // but the ?. will avoid the error
  template: `
    <h1>PonyRacer</h1>
    <h2>Welcome {{ user()?.name }}</h2>
  `
})
export class App {
  protected readonly user = signal<{ name: string } | undefined>(undefined);
}
```

And you don't have errors anymore! The `?.` is called the "optional chaining operator". It was usable in Angular expressions even before being introduced in standard JavaScript and TypeScript.

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

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

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

## 10.2. Using other components in our templates

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

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

```
@Component({
  selector: 'ns-root',
  // added the Races component
  template: `
    <h1>PonyRacer</h1>
    <ns-races />
  `
})
export class App {}
```

As you can see, we added the `Races` 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? Angular doesn't know about this `Races` yet.

The fix is simple: you need to add `Races` to the `imports` of the `App` decorator. That way, when Angular compiles the template of `App`, it will look for a component with the `ns-races` selector in the list of imported components, and will thus know that it must instantiate and display the `Races`.

```
import { Component } from '@angular/core';
// do not forget to import the component
import { Races } from './races';

@Component({
  selector: 'ns-root',
  template: `
    <h1>PonyRacer</h1>
    <ns-races />
  `,
  // add Races to the imports of App
  imports: [Races]
})
export class App {}
```

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 `Races` in its source file `races.ts` (read the section about ES2015 modules again if that's not clear for you). So `Races` will look like:

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

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

```
export class Races {}
```

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

```
<ns-root>
  <h1>PonyRacer</h1>
  <ns-races>
    <h2>Races</h2>
  </ns-races>
</ns-root>
```

## 10.3. 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 you 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` returns `true`, and will not be selected if it returns `false`. And whenever the value of `isPonySelected` changes, the selected property will be updated.

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` signal 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 }}" />
```

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

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

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

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

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

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

All of these are equivalent. You need to remember that if an HTML attribute is not surrounded by square brackets (`name="…"`), then the value is a string. Whereas if it's surrounded by square brackets (`[name]="…"`), then the value is an expression, evaluated by Angular.

## 10.4. 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.

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

Reacting to an event can be done as follows:

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

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

Let's add this to our component:

```
@Component({
  selector: 'ns-races',
  template: `
    <h2>Races</h2>
    <button (click)="refreshRaces()">Refresh the races list</button>
    <p>{{ races().length }} races</p>
  `
})
export class Races {
  protected readonly races = signal<Array<RaceModel>>([]);

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

If you try this in your browser, you should initially see "0 races". 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.set('Cédric'); lastName.set('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 `Races` component emits a custom event to notify the app that a new race is available.

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

```
@Component({
  selector: 'ns-root',
  template: `
    <h1>PonyRacer</h1>
    <ns-races (newRaceAvailable)="onNewRace()" />
  `,
  imports: [Races]
})
export class App {
  protected onNewRace(): void {
    // add a flashy message for the user.
  }
}
```

We can easily figure out that the `<ns-races>` component has a custom event `newRaceAvailable` and that, when this event is fired, the method `onNewRace()` of our `App` 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: 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) {
  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 a crazy combo, like `(keydown.alt.space)`, etc.

To conclude this part, I want to point out 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 might suspect, they have different restrictions.

## 10.5. 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 cannot contain 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 semicolon. 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.

## 10.6. 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.

# 10.7. If, For and Switch with the control flow syntax

Now, our `Races` is still not displaying the races :) The "proper way" in Angular would mean creating another component `Race` 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 can use special instructions in the template, that have been introduced in Angular v18, under the name *Built-in control flow*.

Previously, we needed to use special directives, that are called structural directives, like `ngIf`, `ngFor`, `ngSwitch` to handle these cases. 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. Structural directives are a subset of directives that can change the structure of the DOM. They still exist, so you can still use `ngIf`, `ngFor`, `ngSwitch`, etc. in your templates, but they are deprecated and the recommended way is to use the new control flow syntax. If you're interested in the old way, you can read the chapter about structural directives later in this book.

Let's see how we can use the control flow syntax.

## 10.7.1. `@if`

Let's say we want to display a title in the template only if `races` has no element. We can use the `@if` instruction:

```
<div>
  @if (races().length === 0) {
    <h2>No races to come</h2>
  }
</div>
```

If we want to display something else if there are some races, we can use `@else`:

```
<div>
  @if (races().length === 0) {
    <h2>No races to come</h2>
  } @else {
    <h2>Some races to come</h2>
  }
</div>
```

It is also possible to use `@else if`:

```
<div>
  @if (races().length === 0) {
    <h2>No races to come</h2>
  } @else if (races().length === 1) {
    <h2>Only one race to come</h2>
  } @else {
    <h2>Some races to come</h2>
  }
</div>
```

You can also alias the result of the condition in a local variable, which can be useful if you want to use it several times.

```
@if (races().length; as raceCount) {
  <h2>{{ raceCount }} races to come</h2>
  <!-- displays "2 races to come" -->
}
```

**Tip for signals with nullable values**

Signals can sometimes contain a `null` or `undefined` value. This is usually the case when they are used to store an entity fetched from the backend. For example, this component has a `race` signal that contains a race or `undefined`:

```
protected readonly race = signal<RaceModel | undefined>(undefined);
```

In the template, if we use `race().name`, we get a TypeScript error indicating that `race()` can be `undefined`.

Usually, with a property that is not a signal, we can use a `@if` to display the name only the property is not `undefined`, which makes sense. But here it won't be enough to solve the error.

As a signal is a function, TypeScript has no way to know that the returned value is *still* not null inside the `if`.

In this case, it can be handy to use an alias to store the value of the signal.

```
@if (race(); as raceValue) {
  <h1>{{ raceValue.name }}</h1>
}
```

You can also use `@let` to solve this problem. We'll talk bout this in a few seconds.

### 10.7.2. `@for`

Working with real data will inevitably lead you to display a list of something. That's when `@for` proves very useful: it allows displaying a template for each item of a collection. Our `Races` component contains a field `races` which, as you can probably guess, is an array of races to display.

```
import { Component, signal } from '@angular/core';
import { RaceModel } from './race.model';

@Component({
  selector: 'ns-races',
  templateUrl: './for.html'
})
export class Races {
  protected readonly races = signal<Array<RaceModel>>([
    { id: 1, name: 'London' },
    { id: 2, name: 'Lyon' }
  ]);
}
```

The template uses `@for` to repeat the `li` tag for each item in the `races` array, and we chose to name the current item `race`:

```
<div>
  <h2>Races</h2>
  <ul>
    @for (race of races(); track race.id) {
      <li>{{ race.name }}</li>
    }
  </ul>
</div>
```

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

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

You'll note that `@for` requires to provide a `track` parameter. This is mainly for performance reasons, as it helps Angular to track the items in the collection and to update the DOM only when needed (we'll dive into this in a later chapter). You'll usually use a unique identifier for each item, like `track race.id`, but it can be the item itself (`track race`) if you don't have a better choice.

`@for` can be used with `@empty` to display something when the collection is empty (or `null` or `undefined`):

```
<div>
  <h2>Races</h2>
  <ul>
    @for (race of races(); track race.id) {
      <li>{{ race.name }}</li>
    } @empty {
      <li>No races</li>
    }
  </ul>
</div>
```

`@for` also exposes some variables that can be useful:

- `$index`, the index of the current item, starting at zero
- `$first`, a boolean that is true if the element is the first of the collection
- `$last`, a boolean that is true if the element is the last of the collection
- `$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

For example, you can use `$even` to add a `"grey"` CSS class to the even elements:

```
<ul>
  @for (race of races(); track race.id) {
    <li [class.grey]="$even">{{ race.name }}</li>
  }
```

You can alias these variables to a local variable if you need, which can be useful if you have nested loops:

```
<ul>
  @for (race of races(); track race.id; let isEven = $even) {
    <li [class.grey]="isEven">{{ race.name }}</li>
  }
```

### 10.7.3. `@switch`

As you can guess from its name, this instruction allows to switch between different templates based

on the value of an expression.

Let's take our `@if/@else if/@else` example and rewrite it using `@switch`:

```
<div>
  @switch (races().length) {
    @case (0) {
      <h2>No races to come</h2>
    }
    @case (1) {
      <h2>Only one race to come</h2>
    }
    @default {
      <h2>Some races to come</h2>
    }
  }
</div>
```

### 10.7.4. Migration from structural directives

If you have existing code using structural directives, you can migrate to the built-in control flow syntax using an automated migration, by running:

```
ng g @angular/core:control-flow
```

## 10.8. Template variables with `@let`

Angular v18.1 added a new feature to the template syntax: template variables. It is now possible to define a variable in the template, using the `@let` instruction, without having to declare it in the component class.

The syntax is simple: `@let variableName = expression;`. Let's say our component has a `count` field defined in its class, then we can define a variable in the template like this:

```
@let countPlusTwo = count() + 2;
<p>{{ countPlusTwo }}</p>
```

This can be handy when you want to use a value in several places in your template, especially if it is a complex expression. Sometimes you can create a dedicated field in the component class, but sometimes you can't, for example in a for loop:

```
@for (user of users(); track user.id) {
  <div class="name">{{ user.lastName }} {{ user.firstName }}</div>
  <div class="address">
    <span>{{ user.shippingAddress.default.number }} </span>
    <span>{{ user.shippingAddress.default.street }} </span>
```

```
    <span>{{ user.shippingAddress.default.zipcode }} </span>
    <span>{{ user.shippingAddress.default.city }}</span>
  </div>
}
```

This can be written more cleanly using `@let`:

```
@for (user of users(); track user.id) {
  <div class="name">{{ user.lastName }} {{ user.firstName }}</div>
  <div class="address">
    @let address = user.shippingAddress.default;
    <span>{{ address.number }} </span>
    <span>{{ address.street }} </span>
    <span>{{ address.zipcode }} </span>
    <span>{{ address.city }}</span>
  </div>
}
```

## 10.9. Structural directives

You can skip this section if you are using Angular v18 or later. It is still valid, but they are deprecated and the new syntax is recommended. We picked similar examples as the control flow syntax, so you can compare them.

The structural directives provided by Angular rely on using a `ng-template` element, inspired by the `template` standard tag of the HTML specification.

```
<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!

### 10.9.1. 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 that we discussed earlier: `CommonModule`. 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 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 have no special treatment: to be able to use them, a standalone component must import them. The following example explicitly imports `NgIf`. You might choose instead to import the whole `CommonModule`, which would make `NgIf` and other common directives and pipes available to the template.

```
import { Component, signal } from '@angular/core';
import { NgIf } from '@angular/common';
import { RaceModel } from './race.model';

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

It's also possible to use an `else` syntax:

```
import { Component, signal } from '@angular/core';
import { NgIf } from '@angular/common';
import { RaceModel } from './race.model';

@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>
  `,
  imports: [NgIf]
```

```
})
export class Races {
  protected readonly races = signal<Array<RaceModel>>([]);
}
```

### 10.9.2. NgFor

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

```
import { Component, signal } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RaceModel } from './race.model';

@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>
  `,
  imports: [CommonModule]
})
export class Races {
  protected readonly races = signal<Array<RaceModel>>([{ 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>
```

in the above example, which uses both `NgIf` and `NgFor`, we chose to import the whole `CommonModule`. If you prefer to be more explicit, you can instead have `NgIf` and `NgFor` in the `imports` array. It's mainly a matter of preferences.

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

```
<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

### 10.9.3. 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.

### 10.9.4. Understanding structural directives and their limitations

This content is really advanced, so you can skip it if you want: it is targeted at people who want to understand why the new syntax was introduced, and who have already a good understanding of Angular.

In Angular v18, the built-in control flow syntax (`@if`, `@for`, `@switch`) is promoted as a replacement for the venerable structural directives.

To understand why this new syntax was introduced, let's see how structural directives work in Angular.

### Structural directives under the hood

Structural directives are directives that change the structure of the DOM by adding, removing, or manipulating elements. They are easy to recognize in Angular because they begin with an asterisk `*`.

But how do they *really* work?

Let's take a simple template with `ngIf` and `ngFor` directives as an example:

```
<h1>Ninja Squad</h1>
<ul *ngIf="condition()">
  <li *ngFor="let user of users()">{{ user.name }}</li>
</ul>
```

If you read our chapter about the Angular compiler, you know that the framework generates JavaScript code from this template. And maybe you imagine that `*ngIf` gets converted to a JavaScript `if` and `*ngFor` to a `for` loop like:

```
createElement('h1');
if (condition()) {
  createElement('ul');
  for (user of users()) {
    createElement('li');
  }
}
```

But Angular does not work exactly like that: the framework decomposes the component's template into "views". A view is a fragment of the template that has static HTML content. It can have dynamic attributes and texts, but the HTML elements are stable.

So our example generates in fact three views, corresponding to three parts of the template:

```
<h1>Ninja Squad</h1>
<!-- special comment -->
```

```
<ul>
  <!-- special comment -->
</ul>
```

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

This is because the * syntax is in fact syntactic sugar to apply an attribute directive on an ng-template element. So our example is the same as:

```
<h1>Ninja Squad</h1>
<ng-template [ngIf]="condition()">
  <ul>
    <ng-template ngFor [ngForOf]="users()" let-user>
      <li>{{ user.name }}</li>
    </ng-template>
  </ul>
</ng-template>
```

Here ngIf and ngFor are plain directives. Each ng-template then generates a "view". Each view has a static structure that never changes. But these views need to be dynamically inserted at some point. And that's where the <!-- special comment --> comes into play.

Angular has the concept of ViewContainer. A ViewContainer is like a box where you can insert/remove child views. To mark the location of these containers, Angular uses a special HTML comment in the created DOM.

That's what ngIf actually does under the hood: it creates a ViewContainer, and then, when the condition given as input changes, it inserts or removes the child view at the location of the special comment.

This view concept is quite interesting as it will allow Angular to only update views that consume a signal in the future, and not the whole template of a component!

You can create your own structural directives if you want to. Let's say you want to write a *customNgIf directive. You can create a directive that takes a condition as an input and injects a ViewContainerRef (the service that allows to create the view) and a TemplateRef (the ng-template on which the directive is applied).

```
import {
  Directive,
  DoCheck,
  EmbeddedViewRef,
  inject,
  input,
  signal,
  TemplateRef,
  ViewContainerRef
} from '@angular/core';

@Directive({
  // eslint-disable-next-line @angular-eslint/directive-selector
```

```
  selector: '[customNgIf]'
})
export class CustomNgIf implements DoCheck {
  /**
   * The container where the view will be inserted
   */
  private readonly vcr = inject(ViewContainerRef);

  /**
   * The template to render
   */
  private readonly tpl = inject<TemplateRef<unknown>>(TemplateRef);

  /**
   * The condition to check
   */
  readonly condition = input.required<boolean>({ alias: 'customNgIf' });

  /**
   * The view created by the directive
   */
  readonly conditionalView = signal<EmbeddedViewRef<unknown> | null>(null);

  /**
   * This method is called every time the change detection runs
   */
  ngDoCheck() {
    // if the condition is true and the view is not created yet
    const conditionalView = this.conditionalView();
    if (this.condition() && !conditionalView) {
      // create the view and insert it in the container
      this.conditionalView.set(this.vcr.createEmbeddedView(this.tpl));
    } else if (!this.condition() && conditionalView) {
      // if the condition is false and the view is created
      // destroy the view
      conditionalView.destroy();
      this.conditionalView.set(null);
    }
  }
}
```

This works great! And as you can see, it lets developers like us create powerful structural directives if we want to: the built-in directives offered by Angular are not special in any way.

But this approach has some drawbacks: for example, it is a bit clunky to have an `else` alternative with `*ngIf`:

```
<div *ngIf="condition(); else elseBlock">If</div>
<ng-template #elseBlock><div>Else</div></ng-template>
```

`elseBlock` is another input of the `NgIf` directive, of type `TemplateRef`, that the directive will display if the condition is falsy. But this is not very intuitive to use, so we often see this instead:

```
<div *ngIf="condition()">If</div>
<div *ngIf="!condition()">Else</div>
```

The structural directives are also not perfect type-checking-wise. Even if Angular does some magic (with some special fields called `ngTemplateGuard` in the directives to help the type-checker), some cases are too tricky to handle. For example, the "else" alternative of `*ngIf` is not type-checked:

```
<div *ngIf="!adminUser; else userNotNullBlock">No user</div>
<ng-template #userNotNullBlock>
  <div>
    <!-- should compile as adminUser is not null here -->
    <!-- but it doesn't -->
    {{ adminUser.name }}
  </div>
</ng-template>
```

`NgSwitch` is even worse, as it consists of 3 separate directives `NgSwitch`, `NgSwitchCase`, and `NgSwitchDefault`. The compiler has no idea if the `NgSwitchCase` is used in the right context.

```
<!-- user.type can be `'user' | 'anonymous'` -->
<ng-container [ngSwitch]="user().type">
  <div *ngSwitchCase="'user'">User</div>
  <!-- compiles even if user.type can't be 'admin' -->
  <div *ngSwitchCase="'admin'">Admin</div>
  <div *ngSwitchDefault>Unknown</div>
</ng-container>
```

It's also worth noting that the `*` syntax is not very intuitive for beginners. So, to sum up, structural directives are powerful but have some drawbacks. Fixing these drawbacks would require a lot of work in the compiler and the framework.

That's why the Angular team decided to introduce a new syntax to write control flow statements in templates!

# 10.10. Template directives

Angular comes with plenty of other useful directives, but which are not structural. We'll use several of them when writing forms, for example. Two of them are commonly used to act on the CSS styles of the elements.

## 10.10.1. 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'`).

### 10.10.2. NgClass

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

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

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

or set several classes:

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

You can also use `ngClass`:

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

## 10.11. 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
- `@if/@for/@switch` for the control flow

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 `Ponies` component, displaying a list of ponies. Each pony should be represented by a `Pony` 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: ``,
  imports: []
})
export class Ponies {}
```

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

Our new component has a list of ponies wrapped in a signal:

```
import { Component, signal } from '@angular/core';
import { PonyModel } from '../pony.model';

@Component({
  selector: 'ns-ponies',
  template: ``,
  imports: []
})
export class Ponies {
  protected readonly ponies = signal<Array<PonyModel>>([
    { id: 1, name: 'Rainbow Dash' },
    { id: 2, name: 'Pinkie Pie' }
  ]);
}
```

We are going to display this list, using `@for`:

```
import { Component, signal } from '@angular/core';
import { PonyModel } from '../pony.model';

@Component({
  selector: 'ns-ponies',
```

```
  template: `
    <ul>
      @for (pony of ponies(); track pony.id) {
        <li>{{ pony.name }}</li>
      }
    </ul>
  `,
  imports: []
})
export class Ponies {
  protected readonly ponies = signal<Array<PonyModel>>([
    { id: 1, name: 'Rainbow Dash' },
    { id: 2, name: 'Pinkie Pie' }
  ]);
}
```

One thing is missing, the refresh button:

```
import { Component, signal } from '@angular/core';
import { PonyModel } from '../pony.model';

@Component({
  selector: 'ns-ponies',
  template: `
    <button (click)="refreshPonies()">Refresh</button>
    <ul>
      @for (pony of ponies(); track pony.id) {
        <li>{{ pony.name }}</li>
      }
    </ul>
  `,
  imports: []
})
export class Ponies {
  protected readonly ponies = signal<Array<PonyModel>>([
    { id: 1, name: 'Rainbow Dash' },
    { id: 2, name: 'Pinkie Pie' }
  ]);

  protected refreshPonies(): void {
    this.ponies.set([
      { id: 3, name: 'Fluttershy' },
      { id: 4, name: 'Rarity' }
    ]);
  }
}
```

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

```
import { Component, signal } from '@angular/core';
import { PonyModel } from '../pony.model';

@Component({
  selector: 'ns-ponies',
  template: `
    <button (click)="refreshPonies()">Refresh</button>
    <ul>
      @for (pony of ponies(); track pony.id) {
        <li [style.color]="$even ? 'green' : 'black'">
          {{ pony.name }}
        </li>
      }
    </ul>
  `,
  imports: []
})
export class Ponies {
  protected readonly ponies = signal<Array<PonyModel>>([
    { id: 1, name: 'Rainbow Dash' },
    { id: 2, name: 'Pinkie Pie' }
  ]);

  protected refreshPonies(): void {
    this.ponies.set([
      { id: 3, name: 'Fluttershy' },
      { id: 4, name: 'Rarity' }
    ]);
  }
}
```

As you can see, we have used all the range of the templating syntax, and we have a perfectly working component.

Try our quiz 🐴 and the 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 in building another component: the list of races.

# Chapter 11. Building components and directives

## 11.1. Introduction

So far, we have seen some small components. And of course, you can sense 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 best practices to build these things?

Directives: What do they do? Let's find out!

## 11.2. Directives

A directive is very much like a component, except it does not have a template. The goal of a component is to enrich HTML by using custom HTML elements like `<ns-pony />` to display a pony. Directives also enrich HTML by letting you attach a custom behavior to existing HTML elements. For example, you could attach a drag directive to a `<div>`, or a `<section>`, or even a `<ns-pony>` element to make it draggable.

We've already seen the `ngClass` directive: it allows adding or removing CSS classes to HTML elements. You can attach several directives to the same element. For example, a `<div>` could need CSS classes added to them by `ngClass`, and could also be made draggable by the drag directive.

The way we define directives is similar to the way we define components. We create a class. We decorate this class with a `@Directive` decorator. And we pass metadata to this decorator as we did for the `@Component` decorator. In fact, components are directives. All the metadata that we specify on a directive, we can also pass them to a component. Their lifecycles are also pretty much identical.

Even though directives are a big part of what makes Angular extremely powerful, they're more commonly used in low-level, foundational parts of libraries, or of the framework itself. A typical web application developer rarely creates directives, and more commonly creates components.

Let's learn about the most common things we can define on components and directives. More advanced stuff will be described later.

## 11.3. Selectors

The selector is what allows Angular to identify a component or directive inside HTML templates.

Selectors can be of various types:

- an element, as it's usually the case for components: `ns-pony`.
- 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 DoNothing {
  constructor() {
    console.log('Do nothing directive');
  }
}
```

Such a directive will be activated in a component like this `Test`:

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

A more complex selector could be:

```
@Directive({
  selector: 'div.loggable[logText]:not([notLoggable=true])'
})
export class ComplexSelector {
  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! 😶

CSS selectors like descendants, siblings, ids, wildcards, and pseudos (other than `:not`) are not supported.

## 11.4. Inputs with `input()`

Inputs are what allows a component or directive to receive information from its parent component. You can see them as parameters of a function: the calling function uses parameters to pass data to the called function.

To pass an input to a component or directive, we use the property binding syntax that we described in the previous chapter. For example, if we wanted to pass a color to the pony component, we would use:

```
<ns-pony color="blue" />
```

to pass the literal `'blue'` string, or:

```
<ns-pony [color]="selectedColor()" />
```

to pass the value of the Angular expression `selectedColor()`.

In order for a directive or component to accept information from its parent, it must define *inputs*.

Here's how the pony component can receive a color from its parent component:

```
@Component({
  selector: 'ns-pony',
  template: 'My color is {{ color() }}'
})
export class Pony {
  protected readonly color = input<string>();
}
```

The `color` property is a special kind of signal: an `InputSignal`. Such a signal is not writable. The `Pony` can't change its color. The only way to change its value is to bind another value from the parent component.

The input in the above example is optional. That means that the parent component is allowed to not pass a color to the component `<ns-pony />`. But the input is typed, so if it passes a value, the value must be a string. If the parent doesn't pass any color, the `color` signal will contain `undefined`. The type of `color` is `InputSignal<string | undefined>`.

The component could choose to use a default value for the color, in case the parent doesn't pass any. The type of `color` would then be `InputSignal<string>`: it always has a string value.

```
readonly color = input('red');
```

Or it could choose to force the parent to pass a value by making the input *required*. In this case, the parent component template would fail to compile if it didn't pass a value.

```
readonly color = input.required<string>();
```

If you wanted to use `color` as the property name, but prefer to use another name in the templates for the input, you can choose to give it an alias.

For example, if the input is defined that way:

```
readonly color = input.required<string>({ alias: 'c' });
```

then the parent will use `c` to pass a color value:

```
<ns-pony [c]="selectedColor()" />
```

The above examples all define an input of type `string`. But anything can be passed as input. It's very common to pass complex objects as inputs, such as a `PonyModel` object that would contain a name, a color, a birthdate, etc.

## 11.5. The `@Input` decorator

We've just seen how to define inputs as signals. That's the modern way of doing it. If you work on an existing code base, the inputs of your components are probably defined the old way, using the `@Input` decorator.

Inputs defined this way are passed by the parent component using the exact same property binding syntax. The difference is only in the component which defines the input. Such "legacy" inputs are simple properties (not signals), decorated with `@Input()`. Here are the above examples rewritten in the legacy way.

An optional input:

```
@Input() color: string | undefined;
```

An optional input with a default value:

```
@Input() readonly color = 'red';
```

A required input:

```
@Input({ required: true }) color!: string;
```

A required input with an alias:

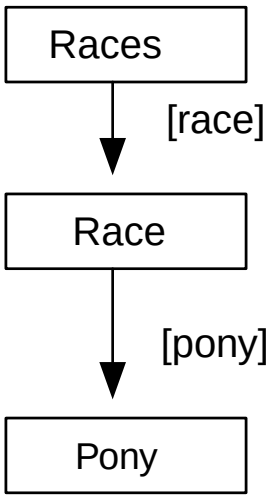
```
@Input({ required: true, alias: 'c' }) color!: string;
```

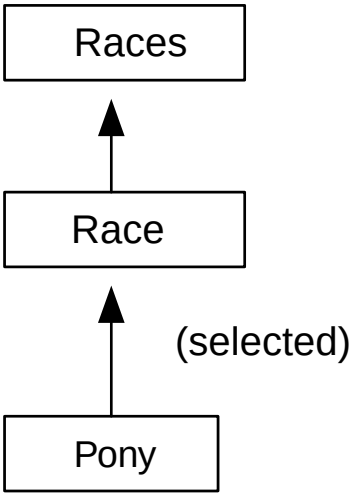
OK, now what about passing data up? We can't use properties to pass data from `Pony` to `Ponies`. But we can use events!

## 11.6. Outputs with `output()`

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.






Custom events are emitted using an `OutputEmitterRef`. You don't really need to use this awful type name. Such objects are created for you by the `output()` function.

You're free to choose the type of the events that you emit. It could be `undefined`, to just signal that something happened. Or it could be a `PonyModel`, to pass all the information about the pony that has just been selected, for example.

Let's say we want to emit an event called `ponySelected`. We have two things to do:

- define a property `ponySelected` initialized by calling `output()`
- call `this.ponySelected.emit()` to emit an event when the pony is being selected

```
@Component({
  selector: 'ns-pony',
  template: `
    <div>I'm the pony {{ ponyModel().name }}</div>
    <div><button (click)="selectMe()">Select me</button></div>
  `
})
export class Pony {
  readonly ponyModel = input.required<PonyModel>();

  // define the output
  readonly ponySelected = output<PonyModel>();
  // ☝OutputEmitterRef<PonyModel>

  /**
   * Selects the pony when the "Select me" button is clicked.
   * Emits a custom event of type PonyModel
   */
  protected selectMe() {
    this.ponySelected.emit(this.ponyModel());
```

```
  }
}
```

To use it in the template of the parent component:

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

In the above example, every time the user clicks on the button in the pony component, 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 emitted event. Here, it is the `PonyModel` object.

The parent component must then have a method `betOnPony()`, which will be called with the selected pony:

```
protected betOnPony(event: PonyModel) {
  // do something with the received event
}
```

As for the inputs, if you wish, you can choose an alias for the output:

```
readonly ponySelected = output<PonyModel>({ alias: 'activated' });
```

The parent component will then have to use this alias to listen to the event:

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

## 11.7. The `@Output` decorator

Even though outputs have nothing to do with signals, the `output()` function described above was introduced at the same time as the `input()` function, to have a pleasant symmetry between inputs and outputs.

Before that, outputs were defined using the `@Output` decorator. And they were also using a different class: `EventEmitter` which was unnecessarily tied to the RxJS library (we'll talk about this library in a few chapters). In case your codebase still uses them, here's how an output is defined in the legacy way:

```
@Output() readonly ponySelected = new EventEmitter<PonyModel>();
```

and here's how an alias can be specified for the input:

```
@Output('activated') ponySelected = new EventEmitter<PonyModel>();
```

## 11.8. Lifecycle

Components and directives have a lifecycle. When Angular must display a component, it starts by constructing it and thus calls the component class constructor. Then it will pass the initial input values to the component. If the expressions passed as inputs later have different values, then these new values are written to the inputs again. And finally, if the user navigates away from this component, for example, the component is destroyed. Hopefully, the component isn't reachable from anywhere anymore at this point, and the JavaScript virtual machine can garbage collect the component object.

Angular allows you to react to those various phases (and other ones, more advanced, that we'll discover later). One thing is very important to understand: inputs are passed **after** the component has been constructed. It's thus forbidden to read inputs while the component is being constructed.

With the signal inputs, trying to do that will lead to an exception being thrown by the `InputSignal`. With the legacy inputs, trying to do that will just give you the default value of the property, rather than the one actually bound by the parent component.

That means that the following component will not work:

```
export class Pony {
  readonly color = input.required<string>();

  constructor() {
    // ❌ Don't do this
    console.log(`My initial color is ${this.color()}`);
  }
}
```

If you want to access the value of an input, to load additional data from the server, for example, you can use a lifecycle hook. More powerful options are available for signal inputs, which we'll discover in the next chapter.

Several hooks are available:

- `ngOnChanges` is called when the value of one or several bound inputs change. It will receive a `changes` map, containing the current and previous values of the inputs, 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's guaranteed to be called even if there are no inputs. But if there are inputs, it's called after the first `ngOnChanges` call. It makes this phase perfect for initialization work that depends on input values, as the name suggests.
- `ngOnDestroy` is called when the component is destroyed. Really useful to do some cleanup.

Other phases are available but are for more advanced use cases. We will describe them in the Advanced components and directives chapter a bit further.

Our previous example will work better using `ngOnInit`. Angular invokes the method `ngOnInit()` if it's present, so you just have to implement it in your directive/component. But the best practice is to implement the `OnInit` interface. That forces you to implement the method, and make sure you have defined it correctly:

```
export class Pony implements OnInit {
  readonly color = input.required<string>();

  ngOnInit() {
    // ✅ Do this
    console.log(`My initial color is ${this.color()}`);
  }
}
```

Now we have access to our inputs!

If you want to do something every time an input changes, use `ngOnChanges`:

```
export class Pony implements OnChanges {
  readonly color = input.required<string>();

  ngOnChanges(changes: SimpleChanges): void {
    const ponyChange = changes['color'];
    console.log(`Color changed from ${ponyChange.previousValue}`);
    console.log(`to ${ponyChange.currentValue}`);
    console.log(`Is it the first change? ${ponyChange.isFirstChange()}`);

    // but you can also access the new value by reading the input
    console.log(`My new color is ${this.color()}`);
  }
}
```

The `changes` parameter is a record, with the input 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.

The `ngOnDestroy` phase is designed to clean the component – for example, to cancel background tasks. Here, the `Pony` 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:

```
export class Pony implements OnDestroy {
  readonly color = input.required<string>();
  private readonly interval: number;

  constructor() {
```

```
    this.interval = window.setInterval(() => console.log(`My color is ${this.color()}
`), 1000);
  }

  ngOnDestroy(): void {
    window.clearInterval(this.interval);
  }
}
```

If you don't do this, JavaScript will keep the instance of the component in memory, and it will log every second forever.

## 11.9. Component-specific metadata

Inputs, outputs, and the hooks we've just described are common to directives and components. Components, however, can define styles that will apply to their template, and **must** have an associated template.

## 11.10. 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.

## 11.11. Styles / Style 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 `styleUrl` (or `styleUrls` if you have several files).

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 `styleUrl` is a good idea. As the name of the latter suggests, you can specify an array of URLs.

```
@Component({
  selector: 'ns-pony',
  template: '<div class="pony">My color is {{ color() }}</div>',
  styles: ['.pony { background-color: lightgray; }']
})
export class Pony {
  readonly color = input.required<string>();
}
```

Try our exercise Race detail 🦄! This exercise will guide you in building a more

> advanced component with inputs.

# Chapter 12. Reacting to signal changes

We just discovered how to use `ngOnChanges` to react to changes in the value of inputs. This is very helpful, but it's a bit cumbersome when you need to only react to the changes of some inputs. And of course it's limited to inputs, and inputs only.

We've also learnt that signals have super-powers. Thanks to them, Angular knows that a view that reads a signal must be refreshed when the signal changes. We can also use those super-powers ourselves. Two functions allow us to do that: `computed()` and `effect()`.

## 12.1. Computed signals

Have you ever used a spreadsheet with formulas? It's great, isn't it? You can sum all the amounts of a *sales* column and store the result in a *total* cell. Then you can use another formula to apply the VAT to that total and store the result in yet another cell. If you add a sale, or if you change an amount, the total is automatically recomputed, and the VAT total as well. Computed signals allow doing the same thing.

Let's say our pony component receives a `PonyModel` as input, containing (among other properties) a name and a color. And let's say its template wants to display its identity as `NAME (color)` (for example "RAINBOW DASH (blue)").

The first thing that comes to mind is to do it this way in the template:

*pony.html*

```
<div>{{ ponyModel().name.toUpperCase() }} ({{ ponyModel().color }})</div>
```

This is absolutely fine. But if the application uses the "brute-force" change detection, this transformation will be done many times, even if the pony is the same as before. And if the template must display the identity several times, copy and pasting this is not the best idea.

We might improve that by delegating to a method in the component:

*pony.html*

```
<div>{{ identity(ponyModel()) }}</div>
```

This avoids the duplication, but it still invokes the `identity()` method many times unnecessarily.

A better way would be to use `ngOnChanges` to compute the identity when the `ponyModel` changes, and store the result in another property:

*pony.ts*

```
readonly ponyModel = input.required<PonyModel>();
protected readonly identity = signal<string>('');
```

```
ngOnChanges() {
  this.identity.set(`${this.ponyModel().name.toUpperCase()} (${this.ponyModel().color
})`);
}
```

That is nice. But if we later introduce another input, we'll recompute the identity whenever this unrelated input changes, unless we make our `ngOnChanges` method more complex. Also, note how we're forced to initialize the identity signal with a fake default value. There got to be an even better way.

The better way is to use a **computed** signal. A computed signal is a read-only signal, whose value is derived from another (or several other) signal(s):

```
readonly ponyModel = input.required<PonyModel>();
protected readonly identity = computed(() => `${this.ponyModel().name.toUpperCase()}
(${this.ponyModel().color})`);
```

The `identity` property is now a `Signal<string>`, whose value will always contain the identity of the pony contained in the `ponyModel` signal. There is a dependency that is created and tracked by Angular between the two signals. Whenever `ponyModel` changes, `identity` also changes.

Computed signals are *memoized* and *lazy*, which make them perfect for representing state that can be read many times. This means that the identity of a pony is *stored* inside the computed signal. And it's not recomputed every time we write a new pony, but only the next time the identity signal is read. You can try to execute the following example, and see what is being logged:

```
const count = signal(1);
const double = computed(() => {
  console.log(`computing double of ${count()}`);
  return count() * 2;
});

console.log(double());
count.set(2);
count.set(3);
count.set(4);
console.log(double());
console.log(double());
console.log(double());
```

The output will be:

```
'computing double of 1',
2,
'computing double of 4',
8,
8,
```

```
8
```

Computed signals are even greater when they depend on multiple signals. Let's say you have a form letting you enter a price and a quantity, and displaying a total, as well as a vat-included total. Programming that imperatively requires you to think about computing the total and the vat-included total whenever you change the price, and also whenever you change the quantity. Dependencies are not so obvious, and it can quickly become spaghetti code. Including yet another rule (like a discount percentage) could easily introduce a bug. Once the form input values are available as signals, computed signals make the logic very easy to implement, and very efficient as well:

```
const total = computed(() => price() * quantity());
const totalAfterDiscount = computed(() => total() * (1 - discountRate()));
const vatIncludedTotalAfterDiscount = computed(() => totalAfterDiscount() *
vatRate());
```

## 12.2. Effects

Effects are another way to react to signal changes. Instead of being used to compute a derived value, they're used to trigger a *side effect* (hence the name) when a signal changes. Examples of such a side effect are storing a value of a signal in local storage when a signal changes, or updating the DOM when a signal changes. It might be tempting to use an effect to update other signals when a source signal changes. That's possible, but it can lead to infinite loops, and computed signals are more suited to such a task.

To illustrate how effects work, we will just use one that logs to the console.

*pony.ts*

```
readonly ponyModel = input.required<PonyModel>();

constructor() {
  effect(() => console.log(`${this.ponyModel().name} - ${this.ponyModel().color}`));
}
```

This example will log the name and color of the pony every time the parent passes a pony as input.

Unlike computed signals, effects are bound to the component (or directive, or service, or pipe) in which they are constructed. An effect must be created while the component is being constructed. It's possible to do it elsewhere, but it's more complex, and we will learn about that later. The advantage of this restriction is that Angular takes care of destroying the effect when its owning component is destroyed.

Another feature that must be understood is that effects are not executed synchronously, whenever one of their source signals change. Instead, they're called asynchronously, before the change detection. This means that intermediate signal values won't be noticed by effects.

Let's illustrate this using the following component:

*score.ts*

```
protected readonly score = signal(0);

constructor() {
  effect(() => console.log(`The score is now ${this.score()}`));
}

protected plusOne() {
  this.score.update(s => s + 1);
}

protected plusTwo() {
  this.plusOne();
  this.plusOne();
}
```

Creating the component will log *"The score is now 0"*, because the effect function is always executed once initially to be able to know which signals the effect depends on.

Calling the `plusTwo()` method (as a reaction to a button click for example), will set the score signal to 1, then to 2. But only the final value of the score (i.e. the value that is stored in the signal after the click event has been handled), will be logged by the effect. So the console will only log *"The score is now 2"*.

There are still more things to learn about signals, computed and effect (such as the `untracked` function and the equality function, effect cleaning functions), but we will keep that for later.

You now know enough to start using signals. And hopefully, you understand why they are so great in helping us to manage the state of our components in a simple, expressive and concise way.

Try our exercise Pony component 🦄! This exercise will guide you in building a more advanced component with inputs, outputs and computed signals.

# Chapter 13. Styling components and encapsulation

Let's stop to talk about styles and CSS for a minute. I know right? Why talk 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 `styleUrl` or `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 styles. 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.ShadowDom`, which relies on Shadow DOM v1
- `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 `Pony`. 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, signal, 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 Pony {
```

```
  protected readonly name = signal('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.

## 13.1. Shadow DOM strategy

If you use the `ShadowDom` 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 Web Component specification. This specification allows you 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 `ShadowDom` strategy, you are sure that your component's styles are not "bleeding" into your child components if they are using the `ShadowDom` strategy as well. But they *will* bleed into the child components if they don't, which can be useful in some cases.

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!

As a side note: the component selector must be hyphenated otherwise the `ShadowDOM` strategy won't work.

## 13.2. Emulated strategy

As said earlier, this is the default strategy. And the reason is really simple: it emulates (hence the name) the `ShadowDom` strategy, but without using the Shadow DOM. It's safe to use everywhere and styles won't bleed into child components.

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 ShadowDom 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.

## 13.3. 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.

## 13.4. 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 ShadowDom 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 ShadowDom 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 14. Pipes

## 14.1. Pied piper

Sometimes the raw data is not what we want to display in the view. We often want to transform it, format it, 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' :).

Similarly as components, pipes can be standalone or not. The pipes that are provided by Angular and that we will discuss here are all standalone, and are all part of `CommonModule`. So, to use them in your components, you'll have to add them, or the whole `CommonModule`, to their `imports`.

Let's take an example and see how we can use pipes.

## 14.2. 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, a signal containing 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>
```

## 14.3. 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 a `@for`:

```
import { Component, signal } from '@angular/core';
import { SlicePipe } from '@angular/common';
```

```
import { PonyModel } from '../templates/pony.model';

@Component({
  selector: 'ns-ponies',
  template: `@for (pony of ponies() | slice: 0 : 2; track pony.id) {
    <div>{{ pony.name }}</div>
  }`,
  imports: [SlicePipe]
})
export class Ponies {
  protected readonly ponies = signal<Array<PonyModel>>([
    { id: 1, name: 'Rainbow Dash' },
    { id: 2, name: 'Pinkie Pie' },
    { id: 3, 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, signal } from '@angular/core';
import { SlicePipe } from '@angular/common';
import { PonyModel } from '../templates/pony.model';

@Component({
  selector: 'ns-ponies',
  template: `@for (pony of ponies() | slice: 0 : size(); track pony.id) {
    <div>{{ pony.name }}</div>
  }`,
  imports: [SlicePipe]
})
export class Ponies {
  protected readonly size = signal(2);
  protected readonly ponies = signal<Array<PonyModel>>([
    { id: 1, name: 'Rainbow Dash' },
    { id: 2, name: 'Pinkie Pie' },
    { id: 3, name: 'Fluttershy' }
  ]);
}
```

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

## 14.4. keyvalue

This pipe, introduced in Angular 6.1, allows you to iterate over a Map or an object, and to display

the keys/values in our templates.

Note that it orders the keys:

- first lexicographically if they are both strings
- then by their value if they are both numbers
- then by their boolean value if they are both booleans (`false` before `true`).

And if the keys have different types, they will be cast to strings and then compared.

```
@Component({
  selector: 'ns-ponies',
  template: `
    <ul>
      <!-- entry contains { key: number, value: PonyModel } -->
      @for (entry of ponies() | keyvalue; track entry) {
        <li>{{ entry.key }} - {{ entry.value.name }}</li>
      }
    </ul>
  `,
  imports: [KeyValuePipe]
})
export class Ponies {
  protected readonly ponies = signal(
    new Map<number, PonyModel>([
      [103, { name: 'Rainbow Dash' }],
      [56, { name: 'Pinkie Pie' }]
    ])
  );
}
```

If you have `null` or `undefined` keys, they will be displayed at the end.

It's also possible to define your own comparator function:

```
@Component({
  selector: 'ns-ponies',
  template: `
    <ul>
      <!-- entry contains { key: PonyModel, value: number } -->
      @for (entry of poniesWithScore() | keyvalue: ponyComparator; track entry) {
        <li>{{ entry.key.name }} - {{ entry.value }}</li>
      }
    </ul>
  `,
  imports: [KeyValuePipe]
})
export class Ponies {
  protected readonly poniesWithScore = signal(
```

```
    new Map<PonyModel, number>([
      [{ name: 'Rainbow Dash' }, 430],
      [{ name: 'Rainbow Dash' }, 680],
      [{ name: 'Pinkie Pie' }, 125]
    ])
  );

  /*
   * Defines a custom comparator to order the elements by the name of the PonyModel
(the key)
   */
  ponyComparator(a: KeyValue<PonyModel, number>, b: KeyValue<PonyModel, number>): -1 |
0 | 1 {
    if (a.key.name === b.key.name) {
      return 0;
    }
    return a.key.name < b.key.name ? -1 : 1;
  }
}
```

## 14.5. 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' -->
```

## 14.6. 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' -->
```

## 14.7. 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' -->
```

## 14.8. number

The following pipes (`number`, `percent`, `currency`, `date`) can help with

> internationalization. They have been completely overhauled in Angular 5.0, and don't use the Intl API of the browsers anymore (it caused numerous bugs). The Angular team has now implemented the internationalization logic themselves. The following examples use the new implementation of the pipes, which comes with Angular 5, without diving into the internationalization details, as they are covered in a chapter at the end of the book. The examples also use the default locale of Angular, `en-US`.

This pipe lets you format a number.

It takes one parameter, a string, formatted as `{integerDigits}.{minFractionDigits}-{maxFractionDigits}`, but every part is optional. Each part indicates:

- how many numbers you want in the integer part
- how many numbers you want at least in the decimal part
- how many numbers you want at most in the decimal part

A few examples, starting with what we have with no pipe:

```
<p>{{ 12345 }}</p>
<!-- will display '12345' -->
```

Using the `number` pipe will group the integer part, even with no digits required:

```
<p>{{ 12345 | number }}</p>
<!-- will display '12,345' -->
```

The `integerDigits` parameter will left-pad the integer part with zeros if needed:

```
<p>{{ 12345 | number:'6.' }}</p>
<!-- will display '012,345' -->
```

The `minFractionDigits` is the minimum size of the decimal part, so it will pad zeros on the right until reached:

```
<p>{{ 12345 | number:'.2' }}</p>
<!-- will display '12,345.00' -->
```

The `maxFractionDigits` is the maximum size of the decimal part. You have to specify a `minFractionDigits`, even at 0, if you want to use it. If the number has more decimals than that, then it is rounded:

```
<p>{{ 12345.13 | number:'.1-1' }}</p>
<!-- will display '12,345.1' -->
```

```
<p>{{ 12345.16 | number:'.1-1' }}</p>
<!-- will display '12,345.2' -->
```

## 14.9. percent

Based on the same principle as `number`, `percent` lets you display… a percentage!

```
<p>{{ 0.8 | percent }}</p>
<!-- will display '80%' -->

<p>{{ 0.8 | percent:'.3' }}</p>
<!-- will display '80.000%' -->
```

## 14.10. currency

As you can imagine, this pipe lets you 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, an option to say if you want to use the symbol ('€', '$', CA$') with `'symbol'` or the ISO code with `'code'`, or even the narrow symbol with `'symbol-narrow'`. The narrow symbol is for example `$`, when the symbol is `CA$` for Canadian dollars. The default value of this option is `'symbol'`.
- optionally also, a string to format the amount, using the same syntax as `number`.

```
<p>{{ 10.6 | currency:'CAD' }}</p>
<!-- will display 'CA$10.60' -->

<p>{{ 10.6 | currency:'CAD':'symbol-narrow' }}</p>
<!-- will display '$10.60' -->

<p>{{ 10.6 | currency:'EUR':'code':'.3' }}</p>
<!-- will display 'EUR10.600' -->
```

If you don't provide the ISO string representing the currency, then `USD` is used, unless you configure it globally (since Angular 9). Check the Internationalization chapter if you want to learn how.

This pipe is way more powerful than what you might expect: it relies on the ISO 4217 specification to determine the number of digits in the decimal part. For example, formatting an amount in Chilean pesos results in an amount with no digits (as pesos don't have cents), whereas formatting an amount in Tunisian dinars results in an amount with 3 digits (as it has millimes).

## 14.11. 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' -->
```

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 .

## 14.12. async

The `async` pipe allows data obtained asynchronously to be displayed. It can handle async data that comes from a Promise or an Observable. I hope you now know what a Promise is (otherwise go back to the ES2015+ chapter), and we'll come back to Observables quickly.

The async pipe returns `null` 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';
import { AsyncPipe } from '@angular/common';

@Component({
  selector: 'ns-greeting',
  template: `<div>{{ asyncGreeting | async }}</div>`,
  imports: [AsyncPipe]
})
export class Greeting {
  protected readonly 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 pipe is destroyed (for example when the user navigates to another component, or because it's part of an `ngIf` that becomes false).

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

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

@Component({
  selector: 'ns-user',
  template: `@if (asyncUser | async; as user) {
    <div>{{ user.name }}</div>
  }`,
  imports: [AsyncPipe]
})
export class User {
  protected readonly asyncUser = new Promise<UserModel>(resolve => {
    // after 1 second, the promise will resolve
    window.setTimeout(() => resolve({ name: 'Cédric' }), 1000);
  });
}
```

If you want to learn more about this pipe, check the Performances chapter near the end of this ebook.

## 14.13. A pipe in your code

It is also possible to use the formatting functions offered by the framework directly. For example, to format a number, we can use the `formatNumber` function:

```
import { Component, computed, signal } from '@angular/core';
// you need to import the function you want to use
import { formatNumber } from '@angular/common';

@Component({
```

```
  selector: 'ns-pony',
  template: `<p>{{ formattedSpeed() }}</p>`
})
export class Pony {
  protected readonly pony = signal({ name: 'Rainbow Dash', speed: 15 });
  protected readonly formattedSpeed = computed(
    // use the format function
    () => formatNumber(this.pony().speed, 'en-US', '.2')
  );
}
```

## 14.14. 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 { Pipe, PipeTransform } from '@angular/core';

export class FromNowPipe implements PipeTransform {
  transform(value: string, ..._args: Array<unknown>): string {
    // do something here
  }
}
```

We are going to use the `parseISO` function from date-fns to parse the date, and the `formatDistanceToNow` function to display how much time has elapsed since the date.

You can install date-fns using `NPM` if you want:

```
npm install date-fns
```

The types for date-fns are already included in the NPM dependency, so the TypeScript compiler should be happy without us doing anything.

```
import { Pipe, PipeTransform } from '@angular/core';
import { formatDistanceToNowStrict, parseISO } from 'date-fns';

export class FromNowPipe implements PipeTransform {
  transform(value: string, ..._args: Array<unknown>): string {
    const date = parseISO(value);
```

```
    return formatDistanceToNowStrict(date, { addSuffix: true });
  }
}
```

Now, we need to tell Angular that this class is a pipe. For this, there is a special decorator we can use: `@Pipe`.

```
import { Pipe, PipeTransform } from '@angular/core';
import { formatDistanceToNowStrict, parseISO } from 'date-fns';

@Pipe({
  name: 'fromNow'
})
export class FromNowPipe implements PipeTransform {
  transform(value: string, ..._args: Array<unknown>): string {
    const date = parseISO(value);
    return formatDistanceToNowStrict(date, { addSuffix: true });
  }
}
```

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 `imports` of the component using it.

```
@Component({
  selector: 'ns-race',
  template: 'The race started {{ race().startInstant | fromNow }}',
  imports: [FromNowPipe]
})
export class Race {
  protected readonly race = signal({
    startInstant: '2023-02-10T10:00:00.000Z'
  });
}
```

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, the Custom pipe with date-fns 🦄 exercise will make you build an awesome custom pipe!

# Chapter 15. Dependency injection

## 15.1. 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 advantages:

- 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 letting us replace an implementation of a service by another one.

It's a concept vastly used on the server side, but not so much on the frontend side, except in Angular.

## 15.2. 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 for injection into components (or other services, or pipes, or directives).
- a way to specify what dependencies must be injected in the current component or service.

The framework does the rest of the job. When we ask for 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 finally 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 a `LoggingService` service. In development, we want to log to the browser's console. In production, we want to aggregate logs on a remote server. Let's start by the

development version.

```
export class LoggingService {
  log(message: string): void {
    console.log(message);
  }
}
```

It's easy to inject that dependency in a component or a service. We just have to use the type system and the `inject` function of Angular.

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

```
import { inject } from '@angular/core';
import { LoggingService } from './logging-service';

export class RaceService {
  private readonly loggingService = inject(LoggingService);
}
```

The `inject` function tells Angular to fetch the `LoggingService` service and returns it.

Be careful: this `inject` function can not be called anywhere we want. It needs to be called in an *injection context*. Calling it to initialize a property (as we're doing in the above example) or as part of the constructor is fine. But calling it after the object has been constructed will throw an exception.

In fact, `inject` is a relatively new way of injecting dependencies. Before it was introduced in Angular 14, the only way to inject a dependency was to declare the dependency as an argument of the constructor. So we could also write our `RaceService` this way:

```
import { LoggingService } from './logging-service';

export class RaceService {
  constructor(private loggingService: LoggingService) {}
}
```

Constructor injection is still very much supported, but `inject` is now generally seen as a preferred option, and has opened the door to interesting API improvements.

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

```
import { inject } from '@angular/core';
import { RaceModel } from './race.model';
import { LoggingService } from './logging-service';
```

```
export class RaceService {
  private readonly loggingService = inject(LoggingService);

  list(): Array<RaceModel> {
    this.loggingService.log('race-service: get races');
    // ...
  }
}
```

In Angular, all services must be decorated with `@Injectable()`:

```
import { inject, Injectable } from '@angular/core';
import { RaceModel } from './race.model';
import { LoggingService } from './logging-service';

@Injectable()
export class RaceService {
  private readonly loggingService = inject(LoggingService);

  list(): Array<RaceModel> {
    this.loggingService.log('race-service: get races');
    // ...
  }
}
```

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

The easiest (and recommended) way to do this is to add the `@Injectable` decorator on `LoggingService` and use `providedIn` directly inside:

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

@Injectable({
  providedIn: 'root'
})
export class LoggingService {
  log(message: string): void {
    console.log(message);
  }
}
```

Another way to do this is to use the `providers` option of `bootstrapApplication()` function we saw earlier:

```
bootstrapApplication(App, { providers: [LoggingService] }).catch(err => console.error
(err));
```

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

```
import { inject, Injectable } from '@angular/core';
import { RaceModel } from './race.model';
import { LoggingService } from './logging-service';

@Injectable({
  providedIn: 'root'
})
export class RaceService {
  private readonly loggingService = inject(LoggingService);

  list(): Array<RaceModel> {
    this.loggingService.log('race-service: get races');
    // ...
  }
}
```

And we're done!

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

```
export class App {
  private readonly raceService = inject(RaceService);
  protected readonly races: Array<RaceModel> = this.raceService.list();
}
```

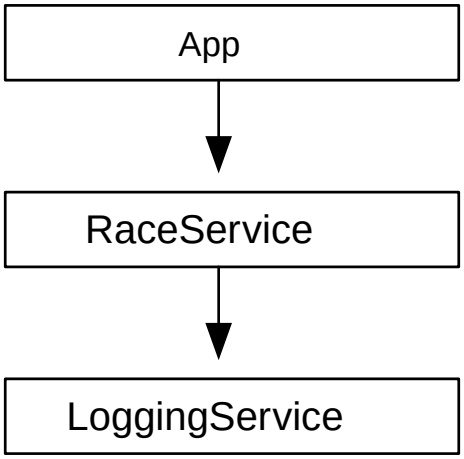
As we want to use an API to log our traces in production, we need to provide a different implementation of the `LoggingService` service.

## 15.3. Easy to configure

I'll come back to the testability advantages brought by dependency injection in a following chapter. We can give an example of how DI makes the application easy to configure, though. We want to call a logging API in production instead of simply logging to the console.

DI provides a nice way to do this.

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:

```
bootstrapApplication(App, {
  providers: [{ provide: LoggingService, useClass: LoggingService }]
}).catch(err => console.error(err));
```

We are telling the `Injector` that we want to create a link between a token (the type `LoggingService`) and the class `LoggingService`. The `Injector` is a service which keeps track of the injectable elements by maintaining a registry and is actually injecting them when needed. For the moment, you can see this registry as a map that associates keys, called tokens, with classes. A token can be anything. But it is usually a class reference (as in the above example) or an instance of `InjectionToken`.

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

```
bootstrapApplication(App, { providers: [LoggingService] }).catch(err => console.error
(err));
```

The token has to uniquely identify the dependency.

We can use the `inject` function to test how the injector works:

```
// in our bootstrapApplication function
providers: [
  { provide: LoggingService, useClass: LoggingService },
  // let's add another provider to the same class
  // with another token
  // A token can be declared like this:
  // const token = new InjectionToken<LoggingService>('LoggingServiceToken');
  { provide: token, useClass: LoggingService }
```

```
]
```

```
console.log(inject(LoggingService));
// logs "LoggingService"
console.log(inject(token));
// logs "LoggingService" again
console.log(inject(LoggingService) === inject(LoggingService));
// logs "true", as the same instance is returned every time for a token
console.log(inject(LoggingService) === inject(token));
// 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 a token. I have provided the `LoggingService` twice, with two different tokens. The injector will create an instance of `LoggingService` the first time it is asked to for a specific token, and then return the same instance for this token every time.

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 for the same token;
- the token used to identify a service does not have to be the class of the service.

The fact that the same service instance is used every time we ask for a service is also a well-known design pattern: it's called a *singleton*. This is really useful, because it allows a service to hold state that you want to share between several components.

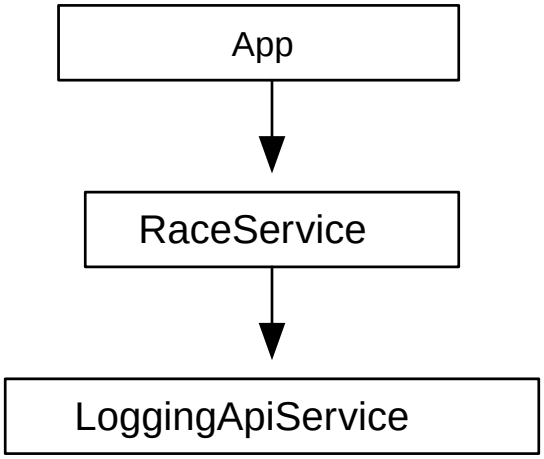
Now, back to our `LoggingService`. I can write a new class, doing the same job as `LoggingService` but this time calling an API to aggregate the logs:

```
@Injectable()
export class LoggingAPIService {
  log(message: string): void {
    // ... calls the logging API
  }
}
```

We can use the provider declaration to replace `LoggingService` with our `LoggingAPIService`:

```
// in our bootstrapApplication function
providers: [
  // we provide a different implementation of the service
  { provide: LoggingService, useClass: LoggingAPIService }
]
```

Now we have a relation like this:




That can be really useful to replace an implementation with another in certain environments and, as we will see soon, when you are writing automated tests.

## 15.4. Other types of provider

In our example, we might want to use `LoggingService` when we are developing our app, and use the real `LoggingAPIService` when we are in production. One way of doing that is to use another type of provider: `useFactory`.

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

  // in our bootstrapApplication function
  providers: [
    RaceService,
    // we provide a factory
    {
      provide: LoggingService,
      useFactory: () => (IS_PROD ? new LoggingAPIService() : new LoggingService())
    }
  ]
```

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 race service with the development or production logging service.

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

```
// in our bootstrapApplication function
providers: [RaceService, { provide: LoggingService, useClass: IS_PROD ?
```

```
LoggingAPIService : LoggingService }]
```

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`.

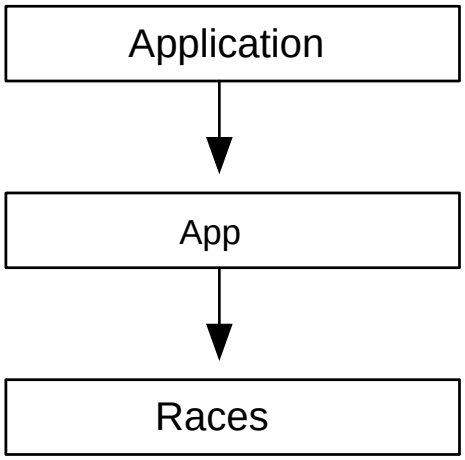
```
export const IS_PROD = new InjectionToken<boolean>('IsProd');
```

```
// in our bootstrapApplication function
providers: [
  { provide: IS_PROD, useValue: true },
  RaceService,
  {
    provide: LoggingService,
    useFactory: () => {
      const isProd = inject(IS_PROD);
      return isProd ? new LoggingAPIService() : new LoggingService();
    }
  }
]
```

## 15.5. 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.

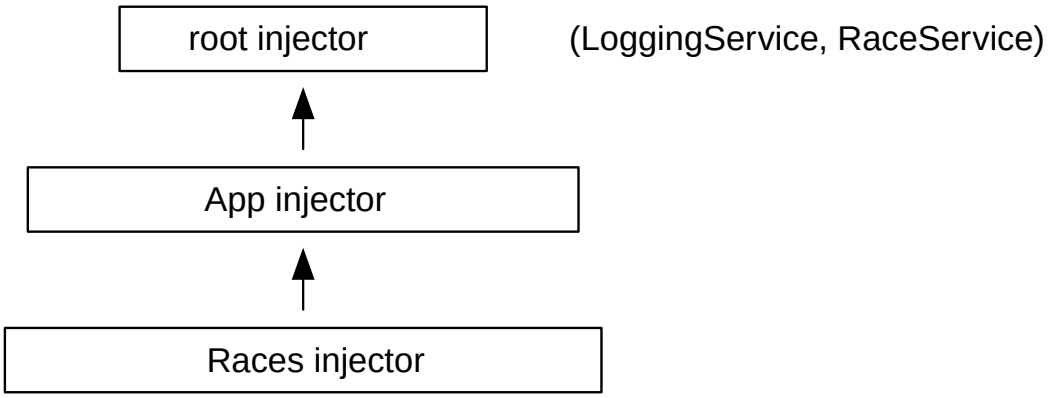
Let's say we have an app looking like:




We have an application with a root component `App`, which has a child component `Races`.

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

When you register a service using the recommended `providedIn: 'root'`, you add these services to the root injector.




It means that when we inject a dependency in a component, Angular begins 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 find the dependency anywhere, it throws an exception.

From this, we can deduce two things:

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

The `@Component` decorator can take another configuration option, called `providers`. This `providers` attribute expects an array of providers, similar the `providers` option of `bootstrapApplication()`.

We can thus imagine a `Races` that would declare its own `LoggingService` provider:

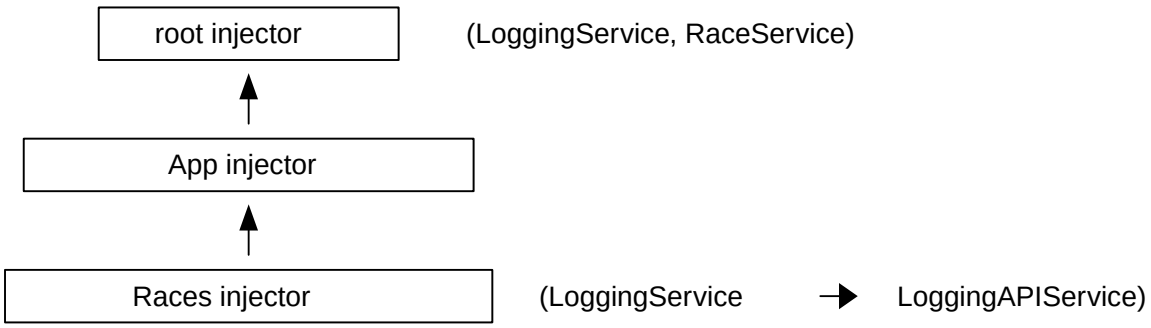
```
@Component({
  selector: 'ns-races',
  providers: [{ provide: LoggingService, useClass: LoggingAPIService }],
  template: `<strong>Races</strong>`
})
export class Races {
  constructor() {
    inject(LoggingService).log('Races created');
  }
}
```

In this component, the provider with the token `LoggingService` will always give an instance of `LoggingAPIService`, regardless of what was defined in the root injector. It's really useful if you want

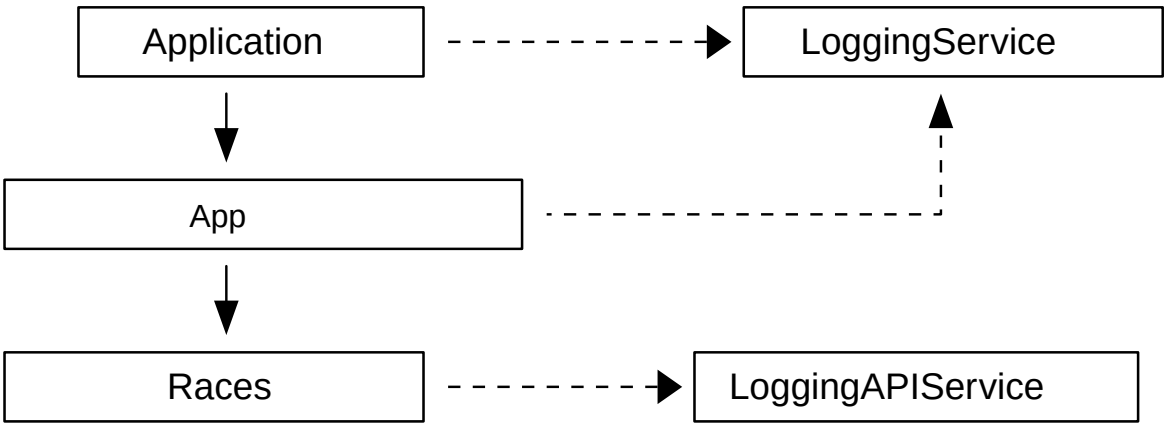
to have a different instance of a service for a given component.

If you declare a dependency at the root 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:




If the service contains state that belongs to a specific component instance, then it should be provided by that component. For stateless services, or services that contain global state, they should be provided in root.

## 15.6. DI without types

We've seen above that we can define tokens to identify services. We've also seen that there are two ways to get a dependency: calling the `inject` function, or declaring the dependency as an argument of the constructor.

When a token is used to identify a service, we can pass this token as argument to the `inject` function to get the associated service.

It's also possible to get the service associated with the token when using constructor injection, thanks to the `@Inject()` decorator.

```
import { Inject, Injectable } from '@angular/core';
import { BACKEND_URL } from './tokens';

@Injectable({
  providedIn: 'root'
})
export class RaceService {
  constructor(@Inject(BACKEND_URL) private url: string) {}
}
```

The above example uses a token `BACKEND_URL` which identifies a simple value stored in the injector. This token is defined like this:

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

export const BACKEND_URL = new InjectionToken<string>('API URL');
```

We then use this token in the providers of the application to define its value:

```
{ provide: BACKEND_URL, useValue: 'http://localhost:8080' }
```

Or you can register it directly with `providedIn`:

```
export const BACKEND_URL_PROVIDED = new InjectionToken<string>('API URL', {
  providedIn: 'root',
  factory: () => 'http://localhost:8080'
});
```

Angular itself uses this token mechanism, and allows us to define the locale of the application for example, by using the token `LOCALE_ID`:

```
bootstrapApplication(App, {
  providers: [
    { provide: LOCALE_ID, useValue: 'fr-FR' }
  ]
}).catch(err => console.error(err));

@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' -->
  `,
  imports: [DecimalPipe]
})
export class CustomLocale {
  protected readonly locale = inject(LOCALE_ID);
}
```

But we'll talk about it later when we see how to internationalize your application!

# 15.7. Services provided by the framework

Angular comes with a few built-in services. Some of them will be discussed in dedicated chapters. For now, let's talk about two of them.

## 15.7.1. Title service

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, inject } from '@angular/core';

@Component({
  selector: 'ns-root',
  template: `<h1>PonyRacer</h1>`
})
export class App {
  constructor() {
    inject(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!

## 15.7.2. Meta service

The other service we'll talk about is somewhat similar: it allows you to get or update the "meta" values of the page.

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

@Component({
  selector: 'ns-root',
  template: `<h1>PonyRacer</h1>`
```

```
})
export class App {
  constructor() {
    inject(Meta).addTag({ name: 'author', content: 'Ninja Squad' });
  }
}
```

Try our exercise Race service 🦄 to build your first service.

# Chapter 16. Reactive Programming

## 16.1. Call me maybe

The basic principle of reactive programming is to try to define the logic in a declarative way, by defining events and telling how to react to those events, instead of doing it in an imperative way. This sounds very abstract, and it is. So let's try to illustrate this with a basic example.

Let's say you have a circle, and you need to store its radius as well as its circumference. And you want to be able to change the radius of the circle. Doing that should of course change its circumference accordingly. One way (the imperative way) of doing this is to use two properties:

```
private radius = 0;
private circumference = 0;

setRadius(r: number) {
  this.radius = r;
  this.circumference = 2 * Math.PI * r;
}
```

Every time you change the radius, you need to remember that you also need to change the circumference accordingly.

Another way of doing this is to rely on a reactive primitive, and to *declaratively* specify that the circumference must be recomputed every time the radius changes. Changing the radius can thus be seen as an event, and we can react to this event by changing the circumference. The reactive primitive we could use here is a signal:

```
private readonly radius = signal(0);
private readonly circumference = computed(() => 2 * Math.PI * this.radius());

setRadius(r: number) {
  this.radius.set(r);
}
```

What if we had a component taking the ID of a race as input, and we wanted this component to show the details of that race? Things get more complicated here, because getting the details of the race is an asynchronous operation. We can't just use a computed signal to "compute" a race based on its ID. Besides, if the ID changes two times in a row, the fetching of the first race should be cancelled before fetching the second one: we're not interested anymore in the first one.

We might also have to send two different HTTP requests to get the full details of the race, and we would then need to wait until we have both responses before displaying it.

Being able to declaratively express that every time an input changes, we have to make two asynchronous calls, cancel the two previous ones if any, wait for the two calls to complete, and

finally display the result, would be great.

That's the kind of stuff for which RxJS shines. It provides another reactive primitive: an `Observable`, which represents a stream of synchronous or asynchronous events. It also comes with plenty of functions and operators that allow transforming, filtering, combining these streams of events.

Angular, since its first version, has chosen to depend on RxJS, and uses observables to represent events and things that change over time. For example, the result of a call to the `HttpClient` is an `Observable`. The changes of parameters of a route are modeled as an `Observable`. The changes of the value of a form are modeled as an `Observable`.

RxJS is not a free meal though. It's quite complex to learn and master. Thinking in a reactive way doesn't come naturally to most developers. And Angular has signals now, which provide another reactive primitive that sometimes overlaps with what RxJS provides. So the general direction that Angular has decided to follow is to try not to depend on RxJS anymore, but to make the usage of RxJS inside Angular applications as smooth as possible, by providing interoperability functions.

## 16.2. RxJS

An `Observable` is a stream of events, that has a well-defined, albeit flexible lifecycle.

When you **subscribe** to an observable, it starts emitting events. It can emit 0, 1, N or even an infinity of events. The emission can be synchronous or asynchronous. The events can be anything (an HTTP response, a form value, etc.): it depends on the type of the Observable.

An observable can signal its **completion**. After the completion, you know that it won't emit any event anymore. And you're thus also unsubscribed: why would you keep listening to something that won't say anything anymore?

In case something goes wrong (for example, you subscribed to send an HTTP request and get an HTTP response but the server is down), an observable can also signal an **error**. After an error, you also have the guarantee that it won't emit any event anymore (nor signal a completion). And you're thus also unsubscribed.

You can create various kinds of observables using RxJS functions. Many RxJS functions are used to transform an observable into another one. Those functions are called *operators*.

For example:

- `take(n)` creates an observable that only emits the first `n` events of the source observable and then completes;
- `map(fn)` creates an observable that transforms each event of the source observable (using the function `fn`) and emits the result;
- `filter(predicate)` creates an observable that only emits the events of the source observable that fulfill the predicate.

There are much more operators than these. We would need a whole book to go through all of them. But hopefully you get an idea of what operators can do. If you want to have a good visual representation of what the RxJS functions and operators do, go to rxmarbles.com, or consult the

official RxJS documentation.

So, if you have an observable of numbers and want to multiply each by 2, then filter those under 5, and print them, you can do:

```
import { filter, from, map, Observable } from 'rxjs';
```

```
const numbers$: Observable<number> = from([1, 2, 3, 4, 5]).pipe(
  map(x => x * 2),
  filter(x => x > 5)
);
numbers$.subscribe(x => console.log(x));
// Logs 6, 8, 10
```

The $ suffix is a common convention that allows remembering that the variable is an Observable.

The observable in the above example emits synchronously: as soon as we subscribe, the numbers are emitted one by one. When the call to `subscribe` returns, all the numbers have already been printed to the console.

But observables are usually asynchronous. They can for example represent DOM events that will happen in the future:

```
import { fromEvent } from 'rxjs';
```

```
const input = document.querySelector('input')!;

fromEvent(input, 'keyup').subscribe(() => console.log('keyup!'));

input.trigger('keyup'); // Logs "keyup!"
input.trigger('keyup'); // Logs "keyup!"
```

When subscribing to such an observable, the function you pass will be executed later, every time the user releases a key in the text field. If you're not interested anymore in those events, you'll have to unsubscribe. The simplest way to do that is to call `unsubscribe` on the `Subscription` returned by `subscribe`. There are other ways to unsubscribe, that will be discussed later.

```
const subscription = fromEvent(input, 'keyup').subscribe(() => console.log('keyup!'));
// later, when not interested anymore
subscription.unsubscribe();
```

You might also have to handle errors. The `subscribe` method accepts an object as an argument instead of a simple callback function. This object can define a callback to handle events (named

`next`), and a callback to handle errors (named `error`).

Here the mapping function throws an exception, so the error callback will log it.

```
range(1, 5)
  .pipe(
    map(x => {
      if (x % 3 === 0) {
        throw new Error('something went wrong');
      } else {
        return x;
      }
    })
  )
  .subscribe({
    next: x => console.log(x),
    error: error => console.log(error)
  });
// Logs 1, 2, something went wrong
```

Finally, if the observable completes, you can know about it too by providing a third handler. Here, the `range` function creates an observable which emits events events from 1 to 5 and then signals its completion:

```
range(1, 5)
  .pipe(
    map(x => x * 2),
    filter(x => x > 5)
  )
  .subscribe({
    next: x => console.log(x),
    error: error => console.log(error),
    complete: () => console.log('done')
  });
// Logs 6, 8, 10, done
```

## 16.3. Signals and RxJS interoperability

Now, how can we use RxJS to declaratively display the details of a race whenever the race ID changes? The race ID is an input, which is a signal. But to use RxJS, we would need to have an `Observable`, not a signal. No issue. Angular provides a `toObservable()` function that transforms a `Signal<T>` into an `Observable<T>`. Internally, it uses an effect to know when the signal changes, and make the observable emit its new value.

So we can do:

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

```
import { toObservable } from '@angular/core/rxjs-interop';
import { Observable } from 'rxjs';

@Component({
  selector: 'ns-race',
  template: '...'
})
export class Race {
  readonly raceId = input.required<number>();
  private readonly raceId$: Observable<number> = toObservable(this.raceId);
}
```

We now need to fetch the details of the race whenever the observable emits, and cancel the previous ongoing fetch if any. That's a job for the `switchMap` operator. We'll talk about it in more details in a future chapter.

```
import { Component, inject, input } from '@angular/core';
import { toObservable } from '@angular/core/rxjs-interop';
import { Observable, of, switchMap } from 'rxjs';

@Component({
  selector: 'ns-race',
  template: '...'
})
export class Race {
  private raceService = inject(RaceService);
  readonly raceId = input.required<number>();
  private readonly race$: Observable<RaceModel> = toObservable(this.raceId).pipe(
    switchMap(raceId => this.raceService.get(raceId))
  );
}
```

Now that's fine, but how can we display the race? We'll need to subscribe to the observable, and store the emitted race in a signal so that the template can display it. We could do that:

```
import { Component, inject, input, signal } from '@angular/core';
import { toObservable } from '@angular/core/rxjs-interop';
import { Observable, of, switchMap } from 'rxjs';

@Component({
  selector: 'ns-race',
  template: '...'
})
export class Race {
  private raceService = inject(RaceService);
  readonly raceId = input.required<number>();
  private readonly race$: Observable<RaceModel> = toObservable(this.raceId).pipe(
    switchMap(raceId => this.raceService.get(raceId))
```

```
  );

  // ❌ don't do this
  protected readonly race = signal<RaceModel | undefined>(undefined);
  constructor() {
    this.race$.subscribe(race => this.race.set(race));
  }
}
```

But then we should also unsubscribe when the component is destroyed. There's a better way. Angular also provides a function to transform an `Observable<T>` into a `Signal<T | undefined>`. You guessed it, it's named `toSignal`.

```
import { Component, inject, input, Signal } from '@angular/core';
import { toObservable, toSignal } from '@angular/core/rxjs-interop';
import { Observable, of, switchMap } from 'rxjs';

@Component({
  selector: 'ns-race',
  template: '...'
})
export class Race {
  private raceService = inject(RaceService);
  readonly raceId = input.required<number>();
  private readonly race$: Observable<RaceModel> = toObservable(this.raceId).pipe(
    switchMap(raceId => this.raceService.get(raceId))
  );

  // ✅ Do this instead
  protected readonly race: Signal<RaceModel | undefined> = toSignal(this.race$);
}
```

Internally, `toSignal` subscribes to the observable. And it automatically unsubscribes when the component is destroyed.

Are you wondering why it creates a `Signal<RaceModel | undefined>` and not a `Signal<RaceModel>`? Well, the answer is logical. An observable is usually asynchronous. It's the case here. So it only emits its first race once it has received the response from the HTTP server. Meanwhile, there's no race that can be stored in the signal, so its value is `undefined`. You can pass an option to `toSignal` if you prefer a having something else than `undefined` as the default value, or if you know that the observable emits synchronously.

Beware: `toObservable` and `toSignal` need to be called in an *injection context*, because they call `inject`. So, just as `inject`, you can only call them while the component is being constructed. You can actually call them from anywhere, but then you'll have to pass an `Injector` as an option. The `Injector` can be obtained, you guessed it, using dependency injection.

> Try our quiz 🐴 and the 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 17. Testing your app

## 17.1. 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 verify that 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 you 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.

## 17.2. Unit tests

As we saw earlier, unit tests are there to check a small unit of code in isolation. These tests can only verify 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 a very efficient way to test (nearly) all your code, especially the tricky cases, which can be hard to manually test in the real app.

One of the core concepts 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 purposes.

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!

### 17.2.1. 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: number): boolean {
    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.

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 test.

```
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.

For example, let's say we have a `Race` class with a `start()` method, that calls `run()` on every pony in the race, and filters the ponies that did not started running (`run()` returns a boolean):

*Race.ts*

```
class Race {
  constructor(private ponies: Array<Pony>) {}

  start(): Array<Pony> {
    return (
      this.ponies
        // start every pony
        // and only keeps the ones that started running
        .filter(pony => pony.run(10))
    );
  }
}
```

We want to test the `start()` method, and see if it properly calls `run()`. So we spy on the `run()` method of all the ponies in the race:

*Race.spec.ts*

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

```
  let rainbowDash: Pony;
  let pinkiePie: Pony;
  let race: Race;

  beforeEach(() => {
    rainbowDash = new Pony('Rainbow Dash');
    // first pony agrees to run
    spyOn(rainbowDash, 'run').and.returnValue(true);

    pinkiePie = new Pony('Pinkie Pie');
    // second pony refuses to run
    spyOn(pinkiePie, 'run').and.returnValue(false);

    // create a race with these two ponies
    race = new Race([rainbowDash, pinkiePie]);
  });

});
```

and test if the methods are called:

*Race.spec.ts*

```
it('should make the ponies run when it starts', () => {
  // start the race
  const runningPonies: Array<Pony> = race.start();
  // should have called `run()` on the ponies
  expect(pinkiePie.run).toHaveBeenCalled();
  // with a speed of 10
  expect(rainbowDash.run).toHaveBeenCalledWith(10);
  // as one pony refused to start, the result should be an array of one pony
  expect(runningPonies).toEqual([rainbowDash]);
});
```

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.

### 17.2.2. 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.

```
@Injectable({
  providedIn: 'root'
})
export class RaceService {
  list(): Array<RaceModel> {
    const race1: RaceModel = { name: 'London' };
    const race2: RaceModel = { name: '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 `inject`.

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

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

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

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

That works, because the service is declared with `providedIn: 'root'`, which make it available in the test. It will be instantiated and injected lazily when needed in the test.

As we did in the simple Jasmine example, we can maybe move the `RaceService` initialization into a `beforeEach` method. We can also use `TestBed.inject` in a `beforeEach`, so let's do it:

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

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

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

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

We moved the `TestBed.inject` logic in a `beforeEach` and now our test is pretty clean.

## 17.3. Fake dependencies

The `TestBed` class will help us to declare fake dependencies.

Even though Angular allows creating applications without defining Angular modules, its `TestBed` utility is still designed around the concept of a testing module, that can be configured pretty much the same way as an Angular module. Since we're using standalone components, the main purpose of that testing module will be to provide fake dependencies instead of the actual ones. The `TestBed.configureTestingModule` method allows you to specify an array of `providers`, that will supersede the actual dependencies injected in our component, pipe, directive or service under test.

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({
  providedIn: 'root'
})
export class RaceService {
  private readonly localStorage = inject(LocalStorageService);

  list(): Array<RaceModel> {
    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`:

```
export class MockLocalStorage {
  get(_key: string): Array<RaceModel> {
    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: MockLocalStorage }]
    })
  );

  it('should return 2 races from localStorage', () => {
    const service = TestBed.inject(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 with a fake one. It also allows you 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>('LocalStorageService
', ['get']);

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

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

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

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

## 17.4. Testing components

The next step after testing a simple service is to test a component. A component test is slightly different because we want to test not only the code inside the TypeScript class, but also the template of the component.

Let's start by writing a component to test. Why not our `Pony` 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]="ponyImageUrl()" [alt]="ponyModel().name"
(click)="clickOnPony()" /> `
})
export class Pony {
  readonly ponyModel = input.required<PonyModel>();
  readonly running = input(false);
  protected readonly ponyImageUrl = computed(
    () => `/images/pony-${this.ponyModel().color.toLowerCase() + (this.running() ? '
-running' : '')}.png`
  );
  readonly ponyClicked = output<PonyModel>();

  protected clickOnPony(): void {
    this.ponyClicked.emit(this.ponyModel());
  }
```

```
}
```

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. As our component has inputs, it is fairly common to create a wrapper component in the test, and to create an instance of this test component (so we can easily pass inputs ou test the outputs). To do this, we use the `TestBed`. This class comes with a utility method, named `createComponent`, to create a component. The method returns a `ComponentFixture`, a representation of our component.

```
import { TestBed } from '@angular/core/testing';
import { Component, signal } from '@angular/core';
import { Pony, PonyModel } from './pony';

@Component({
  imports: [Pony],
  template: `<ns-pony [ponyModel]="ponyModel()" (ponyClicked)="betOnPony($event)" />`
})
class PonyTest {
  readonly ponyModel = signal<PonyModel>({ id: 1, name: 'Rainbow Dash', color: 'BLUE'
});
  readonly betPony = signal<PonyModel | undefined>(undefined);

  betOnPony(event: PonyModel) {
    this.betPony.set(event);
  }
}

describe('Pony', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({});
  });

  it('should have an image', () => {
    // given a pony component
    const fixture = TestBed.createComponent(PonyTest);
    fixture.detectChanges();

    // when we get the image displayed
    const element = fixture.nativeElement as HTMLElement;
    const imageElement = element.querySelector('img')!;

    // then we should have an image with the correct source attribute
    // depending on the pony color
    expect(imageElement.getAttribute('src')).toContain('/images/pony-blue.png');
    expect(imageElement.getAttribute('alt')).toBe('Rainbow Dash');
  });
```

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 set up the test context. We get the test component instance created with a pony. It emulates an input that would come from a parent component in the real app. We then 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.
- a "When" phase, where we get the image element.
- 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', () => {
  // given a pony component
  const fixture = TestBed.createComponent(PonyTest);
  fixture.detectChanges();

  // when we click on the pony
  const element = fixture.nativeElement as HTMLElement;
  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(fixture.componentInstance.betPony()).toBe(fixture.componentInstance.
ponyModel());
});
```

Let's have a look at another component:

```
@Component({
  selector: 'ns-race',
  template: `
    <div>
      <h1>{{ raceModel().name }}</h1>
      @for (currentPony of raceModel().ponies; track currentPony) {
        <ns-pony [ponyModel]="currentPony" />
      }
    </div>
  `,
  imports: [Pony]
})
export class Race {
  protected readonly raceModel = input.required<RaceModel>();
```

```
}
```

and its test:

```
@Component({
  imports: [Race],
  template: `<ns-race [raceModel]="raceModel()" />`
})
class RaceTest {
  readonly raceModel = signal<RaceModel>({
    name: 'Paris',
    ponies: [{ id: 1, name: 'Rainbow Dash', color: 'BLUE' }]
  });
}
describe('Race', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({});
  });

  it('should have a name and a list of ponies', () => {
    const fixture = TestBed.createComponent(RaceTest);
    // given a component instance with a race input initialized
    fixture.componentInstance.raceModel.set({
      name: 'London',
      ponies: [{ id: 1, 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 as HTMLElement;
    expect(element.querySelector('h1')!.textContent).toBe('London');

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

Here we query all the directives of type `Pony` 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 `Pony`. 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).

## 17.5. Testing with fake templates, providers...

When testing a component, we sometimes want to create a *test host component* that uses it. That allows testing that the properties and outputs bindings work well. Let's take our pony component for example. In order to test that we can provide a `running` input, or that we can omit it to use its default value, we should test it with a parent component passing the `running` input, and also test it with a parent component not passing it.

Fortunately, the `TestBed` allows overriding the template of the test host component (or any other component, by the way):

```
import { Component, signal } from '@angular/core';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { Pony, PonyModel } from './pony';

@Component({
  selector: 'ns-test-host',
  template: '',
  imports: []
})
class TestHost {
  protected readonly pony = signal<PonyModel>({
    id: 1,
    name: 'Rainbow Dash',
    color: 'BLUE'
  });
}

describe('Pony', () => {
  let fixture: ComponentFixture<TestHost>;

  beforeEach(() => {
    TestBed.configureTestingModule({});
  });

  it('should display a non-running pony by default', () => {
    // given a test host component where the running input is not passed
    TestBed.overrideTemplate(TestHost, '<ns-pony [ponyModel]="pony()" />');
    TestBed.overrideComponent(TestHost, {
      add: {
        imports: [Pony]
      }
    });
    fixture = TestBed.createComponent(TestHost);

    // when we trigger the change detection
```

```
    fixture.detectChanges();

    // then we should have a not running pony
    const element = fixture.nativeElement as HTMLElement;
    expect(element.querySelector('img')!.src).toContain('/images/pony-blue.png');
  });

  it('should display a running pony if the running input is set to true', () => {
    // given a test host component where the running input is not passed
    TestBed.overrideTemplate(TestHost, '<ns-pony [ponyModel]="pony()" [running]="true"
/>');
    TestBed.overrideComponent(TestHost, {
      add: {
        imports: [Pony]
      }
    });
    fixture = TestBed.createComponent(TestHost);

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

    // then we should have a not running pony
    const element = fixture.nativeElement as HTMLElement;
    expect(element.querySelector('img')!.src).toContain('/images/pony-blue-
running.png');
  });
});
```

We can go further than that. It's also possible to call `TestBed.overrideComponent()` to set, add, or remove any property of the decorator of a component (`template`, `providers`, `imports`, etc.). That is sometimes useful, for example to test a parent component with a stub child component instead of an actual child component, in order to make the test simpler. We could for example replace the `Pony` in the imports of `Race` by a `PonyStub` which has the same selector, inputs and outputs, but which does nothing.

Now you're ready to test your app!

## 17.6. Simpler, cleaner unit tests with `ngx-speculoos`

Angular tests can quickly be very verbose. As we don't really like that, we wrote a tiny open-source library called ngx-speculoos.

Instead of writing a test looking like this:

```
let fixture: ComponentFixture<User>;

beforeEach(() => {
  fixture = TestBed.createComponent(User);
  fixture.detectChanges();
```

```
});

it('should display French cities when selecting the country France', () => {
  const countrySelect = (fixture.nativeElement as HTMLElement).querySelector
<HTMLSelectElement>('#country')!; // countrySelect is of type any
  expect(countrySelect.selectedIndex).toBe(0);
  countrySelect.selectedIndex = 2; // what is at index 2?
  countrySelect.dispatchEvent(new Event('change')); // why do I need to do that?
  fixture.detectChanges();

  const city = (fixture.nativeElement as HTMLElement).querySelector<HTMLSelectElement
>('#city')!; // city is of type any
  expect(city).toBeTruthy();
  expect(city.options.length).toBe(3);
  expect(city.options[0].value).toBe('');
  expect(city.options[0].label).toBe('');
  expect(city.options[1].value).toBe('PARIS');
  expect(city.options[1].label).toBe('Paris');
  expect(city.options[2].value).toBe('LYON');
  expect(city.options[2].label).toBe('Lyon');
});

it('should hide cities when selecting the empty country option', () => {
  const countrySelect = (fixture.nativeElement as HTMLElement).querySelector
<HTMLSelectElement>('#country')!; // I did that previously. What about DRY?
  countrySelect.selectedIndex = 0;
  countrySelect.dispatchEvent(new Event('change')); // why do I need to do that?
  fixture.detectChanges(); // why do I need to do that?

  expect((fixture.nativeElement as HTMLElement).querySelector('#city')).toBeFalsy();
// I did that previously. What about DRY?
});
```

you can write a cleaner and simpler test with `ngx-speculoos`:

```
class UserComponentTester extends ComponentTester<User> {
  constructor() {
    super(User);
  }

  get country(): TestSelect {
    return this.select('#country')!; // returns a TestSelect object, not any. Similar
methods exist for inputs, buttons, etc.
  }

  get city(): TestSelect {
    return this.select('#city')!; // returns a TestSelect object, not any
  }
}
```

```
let tester: UserComponentTester;

beforeEach(() => {
  tester = new UserComponentTester();
  tester.detectChanges();
});

it('should display French cities when selecting the country France', async () => {
  await tester.country.selectLabel('France'); // no dispatchEvent, no detectChanges
needed

  expect(tester.city.optionValues).toEqual(['', 'PARIS', 'LYON']);
  expect(tester.city.optionLabels).toEqual(['', 'Paris', 'Lyon']);
});

it('should hide cities when selecting empty country option', async () => {
  await tester.country.selectIndex(0); // no repetition of the selector, no
dispatchEvent, no detectChanges needed

  expect(tester.city).toBeFalsy(); // no repetition of the selector
});
```

You can go one step further with the custom matcher for Jasmine we wrote:

```
beforeEach(() => jasmine.addMatchers(speculoosMatchers));

it('should contain a pre-populated form', () => {
  expect(tester.informationMessage).toContainText('Please check that everything is
correct');
  expect(tester.country).toHaveSelectedValue('');
  expect(tester.name).toHaveValue('Doe');
  expect(tester.newsletter).toBeChecked();
});
```

Give it a try!

## 17.7. 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 can be 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 great coverage, and some guarantees that your complete application runs as intended.

Angular CLI doesn't have a default solution for E2E tests. After all, these tests don't even need to know that the application is built with Angular, so you can choose the tool you want. Some tools can be integrated inside the CLI though, so that you can run `ng e2e` to serve the app and then run the end-to-end tests.

The most popular tools are probably Cypress and Playwright. Nowadays, our preference goes to Playwright, which is free, well maintained by Microsoft, and can run tests in parallel on the three major browsers (Chrome, Firefox and Safari).

### 17.7.1. Playwright

Playwright is full of really nice features:

- easy to set up
- easy to mock HTTP responses
- easy to test different viewports (awesome for responsive applications)
- nice API
- UI mode or headless mode
- downloads and tests on the 3 major browsers
- tests run in parallel and in isolation

- automatic retries

The time-travel debugging is the feature that won my heart: Playwright takes a snapshot at each step of your tests, so you can debug very easily. Just by hovering the step of the failing test in the Playwright UI, you see exactly the state of the application and can play with it.

Playwright tests provide a `page` object, with a few utility methods like `goto()`, to navigate to a URL. Then you have `locator()` to select elements using various strategies: by CSS selector, by text contained in the element, by associated label, by role, etc.

Once you have a locator for an element, you can interact with it: `click()`, `check()`, `fill()`, etc.

And of course you can also perform assertions: `toBeVisible()`, `toBeEnabled()`, `toContainText()`, etc.

Here is what a test for a login page could look like:

```
test('should display an alert if login fails', async ({ page }) => {
  // mock the http response to the login request (optional)
  await page.route('**/api/users/authentication', async route => {
    await route.fulfill({
      status: 401
    });
  });

  // navigate to the login page
  await page.goto('/login');

  // get the login input, the password input and the submit button
  const loginInput = page.locator('input').first();
  const passwordInput = page.locator('input[type=password]');
  const submitButton = page.locator('form > button');

  // fill in the form
  await loginInput.fill('ced');
  await passwordInput.fill('pa');

  // submit the form and await the response
  const response = page.waitForResponse('**/api/users/authentication');
  await submitButton.click();
  await response;

  // test the URL of the page and the presence of an alert message
  await expect(page).toHaveURL('/login');
  await expect(page.locator('.alert-danger')).toContainText('Nope, try again');
});
```

These tests can be quite long to write, but they are really useful, and cover a lots of things at once.

They can also be great for things that aren't easily doable in unit tests, like taking a screenshot of a chart or map and comparing it with a reference image, pixel by pixel:

```
test('should display the history of the user score in a chart ', async ({ page }) => {
  // navigate to the score page
  await page.goto('/score-history');

  // take a screenshot of the canvas and compare it with a reference screenshot
  await page.locator('canvas').toHaveScreenshot('user-score.png', { maxDiffPixelRatio:
0.005 });
});
```

You can also use the Axe plugin to perform all kinds of automated accessibility tests on the page you're visiting, such as checking that the contrast is sufficient, or that all form elements have a proper label, etc.

With unit tests and e2e tests, you have the keys to build a robust and maintainable application!

All our Pro Pack exercises come with unit and e2e 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 18. Send and receive data through HTTP

It won't come as a surprise, but a big part 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 library to send HTTP requests.

One of the possibilities is the fetch API, which is a standard API provided by the 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 most Angular developers will rather use a service coming with Angular: `HttpClient`.

If you want to use it, you have to use the classes and functions from the `@angular/common/http` package.

Why prefer this service over, say, `fetch`? The answer is simple: testing. As we will show, the Http client allows you to mock your backend server and return fake responses. That's really, really useful.

A last thing before we dive into the API: the Http client 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 ;).

## 18.1. Getting data (`provideHttpClient`)

The `@angular/common/http` module offers a service called `HttpClient` that you can inject in any constructor. This service isn't available by default in an Angular application. You need to configure the application to use it.

To do this, we need to configure a provider when bootstrapping the application.

```
import { bootstrapApplication } from '@angular/platform-browser';
import { provideHttpClient, withInterceptors } from '@angular/common/http';

bootstrapApplication(App, {
  providers: [provideHttpClient()]
}).catch(err => console.error(err));
```

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

```
@Injectable({
  providedIn: 'root'
})
export class RaceService {
  private readonly http = inject(HttpClient);
```

By default, the `HttpClient` service will do AJAX requests 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 available 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<Array<RaceModel>>(`${baseUrl}/api/races`)
```

Note that you don't need to deserialize the response body from a string to a JavaScript array or object. That is done automatically by Angular. However, Angular won't make any check to verify that the JSON in the response indeed conforms to the generic type you specified. It's up to you to make sure that you're using the correct generic type, and that the `RaceModel` interface indeed matches with the JSON that the server sends back.

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

The most typical use case is to have a service with methods that return an observable:

```
@Injectable({
  providedIn: 'root'
})
export class RaceService {
  private readonly http = inject(HttpClient);

  list() {
    return this.http.get<Array<RaceModel>>(`${baseUrl}/api/races`);
  }
```

And then subscribe to this observable in a component using `toSignal`:

```
protected readonly races = toSignal(inject(RaceService).list());
```

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<Array<RaceModel>>(`${baseUrl}/api/races`, { observe: 'response' })
  .subscribe((response: HttpResponse<Array<RaceModel>>) => {
    console.log(response.status); // logs 200
    console.log(response.headers.keys()); // 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()` or `put()` method, with the URL and the object to

post:

```
http
  .post<RaceModel>(`${baseUrl}/api/races`, newRace)
```

Once again, no need to serialize the race object being sent to JSON. Angular does that for you. The generic type `RaceModel` here is, just as with the `get()` method, the type of the response body. So this example endpoint takes a `RaceModel` as input and returns the created `RaceModel`.

I won't show you the other methods - I'm sure you get the idea.

## 18.2. 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.

```
@Injectable({
  providedIn: 'root'
})
export class RaceService {
  private readonly http = inject(HttpClient);

  list() {
    return this.http.get<Array<RaceModel>>(`${baseUrl}/api/races`);
  }
```

You can also leverage the power of RxJS to retry a failed request a few times, for example.

```
raceService
  .list()
  .pipe(
    // if the request fails, retry 3 times
    retry(3)
  )
```

## 18.3. 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 = {
```

```
  sort: 'ascending',
  page: '1'
};

http
  .get<Array<RaceModel>>(`${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 = { Authorization: `Bearer ${token}` };

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

## 18.4. 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:

```
export const githubAPIInterceptor: HttpInterceptorFn = (
  req: HttpRequest<unknown>,
  next: HttpHandlerFn
): Observable<HttpEvent<unknown>> => {
  // 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(clone);
  }
  // if it's not a Github API request, we just hand it to the next handler
  return next(req);
};
```

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

Then configure the HTTP client provider so that it uses the interceptor:

```
providers: [
  provideHttpClient(withInterceptors([githubAPIInterceptor])),
]
```

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:

```
export const errorHandlerInterceptor: HttpInterceptorFn = (
  req: HttpRequest<unknown>,
  next: HttpHandlerFn
): Observable<HttpEvent<unknown>> => {
  const router = inject(Router);
  const errorHandler = inject(ErrorHandler);
  return next(req).pipe(
    // we catch the error
    tap({
      error: (errorResponse: HttpErrorResponse) => {
        // if the status is 401 Unauthorized
        if (errorResponse.status === HttpStatusCode.Unauthorized) {
          // we redirect to login
          router.navigateByUrl('/login');
        } else {
          // else we notify the user
          errorHandler.handle(errorResponse);
        }
      }
    })
  );
};
```

This is one of the cases where the `inject()` function needs to be used: since the interceptor is defined as a function, there is no constructor allowing to inject the dependencies, but the `inject()` function allows getting them.

## 18.5. Context

Sometimes you want to give some context to an interceptor. Since Angular v12, it is possible thanks to `HttpContext`. The context uses a type safe token (`HttpContextToken`), so you can define something like this in your interceptor:

```
export const SHOULD_NOT_HANDLE_ERROR = new HttpContextToken<boolean>(() => false);
```

And change the interceptor to:

```
export const errorHandlerInterceptor: HttpInterceptorFn = (
  req: HttpRequest<unknown>,
  next: HttpHandlerFn
): Observable<HttpEvent<unknown>> => {
  const router = inject(Router);
  const errorHandler = inject(ErrorHandler);
  // if there is a context specifically asking for not handling the error, we don't
handle it
  if (req.context.get(SHOULD_NOT_HANDLE_ERROR)) {
    return next(req);
  }
  return next(req).pipe(
    // ...
```

All HTTP methods accept a `context` option, which is a Map that you can build in a type-safe way by using the token defined previously:

```
const context = new HttpContext().set(SHOULD_NOT_HANDLE_ERROR, true);
return http.get(`${baseUrl}/api/users`, { context });
```

## 18.6. Tests

We now have a service calling an HTTP endpoint to fetch the races. How do we test it?

```
@Injectable({
  providedIn: 'root'
})
export class RaceService {
  private readonly http = inject(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 { TestBed } from '@angular/core/testing';
```

```
import { HttpTestingController, provideHttpClientTesting } from
'@angular/common/http/testing';
import { HttpClient, provideHttpClient } from '@angular/common/http';

describe('RaceService', () => {
  let raceService: RaceService;
  let http: HttpTestingController;

  beforeEach(() =>
    TestBed.configureTestingModule({
      providers: [provideHttpClient(), provideHttpClientTesting()]
    })
  );

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

  afterEach(() => {
    http.verify();
  });

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

    // call the service
    let actualRaces: Array<RaceModel> = [];
    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!

Try our exercise HTTP 🦄 and our quiz 🦄! 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 🦄 and Bet on a pony 🦄. Slightly related, we'll also use WebSockets 🦄.

# Chapter 19. 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: the creation of meaningful URLs reflecting the state of our app, and each URL knowing what 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 when working 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!

## 19.1. En route (`provideRouter`)

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

Similarly to the HTTP client, you have to provide the router in your application 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 { Home } from './home/home';
```

```
import { Races } from './races/races';

export const routes: Routes = [
  { path: '', component: Home },
  { path: 'races', component: Races }
];
```

Then we need to provide the router in our application, initialized with the proper configuration:

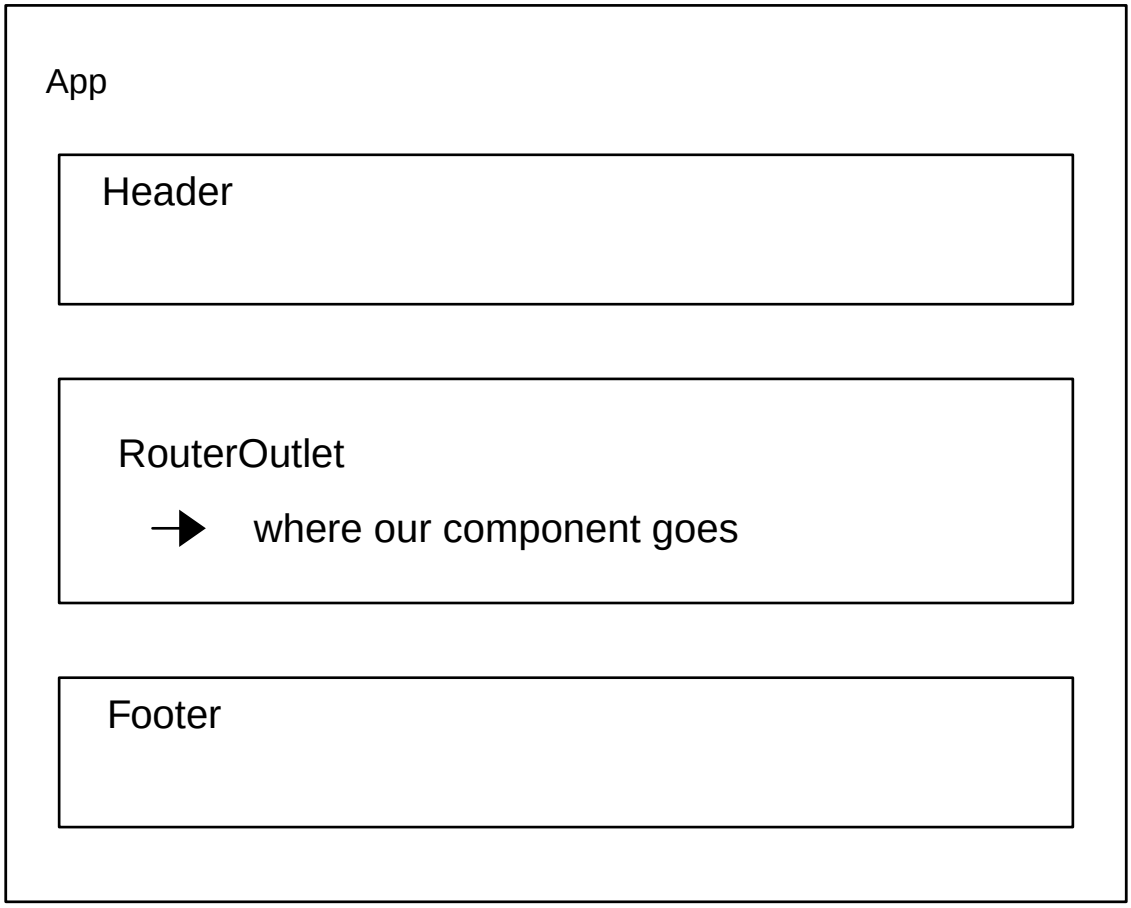
```
import { bootstrapApplication } from '@angular/platform-browser';
import { provideRouter } from '@angular/router';
import { routes } from './app.routes';

bootstrapApplication(App, {
  providers: [provideRouter(routes)]
}).catch(err => console.error(err));
```

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 `Races` 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 />
  <!-- 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.

All the directives of the router module, including `RouterOutlet`, are *standalone* directives. In order to be able to use them in the template, the component must have them in the `imports` of its decorator. Or you can add the whole `RouterModule` to the `imports` to have all the router directives available in the component template.

## 19.2. 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 built-in solution.

In a template, you can insert a link with the directive `RouterLink` pointing to the path you want to go to. 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 `Races` template, if we want to navigate to the `Home`, we can imagine something like:

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

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

The leading slash in the path is necessary. If not included, `RouterLink` builds the URL relatively to the current path (which 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 you, for example, to style a menu item as selected when it points to the current page.

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

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

```
<a 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 Races {
  private readonly router = inject(Router);

  protected saveAndMoveBackToHome(): void {
    // ... 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 parameters.

```
export const routes: Routes = [
  { path: '', component: Home },
  { path: 'races', component: Races },
  { path: 'races/:raceId/ponies/:ponyId', component: Pony }
];
```

We can then define dynamic links with `routerLink`:

```
<a [routerLink]="['/races', raceModel().id, 'ponies', ponyModel().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 has a handy property: `snapshot`. This property has all the parameters of the URL in `paramMap`!

```
export class Pony {
  protected readonly ponyModel: Signal<PonyModel | undefined>;

  constructor() {
    const route = inject(ActivatedRoute);
    const id = route.snapshot.paramMap.get('ponyId')!;
    const ponyService = inject(PonyService);
    this.ponyModel = toSignal(ponyService.get(id));
  }
}
```

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`.

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 neither the constructor, nor `ngOnInit` will 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 PonyReusable {
  protected readonly ponyModel: Signal<PonyModel | undefined>;

  constructor() {
    const route = inject(ActivatedRoute);
    const ponyService = inject(PonyService);
    this.ponyModel = toSignal(
      route.paramMap.pipe(
        map((params: ParamMap) => params.get('ponyId')!),
        switchMap(id => ponyService.get(id))
      )
    );
  }
}
```

Here we subscribe to the observable offered by `ActivatedRoute`. Now, every time the URL changes 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.

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

What you have just learnt should cover your basic routing needs. But the router goes well beyond this and offers many additional features. Covering them all in detail 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.

## 19.3. 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' },
```

## 19.4. 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: Race },
{ path: 'races/new', component: RaceCreation }
```

and the URL is `races/new`, the component that the router will activate is in fact the `Race`. 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: RaceCreation },
{ path: 'races/:id', component: Race }
```

## 19.5. 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 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.

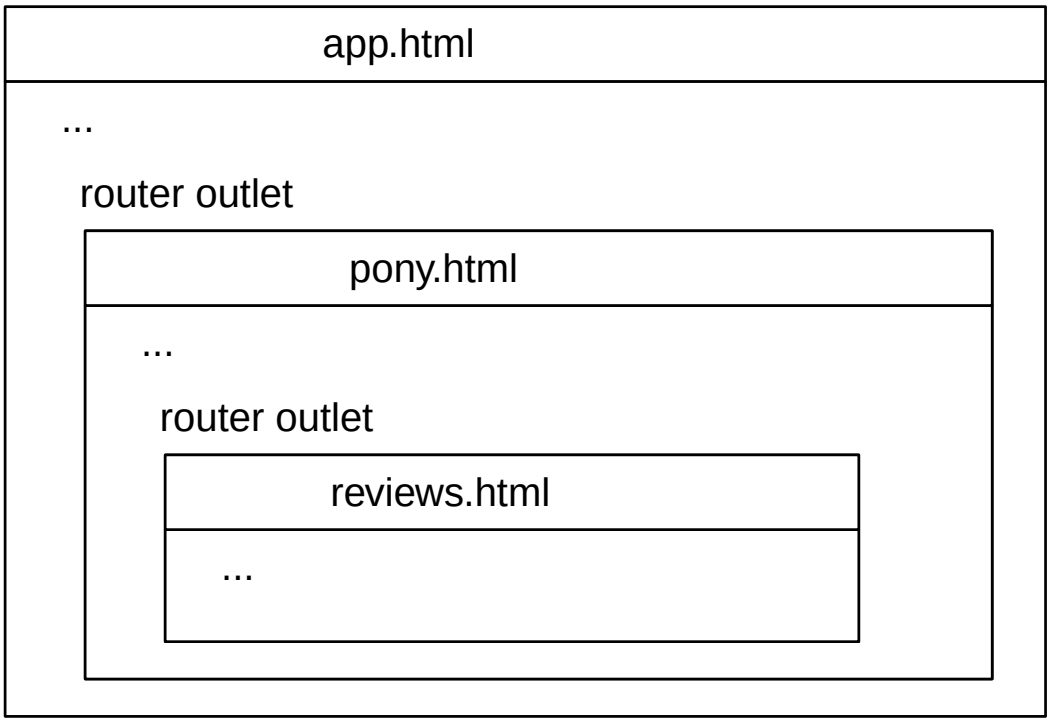
This mechanism can in fact be used in nested components, too. Suppose you have a complex page to display the profile of a pony. This page would display its name and portrait at the top, and would have several tabs at the bottom: one to display its birth certificate, one to display its track record, and 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 template of the `Pony`, and to define a parent pony route, this way:

```
{
  path: 'ponies/:ponyId',
  component: Pony,
  children: [
    { path: 'birth-certificate', component: BirthCertificate },
    { path: 'track-record', component: TrackRecord },
    { path: 'reviews', component: Reviews }
  ]
}
```

When going to the URL `ponies/42/reviews`, for example, the router will insert the Pony at the

location indicated by the main router-outlet, in the root component. The template of Pony, besides the name and the portrait of the pony, contains a second router-outlet. This is where the child `Reviews` will be inserted.




When going to the URL `ponies/42`, the pony component will be displayed, but none of the three children components 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: Pony,
  children: [
    { path: '', pathMatch: 'full', redirectTo: 'birth-certificate' },
    { path: 'birth-certificate', component: BirthCertificate },
    { path: 'track-record', component: TrackRecord },
    { path: 'reviews', component: Reviews }
  ]
}
```

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: Pony,
  children: [
    { path: '', component: BirthCertificate },
    { path: 'track-record', component: TrackRecord },
    { path: 'reviews', component: Reviews }
  ]
}
```

## 19.6. 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 access or modify. But that still won't prevent users from accessing routes that they're not allowed to access, who can simply enter 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 return a URL to navigate elsewhere (to be more accurate, it can return an Angular type called `UrlTree` - keep reading for an example). This can be useful for showing 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;
- `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 for asking for confirmation before leaving a route containing a large form, for example.
- `canMatch`: introduced in Angular v14.1, this guard indicates to the router if the route can be matched or not. If the guards returns `false`, then the route is simply ignored, and the router looks for another to match. A typical usage of `CanMatch` is to have several routes matching the same path with different components, and use `CanMatch` to let the router know which one should be displayed based on the user profile 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: Races, canActivate: [loggedInGuard] }
```

In the above example, `loggedInGuard` is a function.

In earlier versions of Angular (before v15.2), guards were defined as services, but

> this is now deprecated.

The function consists of 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|UrlTree>`, an `Observable<boolean|UrlTree>` or a `UrlTree`.

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`. If the returned value is a `UrlTree`, it cancels the current navigation and triggers a navigation to the returned `UrlTree`.

Here's what the `loggedInGuard` function might look like:

```
export const loggedInGuard: CanActivateFn = (
  _route: ActivatedRouteSnapshot,
  _state: RouterStateSnapshot
): boolean | UrlTree => {
  const userService = inject(UserService);
  const router = inject(Router);
  // returns `true` if the user is logged in or redirects to the login page
  // note that you can also use `router.createUrlTree()` to build a `UrlTree` with
parameters
  return userService.isLoggedIn() || router.parseUrl('/login');
};
```

Hierarchical routes combined with empty-path routes can be very handy when you want 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: Pony, canActivate: [loggedInGuard] },
{ path: 'races', component: Races, 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: Pony },
    { path: 'races', component: Races }
  ]
}
```

## 19.7. 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 `Race`, 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 a 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:

```
export const raceResolver: ResolveFn<RaceModel> = (route: ActivatedRouteSnapshot):
Observable<RaceModel> => {
  const id = route.paramMap.get('raceId')!;
  const raceService = inject(RaceService);
  return raceService.get(id);
};
```

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

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

```
{
  path: 'races/:raceId',
  component: Race,
  resolve: {
    race: raceResolver
  }
}
```

As you can see, the resolver 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 Race {
  protected readonly race = signal(inject(ActivatedRoute).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 Race {
  private readonly route = inject(ActivatedRoute);
  protected readonly raceModel = toSignal(this.route.data.pipe(map(data => 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.

## 19.8. 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`). Version 5.0 added more fine-grained navigation events (`ChildActivationStart`, `ChildActivationEnd`). Version 6.1 added a `Scroll` event, along with the `scrollPositionRestoration` configuration option that allows you to restore the scroll position when navigating back to a component.

## 19.9. 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*.

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, which 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 with 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: Races,
  data: {
    allowDeletion: false
  }
```

```
}
```

## 19.10. Bind parameters and data to component inputs

We saw that the parameters and data of the route are accessible from the component associated with the route, via observables on the `ActivatedRoute` object.

Since Angular v16, it's possible to automatically bind the route parameters and data to the component inputs.

To do so, you need to configure the router with `withComponentInputBinding`:

*app.config.ts*

```
provideRouter(routes, withComponentInputBinding())
```

With this option, a component can declare an input with the same name as a route parameter or data, and Angular will automatically bind the value of the parameter or data to this input.

*race.ts*

```
export class Race {
  readonly raceId = input.required<string>();
```

We can then use this input as a regular input, and react to its change with `toObservable` and `toSignal`:

*race.ts*

```
private readonly raceService = inject(RaceService);
private readonly raceModel$ = toObservable(this.raceId).pipe(switchMap(id => this
.raceService.get(id)));
protected readonly raceModel = toSignal(this.raceModel$);
```

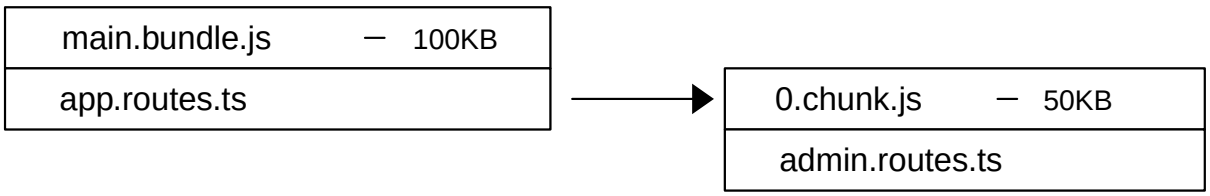
## 19.11. Lazy loading

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 several JavaScript bundles, by loading child routes lazily. The application routes array, instead of defining all the routes of the application, only contains the main, eagerly-loaded routes, as well as parent routes with lazy-loaded children, which are thus unknown initially.

When the user navigates to the path of an unknown child route, then the Angular router loads the JavaScript bundle containing the child routes (and their associated components and services), and adds the child routes, components and services to the application.




it's actually possible to load the lazy-loading bundles in the background, after the main bundle has been loaded and the application has started, without waiting for the user to navigate to the child routes, 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.

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

```
export const adminRoutes: Routes = [{ path: '', component: Admin }];
```

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

```
{ path: 'admin', loadChildren: () => import('./admin/admin.routes').then(m => m
```

```
.adminRoutes) }
```

As you can see, this is achieved by using the `loadChildren` property of the route definition and the dynamic import function from TypeScript.

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.routes'`.

You can even simplify the import if you use a `default` export for the module:

```
const defaultAdminRoutes: Routes = [{ path: '', component: Admin }];
export default defaultAdminRoutes;
```

Then the router will automatically "unwrap" the import:

```
{ path: 'admin', loadChildren: () => import('./admin/admin.routes') }
```

Finally, if (as above) all you want to do is to lazy-load a single component for a given route, then it's even easier: you don't even have to define an additional routes file. The component can be lazy-loaded directly:

```
{ path: 'admin', loadComponent: () => import('./admin/admin').then(m => m.Admin) }
```

Try our quiz 🦄 and the exercises Protected routes with guards 🦄, Nested routes and redirections 🦄 and Lazy loading 🦄 to learn how to use the advanced features of the router.

# Chapter 20. Forms

## 20.1. 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, it 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 to 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, so let's begin with this:

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

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

```
export class RegisterForm {}
```

Nothing fancy: a component with a simple template containing a form. In the next few minutes, we will build a form allowing you 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 the value of the control changes.
- `statusChanges`: an Observable emitting every time there the status of the control changes.
- `events`: an Observable emitting every time the state or value of the control changes. This has been introduced in Angular v18 and allows to handle value, status, pristine, touched changes or to know when the form has been reset or submitted.

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); // '' until the user enters a value
console.log(password.hasError('required')); // false
password.disable(); // disables the control
password.reset(); // resets the value
```

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: ''}
console.log(form.controls.username); // logs the Control
```

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

## 20.2. 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 each component using a template-driven form.

Unlike the directives from `CommonModule` and `RouterModule`, which are standalone, the directives of the `FormsModule` are not. So you can't import them one by one in your components. The whole `FormsModule` must be imported.

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

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

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

@Component({
  selector: 'ns-register',
  template: `
    <h2>Sign up</h2>
    <form (ngSubmit)="register()">
      <button type="submit">Register</button>
    </form>
  `,
  imports: [FormsModule]
})
export class RegisterForm {
  protected register(): void {
    // 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.

You might wonder why there is an `NgForm` directive available on the `form` element, even though it doesn't have any specific attribute. It's simply that the selector of the `NgForm` directive is `form` (it is

actually a bit more specific than that), which means that every standard HTML form element actually triggers the creation of an `NgForm` directive, as long as the `FormsModule` has been imported.

Last thing: our template will quickly grow, so let's extract it to a dedicated file, using `templateUrl`:

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

@Component({
  selector: 'ns-register',
  templateUrl: './register-form.html',
  imports: [FormsModule]
})
export class RegisterForm {
  protected register(): void {
    // 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, which 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 to 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 detail 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';
import { FormsModule } from '@angular/forms';
import { UserModel } from './user.model';

@Component({
  selector: 'ns-register',
  templateUrl: '../code/register-form.html',
  imports: [FormsModule]
})
export class RegisterForm {
  protected register(user: UserModel): void {
    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!

### 20.2.1. 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 `UserModel` interface:

```
export interface UserModel {
```

```
  username: string;
  password: string;
}
```

Our `RegisterForm` should have a field `user` of type `UserModel`:

```
@Component({
  selector: 'ns-register',
  templateUrl: './register-form.html',
  imports: [FormsModule]
})
export class RegisterForm {
  protected readonly user: UserModel = {
    username: '',
    password: ''
  };

  protected register(): void {
    console.log(this.user);
  }
}
```

We don't use signals in these examples, as every property of the `UserModel` would have to be signals, which is not super ergonomic (but doable). We can imagine the situation will improve when forms will be re-worked in Angular to better handle signal.

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 remembering 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!

## 20.3. Code-driven

In AngularJS 1.x you had to build your forms mostly in your templates. Angular introduces an imperative way, which allows you 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, inject } from '@angular/core';
```

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

@Component({
  selector: 'ns-register',
  templateUrl: './register-form.html',
  imports: [ReactiveFormsModule]
})
export class RegisterForm {
  private readonly fb = inject(FormBuilder);
  // we will have to build the form

  protected register(): void {
    // 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, inject } from '@angular/core';
import { FormBuilder, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: './register-form.html',
  imports: [ReactiveFormsModule]
})
export class RegisterForm {
  private readonly fb = inject(FormBuilder);
  protected readonly userForm = this.fb.group({
    username: '',
    password: ''
  });
  // `userForm` is of type `FormGroup<{
  //   username: FormControl<string | null>;
  //   password: FormControl<string | null>;
  // }>`

  protected register(): void {
    // 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, inject } from '@angular/core';
import { FormBuilder, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: './register-form.html',
  imports: [ReactiveFormsModule]
})
export class RegisterForm {
  private readonly fb = inject(FormBuilder);
  protected readonly userForm = this.fb.group({
    username: '',
    password: ''
  });

  protected register(): void {
    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 into your component. Their names begin with `form` instead of `ng` as 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, inject } from '@angular/core';
import { FormBuilder, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: './register-form.html',
  imports: [ReactiveFormsModule]
})
export class RegisterForm {
  private readonly fb = inject(FormBuilder);
  protected readonly usernameCtrl = this.fb.control('');
  protected readonly passwordCtrl = this.fb.control('');
  protected readonly userForm = this.fb.group({
    username: this.usernameCtrl,
    password: this.passwordCtrl
  });

  protected setAnotherNinja(): void {
    this.usernameCtrl.setValue('JB');
  }

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

# 20.4. Adding some validation

Validation is usually a big part of 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.

## 20.4.1. 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 finding the correct regular expression by yourself for this one…)
- `Validators.min(n)` (available since version 4.2) to ensure that the value entered is at least *n*
- `Validators.max(n)` (available since version 4.2) to ensure that the value entered is at most *n*
- `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, inject } from '@angular/core';
import { FormBuilder, ReactiveFormsModule, Validators } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: './register-form.html',
  imports: [ReactiveFormsModule]
})
export class RegisterForm {
  private readonly fb = inject(FormBuilder);
  protected readonly userForm = this.fb.group({
    username: this.fb.control('', [Validators.required, Validators.minLength(3)]),
    password: this.fb.control('', Validators.required)
  });

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

### 20.4.2. 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` (`min` and `max` are not yet available as directives).

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.

## 20.5. 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.

### 20.5.1. 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">
    @if (userForm.controls.username.hasError('required')) {
      <div>Username is required</div>
```

```
    }
    @if (userForm.controls.username.hasError('minlength')) {
      <div>Username should be 3 characters min</div>
    }
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
    @if (userForm.controls.password.hasError('required')) {
      <div>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">
    @if (userForm.controls.username.dirty &&
userForm.controls.username.hasError('required')) {
      <div>Username is required</div>
    }
    @if (userForm.controls.username.dirty &&
userForm.controls.username.hasError('minlength')) {
      <div>Username should be 3 characters min</div>
    }
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
    @if (userForm.controls.password.dirty &&
userForm.controls.password.hasError('required')) {
      <div>Password is required</div>
    }
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>
```

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

```
@Component({
  selector: 'ns-register',
  templateUrl: './register-form.html',
  imports: [ReactiveFormsModule]
})
```

```
export class RegisterForm {
  private readonly fb = inject(FormBuilder);
  protected readonly usernameCtrl = this.fb.control('', Validators.required);
  protected readonly passwordCtrl = this.fb.control('', Validators.required);
  protected readonly userForm = this.fb.group({
    username: this.usernameCtrl,
    password: this.passwordCtrl
  });

  protected register(): void {
    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">
    @if (usernameCtrl.dirty && usernameCtrl.hasError('required')) {
      <div>Username is required</div>
    }
    @if (usernameCtrl.dirty && usernameCtrl.hasError('minlength')) {
      <div>Username should be 3 characters min</div>
    }
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
    @if (passwordCtrl.dirty && passwordCtrl.hasError('required')) {
      <div>Password is required</div>
    }
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>
```

### 20.5.2. 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 you to know 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
">
    @if (username.dirty && username.hasError('required')) {
      <div>Username is required</div>
    }
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel required
#password="ngModel">
    @if (password.dirty && password.hasError('required')) {
      <div>Password is required</div>
    }
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>
```

Yay!

## 20.6. 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 change 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`.

## 20.7. Creating a custom validator

Pony races are an addictive game so you are 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: AbstractControl<Date | null>) => {
  // 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.

### 20.7.1. 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, inject } from '@angular/core';
import { AbstractControl, FormBuilder, ReactiveFormsModule, ValidationErrors,
Validators } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: './register-form.html',
  imports: [ReactiveFormsModule]
})
export class RegisterForm {
  private readonly fb = inject(FormBuilder);
  protected readonly usernameCtrl = this.fb.control('', Validators.required);
  protected readonly passwordCtrl = this.fb.control('', Validators.required);
  protected readonly birthdateCtrl = this.fb.control('', [Validators.required,
RegisterForm.isOldEnough]);
  protected readonly userForm = this.fb.group({
    username: this.usernameCtrl,
    password: this.passwordCtrl,
    birthdate: this.birthdateCtrl
  });

  private static isOldEnough(control: AbstractControl): ValidationErrors | null {
    // 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 };
  }

  protected register(): void {
    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">
    @if (usernameCtrl.dirty && usernameCtrl.hasError('required')) {
      <div>Username is required</div>
    }
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
    @if (passwordCtrl.dirty && passwordCtrl.hasError('required')) {
      <div>Password is required</div>
    }
  </div>
  <div>
    <label>Birth date</label><input type="date" formControlName="birthdate">
    @if (birthdateCtrl.dirty) {
      @if (birthdateCtrl.hasError('required')) {
        <div>Birth date is required</div>
      } @else if (birthdateCtrl.hasError('tooYoung')) {
        <div>You're way too young to be betting on pony races</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.html',
  imports: [ReactiveFormsModule]
})
export class RegisterForm {
  private readonly fb = inject(FormBuilder);
  protected readonly usernameCtrl = this.fb.control('', Validators.required, control
=>
    this.isUsernameAvailable(control)
  );
```

```
  protected readonly userForm = this.fb.group({
    username: this.usernameCtrl
  });

  private readonly userService = inject(UserService);

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

  protected register(): void {
    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).

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

### 20.7.2. 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 when using a "code-driven" form…

## 20.8. 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.

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

```
import { Component, inject } from '@angular/core';
import { AbstractControl, FormBuilder, ReactiveFormsModule, ValidationErrors,
```

```
Validators } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: './register-form.html',
  imports: [ReactiveFormsModule]
})
export class RegisterForm {
  private readonly fb = inject(FormBuilder);
  protected readonly usernameCtrl = this.fb.control('', Validators.required);
  protected readonly passwordCtrl = this.fb.control('', Validators.required);
  protected readonly confirmCtrl = this.fb.control('', Validators.required);
  protected readonly passwordGroup = this.fb.group(
    { password: this.passwordCtrl, confirm: this.confirmCtrl },
    { validators: RegisterForm.passwordMatch }
  );

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

  private static passwordMatch(group: AbstractControl): ValidationErrors | null {
    const password = group.value.password;
    const confirm = group.value.confirm;
    return password === confirm ? null : { matchingError: true };
  }

  protected register(): void {
    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">
    @if (usernameCtrl.dirty && usernameCtrl.hasError('required')) {
      <div>Username is required</div>
    }
  </div>
  <div formGroupName="passwordForm">
    <div>
      <label>Password</label><input type="password" formControlName="password">
      @if (passwordCtrl.dirty && passwordCtrl.hasError('required')) {
        <div>Password is required</div>
      }
```

```
    </div>
    <div>
      <label>Confirm password</label><input type="password" formControlName="confirm">
      @if (confirmCtrl.dirty && confirmCtrl.hasError('required')) {
        <div>Confirm your password</div>
      }
    </div>
    @if (passwordGroup.dirty && passwordGroup.hasError('matchingError')) {
      <div>Your password does not match</div>
    }
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>
```

Voilà!

## 20.9. Reacting to changes

A last cool feature when using a code-driven form: you can easily react to 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:

```
private readonly fb = inject(FormBuilder);
protected readonly usernameCtrl = this.fb.control('', Validators.required);
protected readonly passwordCtrl = this.fb.control('', Validators.required);
protected readonly userForm = this.fb.group({
  username: this.usernameCtrl,
  password: this.passwordCtrl
});
protected readonly passwordStrength = signal(0);

constructor() {
  // we subscribe to every password change
  this.passwordCtrl.valueChanges
    .pipe(
      // only recompute when the user stops typing for 400ms
      debounceTime(400),
      // only recompute if the new value is different from the last
      distinctUntilChanged()
    )
    .subscribe(newValue => this.passwordStrength.set(this.computePasswordStrength
(newValue)));
}
```

or with `toSignal`:

```
protected readonly passwordStrength = toSignal(
```

```
  this.passwordCtrl.valueChanges.pipe(
    // only recompute when the user stops typing for 400ms
    debounceTime(400),
    // only recompute if the new value is different from the last
    distinctUntilChanged(),
    // compute the length of the password
    map(newValue => this.computePasswordStrength(newValue))
  ),
  { initialValue: 0 }
);
```

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">
    @if (usernameCtrl.dirty && usernameCtrl.hasError('required')) {
      <div>Username is required</div>
    }
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
    <div>Strength: {{ passwordStrength() }}</div>
    @if (passwordCtrl.dirty && passwordCtrl.hasError('required')) {
      <div>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 computing 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 `HttpClient` service uses observables too.

## 20.10. Updating on blur or on submit only

Angular 5.0 introduced the possibility of waiting for the `blur` or the `submit` event to update the field's value and validity. To do so, the `FormControl` constructor accepts an `options` object as the second parameter, to define the synchronous and asynchronous validators, and also the `updateOn` option. Its value can be:

- `change`, it's the default: the value and validity are updated on every change;
- `blur`, the value and validity are then updated only when the field loses the focus.
- `submit`, the value and validity are then updated only when the parent form is submitted.

```
protected readonly usernameCtrl = this.fb.control('', Validators.required);
protected readonly passwordCtrl = this.fb.control('', {
  validators: Validators.required,
  updateOn: 'blur'
});
```

It's also possible to configure this option on a group of fields all at once:

```
protected readonly userForm = this.fb.group(
  {
    username: this.usernameCtrl,
    password: this.passwordCtrl
  },
  {
    updateOn: 'blur'
  }
);
```

The same feature is available in template-driven forms, with the `ngModelOptions` input of the `NgModel` directive:

```
<label>Username</label>
<input name="username" #usernameCtrl="ngModel"
  [(ngModel)]="user.username" [ngModelOptions]="{ updateOn: 'blur' }" required>
@if (usernameCtrl.dirty && usernameCtrl.hasError('required')) {
  <div>Username is required</div>
}
```

or globally on a form with the `NgFormOptions` input (which appeared with Angular 5.0) of the directive `NgForm`:

```
<form (ngSubmit)="register()" [ngFormOptions]="{ updateOn: 'blur' }">
  <div>
    <label>Username</label>
```

```
    <input name="username" #usernameCtrl="ngModel"
      [(ngModel)]="user.username" required>
    @if (usernameCtrl.dirty && usernameCtrl.hasError('required')) {
      <div>Username is required</div>
    }
```

## 20.11. FormArray and FormRecord

FormControl and FormGroup are not the only entities that we can use to build a form. If you want a part of the form to be dynamic, you can use a FormArray or a FormRecord.

FormArray is an array of controls that contains the same types of value. A typical example is a form where you want the users to be able to add/remove values, like for example tags when editing a blog post:

```
export class EditBlogPost {
  private readonly fb = inject(FormBuilder);
  // one empty tag by default
  protected readonly tagsArray = this.fb.array(['']);
  protected readonly blogPostForm = this.fb.group({
    title: '',
    content: '',
    tags: this.tagsArray
  });

  protected addTag() {
    this.tagsArray.push(this.fb.control(''));
  }

  protected removeTag(index: number) {
    this.tagsArray.removeAt(index);
  }
```

Then you can iterate on the controls in the template, and have buttons to add/remove a tag:

```
<div formArrayName="tags">
  <p>Tags</p>
  <button id="add-tag" (click)="addTag()">Add tag</button>
  @for (tagControl of tagsArray.controls; track tagControl) {
    <div>
      <input class="tag" [formControlName]="$index" />
      <button class="remove-tag" (click)="removeTag($index)">Remove tag</button>
    </div>
  }
</div>
```

The value of the FormArray is an array of strings in this example. Of course, form arrays can contain

any kind of controls. For example, if you want to add lines to an invoice, the `FormArray` will contain form groups with a label, quantity, price, etc.

`FormRecord` is another entity you can use to build forms. It has been introduced in Angular v14, and allows building key-value maps.

Let's say you want to build the packing list for a trip, with the possibility to add and remove items, and to check/uncheck them. `FormRecord` can help in that case:

```
export class EditPackingList {
  private readonly fb = inject(NonNullableFormBuilder);
  protected readonly equipmentRecord = this.fb.record({
    // always bring your toothbrush
    toothbrush: true
  });
  protected readonly packingListForm = this.fb.group({
    equipments: this.equipmentRecord
  });

  protected addEquipment(equipment: string) {
    this.equipmentRecord.addControl(equipment, this.fb.control(true));
  }

  protected removeEquipment(equipment: string) {
    this.equipmentRecord.removeControl(equipment);
  }
```

Then you can iterate on the controls in the template, and have buttons to add/remove equipment:

```
<div formGroupName="equipments">
  <label for="equipment-to-add">Equipments</label>
  <input #equipment id="equipment-to-add" />
  <button id="add-equipment" (click)="addEquipment(equipment.value)">Add</button>
  @for (control of equipmentRecord.controls | keyvalue; track control) {
    <div>
      <label [for]="'eq-' + control.key">{{ control.key }}</label>
      <input [id]="'eq-' + control.key" type="checkbox" class="equipment"
[formControlName]="control.key" />
      <button class="remove-equipment" (click)="removeEquipment(control.key)">Remove
equipment</button>
    </div>
  }
</div>
```

The value of the `FormRecord` will then be an object, with the equipment name as the key and a boolean as value. For example:

```
{
```

```
  toothbrush: false,
  jacket: true
}
```

## 20.12. Strictly typed forms

Until version 14 of Angular, forms were not typed. What does that mean?

Well, the value of a `FormControl` or a `FormGroup` was of type `any`, which is obviously far from being ideal.

If you upgrade an application from version 13 to version 14, in order not to break your code, the types `FormControl`, `FormGroup` and `FormArray` are replaced with `UntypedFormControl`, `UntypedFormGroup` and `UntypedFormArray`.

The original types are now typed. But what are the concrete changes in Angular 14?

The form elements became generic. So, we now use `FormControl<string>` for example, to indicate that the value of the control is a string. The FormGroups, on the other hand, take the type of the controls that they contain. For example, a `FormGroup` used to register a user will have the type

```
FormGroup<{
  username: FormControl<string>;
  password: FormControl<string>;
}>;
```

It's verbose, but don't panic: these generic types are, most of the time, inferred by the compiler when the variables are initialized.

### 20.12.1. Nullability

I have simplified things a little bit in the previous section. Indeed, the form elements typing must consider two harsh realities:

1. form elements can be disabled
2. the behavior of the `reset()` method is to set the form control values to `null`.

When a control is disabled, its value doesn't appear in the value of its parent `FormGroup`. So, if I decide to disable the *password* control of my form, the `FormGroup` value will simply be

```
{
  username: 'cedric'
}
```

And that is slightly annoying, because it implies that the value of the `FormGroup` is not of type

```
{
  username: string;
  password: string;
}
```

as one might expect, but it is in fact of type

```
{
  username?: string;
  password?: string;
}
```

Indeed, Angular can't possibly know whether you're planning to disable the form controls or not. It's thus forced to make every of the properties of the `FormGroup` value optional. It's your responsibility, as a developer, to handle that. It's typically done by using `value.username!` in order to get a value of type `string` rather than a value of type `string | undefined`, if you know that this control is not disabled. Another possibility is to use the *raw* value of the form group (returned by `getRawValue()`). This raw value contains all the properties, whether or not their matching control is disabled. And these properties are thus not marked as optional.

The second point causes the same kind of problem. Since Angular can't know if you're planning to call the `reset()` method or not, the value of the `FormGroup` is actually of type

```
{
  username?: string | null;
  password?: string | null;
}
```

This second annoyance, unlike the first one, can be avoided. To avoid it, every element of the form must be configured with the option `nonNullable: true`. This option changes the behavior of `reset()`, which then resets the control to its initial value rather than `null`. Configuring this option on every form element is quite verbose and tedious. It's however the only way if you use the form elements constructors to create them. It's much easier if you choose to use the `FormBuilder`. All you need to do is to inject, instead of `FormBuilder`, a `NonNullableFormBuilder`. It has the same API, but configures all the elements it created with the `nonNullable` option.

```
export class RegisterForm {
  protected readonly userForm = inject(NonNullableFormBuilder).group({
    username: '',
    password: ''
  });
}
```

Note that the name `NonNullableFormBuilder` is misleading. Using it doesn't prevent controls to have a nullable value. Consider, for instance, an input of type number. There is no real good default value

other than `null` in this case. And even if you initialize the control with a non-null value, The user can always clear the input value, thus setting the control value to `null`. In this case, you have to explicitly type the control:

```
readonly birthYearCtrl = new FormControl<number | null>(null);
```

or, with the `FormBuilder` or the `NonNullableFormBuilder`:

```
protected readonly birthYearCtrl = inject(NonNullableFormBuilder).control<number |
null>(null);
```

or if the control is part of a `FormGroup`:

```
protected readonly formGroup = inject(NonNullableFormBuilder).group({
  // ...
  birthYear: null as number | null
});
```

All this additional complexity might look off-putting, but one should not overlook what we gained: objects with properties of known names and types, which can be auto-completed by the IDE. In the end, more maintainable and more robust code.

## 20.13. Super simple validation error messages with `ngx-valdemort`

As you may have noticed, the templates can quickly become very verbose with all the error messages that we have to repeat for each error type on each field, in every form. You are quickly stuck with very long `ngIf` and copied/pasted boilerplate between components.

As we find it distasteful as well, we wrote a tiny open-source library to make it easier (heavily inspired by `ngMessages` in AngularJS): ngx-valdemort.

Instead of

```
<input id="email" formControlName="email" class="form-control" type="email" />
@if (form.controls.email.invalid && (f.submitted || form.controls.email.touched)) {
  <div class="invalid-feedback">
    @if (form.controls.email.hasError('required')) {
      <div>The email is required</div>
    }
    @if (form.controls.email.hasError('email')) {
      <div>The email must be a valid email address</div>
    }
  </div>
}
```

the library allows you to write:

```
<input id="email" formControlName="email" class="form-control" type="email" />
<val-errors controlName="email">
  <ng-template valError="required">The email is required</ng-template>
  <ng-template valError="email">The email must be a valid email address</ng-template>
</val-errors>
```

We can even do better by defining default messages once and for all:

```
<val-default-errors>
  <ng-template valError="required" let-label> {{ label || 'This field' }} is required
</ng-template>
  <ng-template valError="email" let-label> {{ label || 'This field' }} must be a valid
email address </ng-template>
  <ng-template valError="min" let-error="error" let-label>
    {{ label || 'This field' }} must be at least {{ error.min | number }}
  </ng-template>
  <!-- same for the other types of error -->
</val-default-errors>
```

And then simply use:

```
<input id="email" formControlName="email" class="form-control" type="email" />
<val-errors controlName="email" label="The email" />
```

We provide an integration with Bootstrap and Material, to have error messages with a coherent style if you use one of these CSS frameworks. Give it a try, you won't regret it!

## 20.14. Going further: define custom form inputs with `ControlValueAccessor`

HTML defines a large set of input types: text, password, checkbox, etc. But sometimes these standard types don't fit the bill.

Angular allows defining custom components, and it's actually possible to make them act as Angular form controls, i.e. bind them to a form control by applying the `NgModel` directive or the `FormControlName` directive, and thus integrating them in a form.

The glue to implement this binding is an interface provided by Angular: `ControlValueAccessor`.

Fulfilling its contract is relatively straightforward. You have to

- accept the value of the FormControl and display it in your component (`writeValue`);
- notify Angular that the user changed the value somehow, by calling a provided callback function (`registerOnChange`);

- notify Angular when the control should be considered as touched, by calling a provided callback function (`registerOnTouched`);
- honor the request from Angular to disable or enable the control (`setDisabledState`).

We will illustrate all of this using a custom *rating* component. This component allows rating a movie, for example, by giving it a note between 0 and 5. But instead of using a `number` or `range` input, we would like the user to do that by simply clicking one of 6 buttons (that would typically be displayed as star icons, but we'll leave that out in the following example).

Here's the code of such a component.

```
export class Rating implements ControlValueAccessor {
  private onChange: (rating: number) => void = () => {
    // do nothing by default
  };

  onTouched: () => void = () => {
    // do nothing by default
  };

  protected readonly value = signal<number | null>(null);
  protected readonly disabled = signal(false);
  protected readonly pickableValues = [0, 1, 2, 3, 4, 5];

  registerOnChange(fn: (rating: number) => void): void {
    this.onChange = fn;
  }

  registerOnTouched(fn: () => void): void {
    this.onTouched = fn;
  }

  setDisabledState(isDisabled: boolean): void {
    this.disabled.set(isDisabled);
  }

  writeValue(v: number | null): void {
    this.value.set(v);
  }

  protected setValueAndPropagateChanges(value: number) {
    this.value.set(value);
    // tell Angular the value has changed
    this.onChange(value);
  }
}
```

and here is its template:

```
@let v = value();
@for (pickableValue of pickableValues; track pickableValue) {
  <button
    [class.selected]="v != null && pickableValue <= v"
    type="button"
    (click)="setValueAndPropagateChanges(pickableValue)"
    [disabled]="disabled()"
    (blur)="onTouched()"
  >
    {{ pickableValue }}
  </button>
}
```

The code to deal with the two callback functions and the disabled state is boilerplate code that is almost identical in every *CVA* (`ControlValueAccessor`).

The interesting part is the handling of the value. Angular calls `writeValue()` to tell the component what value to display. Here, we simply store the value, and use it in the template to display the first buttons as yellow-filled buttons.

When a button is clicked, we change the value of the component of course, but we must also tell Angular about this change. That's what allows it to change the value of the form control, trigger the validations, etc.

The disabled state is honored by disabling all the buttons.

And finally, we choose to make the form control *touched* as soon as one of the rating buttons loses the focus. We do that by calling the provided `onTouched` callback function when a button fires a `blur` event.

The last thing we have to do is to register this component as one of the control value accessors of the application. We do that by adding a provider to the rating component's decorator. Don't bother too much with the syntax: you can simply copy and paste this snippet every time you define a new CVA.

```
providers: [
  {
    provide: NG_VALUE_ACCESSOR,
    useExisting: forwardRef(() => Rating),
    multi: true
  }
]
```

Voilà. We now have a shiny reusable rating component that can be used to rate something in any form, by simply applying one of the standard Angular form directives to the component:

```
<ns-rating id="rating" formControlName="rating" />
```

## 20.15. 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 to changes on a group, or on a field.

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.

Try our quiz 🦄, and the exercises Login form 🦄, Register form 🦄 and Control Value Accessor 🦄. You'll learn how to build forms using `ReactiveFormsModule`, how to build custom form components, how to write custom validators, how to test forms, and how to authenticate your users!

# Chapter 21. Zones and the Angular magic

Hopefully, you remember what we explained in the chapter introducing signals. Angular wants you to store the state of your components in signals. Templates read the signals to display the information they contain. Thanks to the signal super-powers, it can establish a dependency between a template and the signals that it reads, be notified when signals change, and update the DOM of the templates that depend on these signals.

If you experiment with the current version of Angular (or the older ones), you'll see that you actually don't need to store the state in signals. Even if a simple property changes, Angular is able to detect the change and update the DOM. You shouldn't do that though: this works because Angular still supports the "brute-force" change detection mechanism that doesn't rely on signals.

In order to better understand how "legacy" applications work, and why it's a better idea to switch to signals, let's explain this change detection magic.

## 21.1. ZoneJS

The "legacy" change detection mechanism is based on a dirty trick: ZoneJS. ZoneJS is a library that has been written for the needs of Angular, but is actually not coupled to it. Its principle is pretty simple, but ugly: it monkey-patches plenty of functions of the browser. Basically, all the functions of the browser that receive a callback function, like `setTimeout`, `setInterval`, `Element.addEventListener`, `Promise.then`, etc.

The goal of this monkey-patching, for Angular, is to act as a snitch. Every time one of these callback functions is executed, ZoneJS notifies Angular that something has happened.

To illustrate it, let's take `setTimeout` as an example. The principle of the monkey-patching is the following one (this is not the actual code):

```
const originalSetTimeout = window.setTimeout;
window.setTimeout = (callback, timeout) => {
  const snitchingCallback = () => {
    callback();
    angular.notifyThatSomeCodeHasBeenExecuted();
  };
  originalSetTimeout(snitchingCallback, timeout);
};
```

Why does this help? Thanks to ZoneJS, the framework *knows* when some code has been executed. It thus knows that some state *might* have changed. But it still doesn't know *if* something actually changed, and even less *what* has changed.

To have a DOM that is always up-to-date, Angular will thus launch a change detection every time ZoneJS tells it that some code has been executed.

This first part already shows reasons why Angular tries to get rid of ZoneJS by introducing signals:

- every Angular application needs to download the ZoneJS code and wait until it has patched the browser before starting the application, which of course has a performance cost;
- every call to one of the patched functions triggers change detections. Even the calls made by libraries which don't change any application state. So plenty of change detections are actually useless;
- the stack traces of the errors that happen at runtime contain zone-related code that makes them harder to read and understand.

## 21.2. Change detection

The second part of the problem is the 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 the change detection runs, it goes through *all* the components in the tree, from the root to the leaves. It evaluates *all* the expressions used in the templates, and compares each result with the previous one. If an evaluation result is not the same, then the DOM is updated. The expression that has a new value could also be used as the input of a child component. In that case, Angular updates the input of the child component and calls its `ngOnChanges` hook, which might change the state of the child component. Then the child component is checked.

The tree traversal, fortunately for performance, is done only once per change detection. But this comes with another disadvantage: you can introduce a bug if, during the changed detection of a child component, you update its own state or the state of a parent. Since Angular does a single pass, this change will go unnoticed until the next time ZoneJS triggers a new change detection.

Angular forbids you from doing that. During development, each change detection actually does two traversals. If the second traversal detects a change, then Angular throws an error to warn you that you broke the unidirectional data flow principle (the infamous `ExpressionChangedAfterItHasBeenCheckedError`).

As you can see, change detection is quite brute-force. The expressions in your templates are evaluated many many times. This is why you should make sure they don't invoke complex, slow functions.

### 21.2.1. Inline caching

Another parameter to take into account in the performance of Angular is the time spent by the framework making these expression evaluations. The Angular team has employed all the knowledge it has about computer science and virtual machines.

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 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.

In order to benefit from the inline caching optimizations of the VM, Angular has adopted a clever strategy. Instead of using a single function able to evaluate expressions involving all kinds of objects, the Angular team has decided to compile all the templates into JavaScript code, and to make the compiler generates a set of change detection functions, specific to each component.

The VM can optimize this code because it's monomorphic.

### 21.2.2. Tweaking the change detection

If really needed, it's possible to go further than those automatic optimizations provided by the framework: components can be configured with an alternative *ChangeDetection* strategy. But that's for another chapter.

The future goes beyond that. Thanks to signals, ZoneJS and this brute-force change detection mechanism can be completely eliminated. It's actually already doable if you accept to use experimental features.

If you want to tweak the change detection or learn how to go zoneless, read the performances chapter.

# Chapter 22. Angular compilation: Just in Time vs Ahead of Time

## 22.1. Code generation

In the previous chapter, we talked briefly about how the framework *generates* 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 component definition for each component that represents exactly what you wrote in your template. This component definition is inlined in a static field of your component.

Let's take an example, with our well-known `Pony`. The template is mainly an image with a bound source property, and a `figcaption` element with an interpolation.

```
<figure>
  <img [src]="ponyImageUrl()">
  <figcaption>{{ ponyModel().name }}</figcaption>
</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 "component definition" per component, inlined in a static field of our component class. A component definition contains several things, and among them the template, represented by a function.

```
elementStart(0, 'figure');
{
  element(1, 'img');
}
{
  elementStart(2, 'figcaption');
  text(3);
  elementEnd();
}
elementEnd();
```

This chapter describes the code generated by the compiler/renderer introduced in Angular 8.0 and called "Ivy". We wrote a detailed blog post about Ivy if you want to learn more. This renderer is the third iteration, as the initial renderer has already

> been rewritten in Angular 4.0. These iterations have brought either better performance or bundle size improvements or both, while keeping the same template syntax, allowing a backward compatibility with existing code, and allowing us developers to migrate very easily. Most people haven't noticed that the renderer changed in Angular 4.0 for example.

With this function, Angular is able to create the DOM corresponding to our `Pony`: it basically appends the corresponding `HTMLElement` to the DOM, for every element.

But how does it handle the change detection? The template function is in fact a little bit longer:

```
template: (renderFlags: RenderFlags, component: Pony) => {
  if (renderFlags & RenderFlags.Create) {
    elementStart(0, 'figure');
    {
      element(1, 'img');
    }
    {
      elementStart(2, 'figcaption');
      text(3);
      elementEnd();
    }
    elementEnd();
  }
  if (renderFlags & RenderFlags.Update) {
    advance();
    property('src', component.ponyImageUrl());
    advance(2);
    textInterpolate(component.ponyModel().name);
  }
},
```

This template function has two parts:

- the creation of the component that we explained above
- the update of the component which does basically what you would write by hand (i.e. the image has a source property that reflects the result of the method `ponyImageUrl()` of my component, and the `figcaption` has a text that reflects the pony's name).

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 index of the element to update, and another one with its property and the new value. The framework then 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). This code generation takes some time, so it used to be done in the browser, on the application start. This mode was called the *Just in Time* (JiT) compilation. The Ivy compiler is now fast enough to do this compilation directly when you run

`ng serve` or `ng build`. This mode is called *Ahead of Time* (AoT), and is detailed below.

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.

## 22.2. Ahead of Time compilation

We can generate this code *before* starting the application, using the 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 enabled by default when you run `ng serve` or `ng build` since Angular CLI v8.1.

The build uses the Angular compiler to compile the templates to TypeScript files. To TypeScript? Yes, because it then makes sure 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 immediately), and if you made a mistake in a template, you'll see an error:

```
<figure>
  <!-- wrong computed name -->
  <img [src]="ponyImageUr()">
  <figcaption>{{ ponyModel.name }}</figcaption>
</figure>
```

The Angular compiler then generates this TypeScript code:

```
// ...
<!-- wrong computed name in the generated code -->
property('src', component.ponyImageUr());
advance(2);
textInterpolate(component.ponyModel().name);
```

which raises an error in the TypeScript compilation:

```
Property 'ponyImageUr' does not exist on type 'Pony'.
```

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 `input()` in your component, it will be fine in JiT mode (because JavaScript has no notion of private property), but will break in AoT mode (as the generated

TypeScript code of another component needs to access the property).

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 code 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.

# Chapter 23. Advanced observables

When you start with Angular, RxJS looks like something hard to use. Newbies generally try to avoid using it. After all, other frameworks don't use observables. Besides, Angular has signals now. So should you really learn RxJS?

Well, first of all, you don't really have a choice for now. Angular still forces you to use it if you want to use its HTTP client, its router or its reactive forms.

But even if it didn't, RxJS really a great piece of software, that can make it much easier to make asynchronous programming easier, more efficient and correct. Asynchronous code relying on promises is often full of race conditions or inefficiencies. And RxJS isn't tied to Angular, so learning it can be beneficial even if you work with other frameworks, or without any framework at all.

## 23.1. Some Like It Hot

We learned in the chapter about Reactive Programming that an `Observable` represents a sequence of events to which we can subscribe.

But there is a distinction to understand between two kinds of Observables: *cold* ones and *hot* ones.

Cold observables 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 `HttpClient` class are cold observables: they only send the request when you `subscribe`. And each call to `subscribe` sends a new request and results in a new stream of events (limited to one event in this specific case).

Hot observables are different: they emit events without caring if someone subscribed. You can think of them as live television: you turn on the TV and you land in the middle of a show, that could have started minutes or hours ago. The `valueChanges` observable of a `FormControl` is a hot observable. You will not receive the values emitted before the moment you subscribed, only the values from the moment you subscribe. And unlike with cold observable two different subscribers will receive the same events.

## 23.2. Unsubscriptions

In our Ponyracer app, a live race can be represented by an observable, emitting the positions of the ponies. When the race is over, the observable stops emitting events.

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 forgot to unsubscribe, then the `next` callback will continue to do its job every time an event occurs. Even if the component is no longer displayed!

Not unsubscribing can lead to memory leaks and all sorts of trouble (unnecessary network traffic, server overload, etc.). So the best practice in general is to make sure you unsubscribe before the component is destroyed.

One way of doing that is to store the `Subscription` returned by `subscribe` in a property of the

component, and in the `ngOnDestroy` method, call `unsubscribe` on this subscription:

```
export class LiveRace implements OnDestroy {
  private readonly subscription: Subscription;

  constructor() {
    const raceService = inject(RaceService);
    this.subscription = raceService.live().subscribe(() => {
      // ...
    });
  }

  ngOnDestroy() {
    this.subscription.unsubscribe();
  }
}
```

As we explained before, you're automatically unsubscribed when an observable completes. So another way of avoiding troubles caused by never-ending subscriptions is to make sure that the observable completes when the component is destroyed. Since Angular v16, you can do that easily by using the `takeUntilDestroyed` operator provided by the framework in the `@angular/core/rxjs-interop` package. This operator should be the last operator in the chain, just before the call to `subscribe`. That way, it makes sure that you unsubscribe to all the observable chain when the component is destroyed.

```
export class LiveRace {
  constructor() {
    const raceService = inject(RaceService);
    raceService
      .live()
      .pipe(takeUntilDestroyed())
      .subscribe(() => {
        // ...
      });
  }
}
```

Are there situations where it's safe not to unsubscribe? If you're in doubt, follow the best practice, and unsubscribe. It's interesting however to understand when it's absolutely essential to unsubscribe, and when it's not a big deal not to do it.

Since you're automatically unsubscribed when the observable completes or errors, it's not a big deal not to unsubscribe to HTTP observables for example: they complete or error after a short time, usually before, or shortly after the component is destroyed.

Similarly, if the emitter is destroyed at the same time as the component which subscribes (for example, the `ActivatedRoute` of a routed component, or the form created by a component), then not unsubscribing is not a problem either.

## 23.3. Automatic unsubscriptions

A good way of making sure you never forget to unsubscribe is… to avoid subscribing in the first place. Well, something has to subscribe, otherwise nothing will happen. But if you let the framework subscribe for you, it will also unsubscribe for you.

That's not always possible. To trigger side effects, explicitly subscribing is pretty much unavoidable.

But if the goal of the observable is to get things to display, then you can let the framework subscribe and unsubscribe for you, using `toSignal`:

```
export class LiveRace {
  protected readonly liveRace: Signal<LiveRaceModel | undefined> = toSignal(inject
(RaceService).live());
}
```

That's my preferred solution. Before signals were introduced though, another way was frequently used (and might still be used in your projects): the `async` pipe.

The idea is to expose the observable itself as a property of the component, and to let the `async` pipe subscribe and unsubscribe for you, directly from the template:

```
@Component({
  selector: 'ns-live-race',
  template: `{{ (liveRace$ | async)?.name }}`,
  imports: [AsyncPipe]
})
export class LiveRace {
  protected readonly liveRace$: Observable<LiveRaceModel> = inject(RaceService).
live();
}
```

Similarly to the `toSignal` function, the `async` pipe can't return a race until the observable has emitted. That's why we use the `?.` operator.

Now, what if we also want to display the start instant of the race. We could do this:

```
@Component({
  selector: 'ns-live-race',
  template: `❌ don't do this
    <div>{{ (liveRace$ | async)?.name }}</div>
    <div>{{ (liveRace$ | async)?.startInstant | date }}</div>`,
  imports: [AsyncPipe, DatePipe]
})
export class LiveRace {
  protected readonly liveRace$: Observable<LiveRaceModel> = inject(RaceService).
live();
```

```
}
```

Can you spot the problem? Two `async` pipes means two subscriptions. So if the observable is cold, two different streams of events are generated (two different HTTP requests and responses for example).

We could use the `shareReplay` operator to transform the cold observable into a hot one. But the usual way to avoid the problem is to make sure to only use the async pipe once, using the following pattern:

```
@Component({
  selector: 'ns-live-race',
  template: `  Do this instead
    @if (liveRace$ | async; as liveRace) {
      <div>{{ liveRace.name }}</div>
      <div>{{ liveRace.startInstant | date }}</div>
    } @else {
      Loading in progress...
    }`,
  imports: [AsyncPipe, DatePipe]
})
export class LiveRace {
  protected readonly liveRace$: Observable<LiveRaceModel> = inject(RaceService).
live();
}
```

## 23.4. Leveraging operators

We saw a few operators up until now, 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 you 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 such a component would look like:

```
import { Component, inject, Signal } from '@angular/core';
import { toSignal } from '@angular/core/rxjs-interop';
import { NonNullableFormBuilder, ReactiveFormsModule } from '@angular/forms';
import { PonyModel, PonyService } from './pony-service';

@Component({
  selector: 'ns-typeahead',
  template: `
    <div>
      <input [formControl]="input" />
      <ul>
        @for (pony of ponies(); track pony.id) {
          <li>{{ pony.name }}</li>
        }
      </ul>
    </div>
  `,
  imports: [ReactiveFormsModule]
})
export class PonyTypeAhead {
  protected readonly input = inject(NonNullableFormBuilder).control('');
  protected readonly ponies: Signal<Array<PonyModel>>;

  constructor() {
    const ponyService = inject(PonyService);
    // todo: do something with the input
  }
}
```

In the constructor, 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 a HTTP request behind the scenes to fetch the results from the server, so it returns an `Observable<Array<PonyModel>>`, an observable that emits an array of ponies.

Let's subscribe to this method to update the `ponies` field of our component:

```
const ponies = signal<Array<PonyModel>>([]);
this.ponies = ponies;
this.input.valueChanges.subscribe(value => {
  this.ponyService.search(value).subscribe(results => ponies.set(results));
});
```

OK, that works. But it's not ideal. We have many subscriptions that should be unsubscribed. And there is actually a race condition: the second search result could be emitted before the first one.

We could transform the `Observable<string>` by applying the `map` operator. But since the search method returns an `Observable<Array<PonyModel>>`, we would end up with an `Observable<Observable<Array<PonyModel>>>`. What we want is an `Observable<Array<PonyModel>>`, which emits the successive search results.

We should use one of the flattening operators that RxJS provides. Let's try with `mergeMap`:

```
const ponies$ = this.input.valueChanges.pipe(mergeMap(value => this.ponyService.
search(value)));
this.ponies = toSignal(ponies$, { initialValue: [] });
```

Wow, much more elegant! Every time a new pony name is emitted by the source observable, it creates a new observable of ponies, and it merges all the events of all these observables into a single one. But this has the same race condition as the previous attempt: the second search result could be emitted before the first one.

Let's try with `concatMap`:

```
const ponies$ = this.input.valueChanges.pipe(concatMap(value => this.ponyService
.search(value)));
this.ponies = toSignal(ponies$, { initialValue: [] });
```

This time the search results are correctly ordered. But since the observables are concatenated, the second search request is only sent once the response to the first one has arrived. So it's much too slow, and unnecessarily displays all the successive results.

The correct flattening operator to use here is the one we already talked about: `switchMap`.

```
const ponies$ = this.input.valueChanges.pipe(switchMap(value => this.ponyService
.search(value)));
this.ponies = toSignal(ponies$, { initialValue: [] });
```

When a new pony name is emitted, `switchMap` unsubscribes from the previous observable before creating and subscribing to the next. So we get the results in order, and we don't wait to get a result before fetching the next one.

OK, now let's discard the queries that are less than three characters. Easy: we just have to use a `filter` operator!

```
const ponies$ = this.input.valueChanges.pipe(
  filter(query => query.length >= 3),
  switchMap(value => this.ponyService.search(value))
);
```

```
this.ponies = toSignal(ponies$, { initialValue: [] });
```

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`:

```
const ponies$ = this.input.valueChanges.pipe(
  filter(query => query.length >= 3),
  debounceTime(400),
  switchMap(value => this.ponyService.search(value))
);
this.ponies = toSignal(ponies$, { initialValue: [] });
```

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 from the last one? Of course we can, with `distinctUntilChanged`:

```
const ponies$ = this.input.valueChanges.pipe(
  filter(query => query.length >= 3),
  debounceTime(400),
  distinctUntilChanged(),
  switchMap(value => this.ponyService.search(value))
);
this.ponies = toSignal(ponies$, { initialValue: [] });
```

Last thing: we need to properly handle the errors. We know that the `valueChanges` will not signal any error, but our `ponyService.search()` observable might throw: 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 and replace the faulty observable by one that emits an empty result:

```
const ponies$ = this.input.valueChanges.pipe(
  filter(query => query.length >= 3),
  debounceTime(400),
  distinctUntilChanged(),
  switchMap(value => this.ponyService.search(value).pipe(catchError(() => of([]))))
);
this.ponies = toSignal(ponies$, { initialValue: [] });
```

Quite nice, don't you think? We now only trigger a search when the user enters a text with more than 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.

## 23.5. Using Subjects as triggers

Another common pattern in Angular is to use a `Subject` to trigger some action. For example, let's say we want to display a list of races:

- when the component first loads
- every time the user clicks on a refresh button

We can use a `Subject` for that. A `Subject` is a hot observable, which emits a new event when you call its `next` method. In the code below, `refreshTrigger` is used to emit a new event every time the user clicks on the refresh button. We use this subject as the source of the subscription to the `raceService.list` method. As we also want to call the method when the component is first loaded, we use the `startWith` operator to emit an event when the component is initialized.

*races.ts*

```
private readonly refreshTrigger = new Subject<void>();
protected readonly races: Signal<Array<RaceModel> | undefined>;

constructor() {
  const raceService = inject(RaceService);
  const races$ = this.refreshTrigger.pipe(
    startWith(undefined),
    switchMap(() => raceService.list())
  );
  this.races = toSignal(races$);
}

protected refresh() {
  this.refreshTrigger.next();
}
```

## 23.6. 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, `new Observable(observer ⇒ {})`.

The function passed as a parameter to the constructor is called the `subscribe` function: it will be responsible for emitting events and errors, and for completing when done.

For example, if you want to create an Observable which emits 1, then 2, then completes, you could do:

```
const numbers = new Observable(observer => {
  observer.next(1);
  observer.next(2);
  observer.complete();
});
```

Now we could subscribe to such an observable:

```
numbers.subscribe({
  next: number => console.log(number),
  error: error => console.log(error),
  complete: () => 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';

export class HelloService {
  get(): Observable<string> {
    return new Observable(observer => {
      const interval = setInterval(() => observer.next('hello'), 2000);
    });
  }
}
```

The function passed to `new Observable()` 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';

export class HelloService {
  get(): Observable<string> {
    return new Observable(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.

## 23.7. Managing state with stores (NgRx, NGXS, Elf and friends)

The Angular ecosystem has many state management libraries, usually called *stores*: NgRx, NgRx component store, NGXS, Elf, and probably others.

During our trainings, we often get the questions: "should we use one?" or "which one is the best?". We can't really answer that question. Every project has its own needs. Every developer team has its own preferences and priorities.

We do have an opinion, though: they're not a silver bullet, and they are not a solution to avoid having to learn RxJS. They're based on RxJS, and you'd better know RxJS pretty well before using a store.

For *our* various projects, we've always thought that they were not worthy enough to compensate for the additional complexity and boilerplate that they bring. But your mileage may vary. Make some experiments and see by yourself.

Beware of this trap though: they look smart; you managing to use them makes you smart too; but that doesn't mean that they're a good fit for your project and your team. We also get a lot of feedbacks telling us that the lead developer chose to use a store, and all the other developers on the team suffered from that choice.

## 23.8. Conclusion

I hope you enjoyed this small chapter on observables. They can also be used 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!

We have several exercises that leverage RxJS and let you discover a lot of operators:

- Bet on a pony 🦄
- Live 🦄
- Observable tips and tricks 🦄
- Boost a pony 🦄
- Reactive user score 🦄

# Chapter 24. Advanced components and directives

## 24.1. Input transforms

Since Angular v16.1, it is possible to transform an input with the `transform` option of the `@Input` decorator. Of course, the `input()` function also supports this feature.

It allows transforming the value passed to the input before it is stored in the input signal. The `transform` option requires a function that takes the value as input and returns the transformed value. As the most common use cases are to transform a string to a number or a boolean, Angular provides two built-in functions to do that: `numberAttribute` and `booleanAttribute` in `@angular/core`.

Here is an example of using `booleanAttribute`:

```
readonly disabled = input(false, { transform: booleanAttribute });
```

This will transform the value passed to the input to a boolean so that the following code will work:

```
<ns-button disabled />
<ns-button disabled="true" />
<!-- Before, only the following was properly working -->
<ns-button [disabled]="true" />
```

The `numberAttribute` function works the same way but transforms the value to a number.

```
readonly value = input(0, { transform: numberAttribute });
```

It also allows to define a fallback value, in case the input is not a proper number (default is `NaN`):

```
readonly value = input(0, { transform: (value: unknown) => numberAttribute(value, 42)
});
```

This can then be used like this:

```
<ns-value value="42" />
<ns-value value="not a number" />
<!-- Before, only the following was properly working -->
<ns-value [value]="42" />
```

## 24.2. View queries: `viewChild`

In the Template chapter, we talked about a nice feature called "local variables", allowing you 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" />
@if (loginCtrl.dirty && loginCtrl.hasError('required')) {
  <div>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 is 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` function.

```
@Component({
  selector: 'ns-login',
  template: `<input #loginInput name="login" [(ngModel)]="credentials.login" required
/>`,
  imports: [FormsModule]
})
export class Login implements AfterViewInit {
  protected readonly credentials = { login: '' };

  readonly loginInput = viewChild.required<ElementRef<HTMLInputElement>>('loginInput
');

  ngAfterViewInit(): void {
    this.loginInput().nativeElement.focus();
  }
}
```

The `viewChild` and `viewChild.required` functions are the modern, signal-based alternatives to the `@ViewChild` decorator, in the same way as `input` and `input.required` are the modern, signal-based alternatives to the `@Input` decorator.

We declare a field called `loginInput`, and we initialize it with `viewChild.required`. This function needs a selector as parameter: here we use the local variable declared in our template. The function

indicates to the framework that it needs to query the template to find an element with this local variable name. The signal will be initialized with this element, of type `ElementRef<T>`. This type has only one field, `nativeElement` of type `T`, 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 the constructor or 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 Focus 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 into our directive:

```
@Directive({
  selector: '[nsFocus]'
})
export class Focus implements AfterViewInit {
  protected readonly element = inject<ElementRef<HTMLElement>>(ElementRef);

  ngAfterViewInit(): void {
    this.element.nativeElement.focus();
  }
}
```

And we're done: using this directive will give the focus to its host element!

Let's go back to our `viewChild` function: 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>
  `,
  imports: [FormsModule]
})
export class LoginForm {
  readonly credentialsForm = viewChild.required(NgForm);

  protected authenticate(): void {
    if (this.credentialsForm().valid) {
      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 queried element is destroyed, the signal will be set to `undefined`.

This function also has a twin called `viewChildren`. Unlike `viewChild` which will get a reference to one element matching the selector (the first if there are several ones), `viewChildren` will get a reference to all the matching elements.

Let's say we have a `Race` displaying a list of `Pony` - we can easily be notified every time a `Pony` is added or removed:

```
@Component({
  selector: 'ns-race',
  templateUrl: './race.html',
  imports: [Pony]
})
export class Race {
  readonly raceModel = input.required<RaceModel>();
  readonly ponies = viewChildren(Pony);

  constructor() {
    effect(() => {
      console.log(this.ponies().length);
    });
```

```
  }
}
```

## 24.3. Content: ng-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 CSS framework. The template of such a card looks like this:

```
<div class="card">
  <div class="card-body">
    <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, 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-body">
        <h4 class="card-title">{{ title() }}</h4>
        <p class="card-text">{{ text() }}</p>
      </div>
    </div>
  `
})
export class Card {
  readonly title = input('');
  readonly text = input('');
}
```

And then use it like this:

```
<ns-card title="Card title" text="Some quick example text" />
```

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>.

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-body">
    <h4 class="card-title">{{ title() }}</h4>
    <p class="card-text">
      <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-body">
    <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>
  <span title>Card <strong>title</strong></span>
  <p content>Some quick <strong>example</strong> text</p>
</ns-card>
```

This will produce the following result:

```
<div class="card">
  <div class="card-body">
    <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>
```

Since Angular v18, it is possible to define a fallback content, which will be used if no content is provided by the parent:

```
<div class="card">
  <div class="card-body">
    <h4 class="card-title">
      <ng-content select="[title]">Default title</ng-content>
    </h4>
    <p class="card-text">
      <ng-content select="[content]"></ng-content>
    </p>
  </div>
</div>
```

## 24.4. Content queries: `contentChild`

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 functions: `contentChild` and `contentChildren`.

Let's say you are building another UI component based on Bootstrap, 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 title="About" />
</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 `Tab`:

```
@Directive({
  selector: 'ns-tab'
})
export class Tab {
  readonly title = input('');
}
```

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 `Tabs`:

```
@Component({
  selector: 'ns-tabs',
  template: `
    <ul class="nav nav-tabs">
      @for (tab of tabs(); track tab) {
        <li class="nav-item">
          <a class="nav-link">{{ tab.title() }}</a>
        </li>
      }
    </ul>
  `,
  imports: []
})
export class Tabs {
  readonly tabs = contentChildren(Tab);
}
```

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` function allows you to do.

To grab the list of the tabs, we need to use `contentChildren`, with `Tab` as parameter. This gives us an array of tabs, that you can use in a `for` loop. As each element of this list is a `Tab`, 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" />
  </div>
```

```
  <ns-tabgroup>
    <ns-tab title="About" />
  </ns-tabgroup>
</ns-tabs>
```

Then the array in `Tab` will only contain the first `Tab`. `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 pass to `contentChildren`:

```
@Component({
  selector: 'ns-tabs',
  template: `
    <ul class="nav nav-tabs">
      @for (tab of tabs(); track tab) {
        <li class="nav-item">
          <a class="nav-link">{{ tab.title() }}</a>
        </li>
      }
    </ul>
  `,
  imports: []
})
export class TabsWithDescendants {
  readonly tabs = contentChildren(Tab, { descendants: true });
}
```

Now it finds all the `Tab` again!

The `tabs` property is a signal, so you can also be notified of the changes, like we saw for `viewChildren`. Once again, the signal value is not accessible in the component constructor or even in `ngOnInit`. To be sure that the content can be queried, use the `ngAfterContentInit` lifecycle hook. You can also use the `ngAfterContentChecked` hook, which is called every time the content is checked.

Try our exercise Advanced components 🦄! You'll build a component with `ng-content`!

## 24.5. Conditional and contextual content projection: `ng-template` and `ngTemplateOutlet`

`ng-content` falls short if you want to insert some dynamic HTML content inside your template conditionally, or in a loop, or if you want to provide some context to this dynamic content.

Let's illustrate such a case with a simple enough example. We would like to create a progress bar component. This component accepts a min value, a max value, and a value, computes a percentage, and displays it inside a progress bar.

Let's start with the skeleton of our component:

```
export class Progress {
  readonly min = input(0);
  readonly max = input(100);
  readonly value = input.required<number>();
  protected readonly percentage = computed(() => (100 * (this.value() - this.min())) /
(this.max() - this.min()));
}
```

And its template:

```
<div class="progress">
  <div
    class="progress-bar"
    role="progressbar"
    [style.width.%]="percentage()"
    [attr.aria-valuenow]="value()"
    [attr.aria-valuemin]="min()"
    [attr.aria-valuemax]="max()"
  >
  </div>
</div>
```

Nothing new until now.

But we would also like to display the percentage value inside the component. And the user of the component should also be able to customize the way the percentage is displayed.

So, inside the template, we want the following, where the formatter is something that the user of the component can provide as input.

```
<div class="progress">
  <div
    class="progress-bar"
    role="progressbar"
    [style.width.%]="percentage()"
    [attr.aria-valuenow]="value()"
    [attr.aria-valuemin]="min()"
    [attr.aria-valuemax]="max()"
  >
    @if (formatter(); as f) {
      <span>
      </span>
    } @else {
      {{ percentage() | number }}%
    }
```

```
  </div>
</div>
```

This formatter is not a value. It's not a function either, because the user must be able to format the value using Angular-enriched HTML.

Such a piece of Angular-enriched HTML snippet that can be passed around and inserted anywhere is modelled by Angular by an `<ng-template>`. Its TypeScript counterpart is an object of type `TemplateRef`.

In this case, the formatter's responsibility is to display a percentage. So it needs a rendering context containing this percentage. This rendering context is the generic type of the `TemplateRef` that the component will accept as input.

```
interface ProgressContext {
  percentage: number;
}

export class Progress {
  // ...
  readonly formatter = input<TemplateRef<ProgressContext>>();
  protected readonly formatterContext = computed<ProgressContext>(() => ({
    percentage: this.percentage()
  }));
}
```

Now, how can we insert this `TemplateRef` inside the component's template and give it its context? That's the role of the structural directive `*ngTemplateOutlet`. Here's how we use it:

```
@if (formatter(); as f) {
  <span>
    <ng-container *ngTemplateOutlet="f; context: formatterContext()" />
  </span>
} @else {
  {{ percentage() | number }}%
}
```

Finally, how can the user use this progress component and provide a formatter? Simply by defining an element `<ng-template>`, assigning a variable to it, and pass this variable as the `formatter` input to the progress component.

Inside the `<ng-template>`, we can of course access all the properties of the current component. But we can also access the elements of the context, coming from the progress component, by using this weird `let-p="percentage"` syntax. `p` is simply an alias to the context element named `percentage` (we could choose any other alias name, like for example `let-progress="percentage"`).

```
<!-- without formatter -->
```

```
<ns-progress [value]="75" />

<!-- with formatter -->
<ng-template #myFormatter let-p="percentage"><strong>Your</strong> progress: {{ p |
number }}%</ng-template>
<ns-progress [value]="75" [formatter]="myFormatter" />
```

If, inside the context interface, we choose to name a property `$implicit` (instead of `percentage` here for example), then we can simply use `let-p` instead of `let-p="percentage"` to access this property.

In fact, structural directives work like that. All the directives that we use with a *, like `ngIf` or `ngFor`, have an `<ng-template>` as host element. Angular defines a *micro-syntax* for structural directives, which allows declaring a directive on a `<ng-template>` in a concise way. Let's take `ngFor` as an example (`ngFor` is the directive we used to use before `@for` was introduced). When we write a simple `ngFor` as the following:

```
<div *ngFor="let pony of ponies; i as index">
  {{ i }} - {{ pony }}
</div>
```

Then it is the same as writing the following template:

```
<ng-template ngFor [ngForOf]="ponies" let-pony let-i="index">
  <div>{{ i }} - {{ pony }}</div>
</ng-template>
```

It's less readable, of course. That's why we always use `*ngFor`. But it shows that in fact, Angular itself uses this pattern based on `ng-template`! `ngFor` is nothing more than a directive applied on `ng-template`, with an input `ngForOf` that expects a collection, and a context containing an `$implicit` property with the current element, and other properties like the index.

This pattern is quite powerful, and is of great help to design customizable components.

## 24.6. 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 `InputClear`. 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.

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 InputClear {
  private readonly element = inject<ElementRef<HTMLInputElement>>(ElementRef);
}
```

And use our directive like this:

```
<input nsInputClear />
```

We now need to react to a 'dblclick' event on our host element (here, the `input`) to clear the value.

That's where we can use the `host` metadata in the `@Component` decorator. The syntax is almost identical to the one we would use in a template:

```
host: {
  '(eventtype)': 'statement()'
}
```

In our case, we can write:

```
@Directive({
  selector: '[nsInputClear]',
  host: {
    '(dblclick)': 'clearContent()'
  }
})
export class InputClear {
  private readonly element = inject<ElementRef<HTMLInputElement>>(ElementRef);

  protected clearContent(): void {
    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]',
  host: {
```

```
    '(window:resize)': 'resize($event)'
  }
})
export class WindowResize {
  protected resize(event: Event): void {
    const innerWidth = (event.target as Window).innerWidth;
    console.log(`The screen is being resized to ${innerWidth}`);
  }
}
```

Another way to achieve the same thing is to decorate a method of the directive with `@HostListener`. This was the recommended way in previous versions of Angular, but the host metadata are now the preferred way.

## 24.7. Host binding

We often need to automatically add a CSS class or style, or bind a specific DOM property on the host element of a directive or component. The host metadata, as we just saw, allow to declare event bindings on the host element. We can use the same host metadata to also declare property bindings.

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 AddClassIfRequired {
}
```

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:

```
private readonly formControl = inject(FormControlName, { optional: true });
private readonly ngModel = inject(NgModel, { optional: true });
```

But that's not the best we can do. Indeed, both directives inherit the same base class: `NgControl`. So instead of injecting one or the other, we can simply ask Angular to provide us with the common `NgControl`!

```
@Directive({
  selector: '[nsAddClassIfRequired]',
})
export class AddClassIfRequired {
  private readonly control = inject(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 host property binding enters the scene! Once again, the syntax is almost identical to the syntax we would use in a template:

```
host: {
  '[prop]': 'expression'
}
```

And that would automatically update the host's property `prop`, every time the `expression` changes.

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 control's error.

```
@Directive({
  selector: '[nsAddClassIfRequired]',
  host: {
    '[class.is-required]': 'isRequired'
  }
})
export class AddClassIfRequired {
  private readonly control = inject(NgControl);

  protected get isRequired(): boolean {
```

```
    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.

Another way to achieve the same thing is to decorate a property with `@HostBinding`. This was the recommended way in previous versions of Angular, but the host metadata are now the preferred way.

## 24.8. DOM manipulation with `afterEveryRender` or `afterNextRender`

In Angular 16.2, two new functions have been added to Angular: `afterEveryRender` and `afterNextRender`. They're not traditional hooks implemented as methods of a component. Instead, those are functions that a component can call, whenever it needs to, in order to register a function that must be executed every time (for `afterEveryRender`) or the next time (for `afterNextRender`) Angular has updated the DOM of the whole application.

They're thus useful for DOM manipulation. For example, if some event listener method modifies the state of the application, which will in turn have impacts on the DOM, and if you want to apply some transformation on this new DOM, calling `afterNextRender` inside the event listener method is a good way of doing that.

Beware though: these functions, like `inject`, are contextual functions. If you call them of a constructor, you'll need to pass the component's injector as an option.

Here's an example of usage of `afterNextRender`: clicking a button allows displaying an input to enter the name of a new pony. Once the input has appeared in the DOM, it must take the focus immediately. Of course, simply setting the boolean to `true` doesn't make the input appear immediately. We must wait until Angular has rendered the DOM before accessing the input and giving it the focus. Using `afterNextRender` is a perfect way of doing that. And since we're not calling it from the constructor, we must pass it the injector associated to the component, that has been injected in the constructor.

```
@Component({
  selector: 'ns-new-pony',
  template: `
    <button (click)="showPonyForm()">New pony</button>
    @if (ponyFormDisplayed()) {
```

```
      <label for="pony-name">New pony name:</label>
      <input id="pony-name" #ponyName />
    }
  `
})
export class NewPony {
  protected readonly ponyFormDisplayed = signal(false);
  readonly ponyName = viewChild<ElementRef<HTMLInputElement>>('ponyName');

  private readonly injector = inject(Injector);

  protected showPonyForm() {
    // display the form
    this.ponyFormDisplayed.set(true);
    // and give the focus to the new input once the DOM is rendered
    afterNextRender(() => this.ponyName()!.nativeElement.focus(), { injector: this
.injector });
  }
}
```

Another feature of those functions is that the callback you give them is never executed on the server, when you use Server-Side rendering (SSR). So if you need to only execute code on the client, for example because it accesses the `window` or some other API that is not available on the server, placing this code inside a function passed to `afterNextRender()` is a good way of doing it.

Try our exercise Advanced directives 🦄! You'll build several directives that interact with each other and with `ContentChild` and `HostBinding`! Then you'll learn how libraries like ng-bootstrap use these pattern in the exercise Integrate with a UI library 🦄 and how to add charts to your application in the exercise Charts in your app 🦄.

# Chapter 25. Angular modules

Until version 14 of Angular, applications were organized into Angular modules. Many existing applications probably still are. But even in newer applications using standalone components, pipes and directives, the concept of Angular module, or `NgModule` for short, is still present: you can, or even must, import some of them (`CommonModule`, `ReactiveFormsModule` for example) in your components.

Maybe you're reading this book in order to embark on an existing project where Angular modules are still in use. This chapter will explain their principles, rules, and usages. We encourage you to use standalone components in your new applications though, and even to migrate your NgModule-based applications to standalone. The Angular CLI provides commands to help you automate most of it.

## 25.1. A compilation unit

In the simplest Angular application still using modules, you'll find one Angular module, the *root* module, conventionally named `AppModule`. This module typically looks like this:

```
@NgModule({
  declarations: [App, Home, About, Pony],
  imports: [BrowserModule, HttpClientModule, RouterModule.forRoot(APP_ROUTES)],
  providers: [],
  bootstrap: [App]
})
export class AppModule {}
```

What can we say about this module?

The `declarations` property declares the components, pipes and directives that are part of this module. Those are not standalone. If they were standalone, then they would have to be listed in the `imports` property instead.

the only difference between a standalone component (or pipe, or directive) and a "normal", non-standalone component is that non-standalone components have `standalone: false` in their decorator, and can't have `imports` in their decorator. The module that they belong to decides what they can use in their template.

An Angular module defines a compilation unit. Since they are part of the same module, all these components can use each other. For example, both `Home` and `About` can use the `Pony` in their template.

The `imports` property defines the list of other Angular modules, as well as standalone components, pipes and directives, that are imported into this module. Since we import `RouterModule`, the components can also use the directives of the router module, like `routerLink`. Since we don't import `ReactiveFormsModule`, they cannot use the directives from that module, like `formGroup`. Importing the `HttpClientModule` (which is now deprecated) allows importing the provider for the `HttpClient`

service, which means it can be injected in any service of the application. Importing `BrowserModule`, which transitively imports and exports `CommonModule` is what makes it possible to use the common directives and pipes (`*ngIf`, `ngClass`, `date` etc.) in the components of this module.

The `providers` array defines providers, the same way as they are defined when calling the `bootstrapApplication()` function in standalone applications. Since most services use `providedIn: 'root'`, you won't need to list them there. But it can still be necessary if a service needs to be explicitly provided to configure the application.

Finally `bootstrap` contains the root component (or, much more rarely, the root components) of the application, that must be bootstrapped, because they're at the root of the tree of components.

## 25.2. Module composition

Bigger applications are composed of additional modules, that we can arrange in two categories:

1. Reusable modules, containing reusable components, directives, pipes and sometimes service providers. Those are intended to be imported by other modules.
2. Functional modules, often lazy-loaded, corresponding to a set of routes of the application

Let's start with reusable modules. Suppose the `Pony` must be used in `Races` which is declared in an additional functional module `RacesModule`. As we've seen, in order to make it available to `Races`, we should add `Pony` in the `declarations` of `RacesModule`. Angular, however, won't let you do that: a component may be declared in only one module of the application, and it's already declared in `AppModule` (which also needs it for its `Home`).

The solution is to extract `Pony` in yet another module, sometimes called a *shared* module:

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

This module declares the `Pony` that it contains. It imports `CommonModule` so that `Pony` can use the common Angular directives. And most importantly, its `exports` property also contains `Pony`. This is what allows other modules to use the `Pony` by importing the `PonyModule`. That's what `AppModule` and `RacesModule` will thus do in order to be able to use `Pony`:

```
@NgModule({
  declarations: [App, Home, About],
  imports: [BrowserModule, HttpClientModule, RouterModule.forRoot(APP_ROUTES),
PonyModule],
  providers: [],
  bootstrap: [App]
})
```

```
export class AppModule {}
```

The temptation is big to put all the reusable components in a single `SharedModule`. As the application grows, however, this module will become huge. So a better option is to make tiny reusable modules, so that other modules can import only what they actually need. Such tiny reusable modules, exporting only one component, are known as *SCAM*s (Single Component Angular Modules). The boilerplate that these modules represents is one of the reasons why Angular decided to introduce standalone components: you can see them as a component and its own module.

## 25.3. Functional, routed modules

Functional modules are not much different from reusable modules. The components that they declare, however, are not supposed to be used inside components of other modules. So they're declared, but not exported.

```
@NgModule({
  declarations: [Races],
  imports: [CommonModule, PonyModule, RouterModule.forChild(RACES_ROUTES)]
})
export class RacesModule {}
```

The root module can import the additional functional modules, but then they would be bundled as part of the main bundle, and thus loaded eagerly, at startup. Most of the time, we want to lazily load the functional modules. The way to do that is not very different from what we did to lazy load routes using standalone components. Instead of lazy-loading routes, we will lazy-load the functional module itself. For example, in the application routes:

```
{
  path: 'races',
  loadChildren: () => import('./races/races.module').then(m => m.RacesModule)
}
```

Hopefully now, you're able to understand how an application is organized in Angular modules. The rules are not that hard to grasp, but as you've seen, Angular modules don't really provide any functionality to the application, and make the dependencies of components a bit harder to find. That's why we've chosen, in this book, to promote the usage of standalone components.

# Chapter 26. Internationalization

*Alors comme ça, tu veux internationaliser ton application?*

OK, don't worry if you didn't understand anything of this French introduction. Your role as a developer, fortunately, is not to translate your application into French, Spanish, or whatever other language. What you can do, though, is to allow this to happen. This chapter explains how to achieve that.

## 26.1. The locale

We already mentioned internationalization before, in the chapter about pipes. Four of the built-in Angular pipes deal with internationalization. Those are the `number`, `percent`, `currency` and `date` pipes. Until Angular 5, they used to rely on the standard JavaScript Internationalization API, which is supposed to be provided by the browser. But as that was not always the case, and there were numerous bugs and inconsistencies between browsers, the pipes have been completely overhauled in Angular 5.0.

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' -->
```

```
  `,
  imports: [DecimalPipe]
})
class DefaultLocale {
  protected readonly locale = inject(LOCALE_ID);
}
```

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 the providers of the application. Beware though: this changes the locale, but another step is required to bundle the locale-specific data (month translations, number formatting rules, etc.) with your application. Angular only bundles the `en-US` data by default.

```
import '@angular/common/locales/global/fr';
```

Here's an example showing its effect on our component:

```
bootstrapApplication(App, {
  providers: [
    { provide: LOCALE_ID, useValue: 'fr-FR' }
  ]
}).catch(err => console.error(err));

@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' -->
  `,
  imports: [DecimalPipe]
})
export class CustomLocale {
  protected readonly locale = inject(LOCALE_ID);
}
```

All the pipes that handle internationalization can also take a locale as their last parameter. You can then change it dynamically if needed:

```
@Component({
  selector: 'ns-locale',
  template: `
    <p>The locale is {{ locale }}</p>
    <!-- will display 'en-US' -->
```

```
    <p>{{ 1234.56 | number: '1.0-3' : 'fr-FR' }}</p>
    <!-- will display '1 234,56' -->
  `,
  imports: [DecimalPipe]
})
class DefaultLocaleOverridden {
  protected readonly locale = inject(LOCALE_ID);
}
```

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.

## 26.2. Default currency

The `currency` pipe allows you to specify which currency you want to use by providing an ISO string like `USD`, 'EUR'... If you don't provide one, it uses `USD` by default. So it is a bit cumbersome to have to specify the currency every time, if your application only uses `EUR` for example.

Angular 9 introduced the possibility of configuring the default currency globally using the token `DEFAULT_CURRENCY_CODE`.

```
bootstrapApplication(App, {
  providers: [
    { provide: DEFAULT_CURRENCY_CODE, useValue: 'EUR' },
    { provide: LOCALE_ID, useValue: 'fr-FR' }
  ]
}).catch(err => console.error(err));
```

And you can then use `currency` without specifying `EUR` in a component. You can also get the currency via dependency injection of course:

```
@Component({
  selector: 'ns-currency',
  template: `
    <p>The currency is {{ currency }}</p>
    <!-- will display 'EUR' -->

    <p>{{ 1234.56 | currency }}</p>
    <!-- will display '1 234,56 €' -->
  `,
  imports: [CurrencyPipe]
})
class DefaultCurrencyOverridden {
  protected readonly currency = inject(DEFAULT_CURRENCY_CODE);
}
```

## 26.3. 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, and was completely rewritten in version 9.0. No need for an external dependency anymore. And it uses basically the same strategy based on keys, but with a big difference: it happens at compile-time.

With the compilation time strategy, you prepare one version of your application per locale. When you build your app, Angular parses all the HTML templates of your components, and transforms them to JavaScript code that, basically, analyzes the changes in the model and modifies the DOM accordingly. The translation happens at the end of this compilation phase. 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.

Starting with version 9.0, the Angular team introduced a new `@angular/localize` package. If you use the CLI, you can just run `ng add @angular/localize` and the CLI adds it for you in the proper place.

This new package introduces a `$localize` global function, that is used under the hood for the localization, and that we'll be able to use in the future to translate messages in your code.

## 26.4. 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. 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.

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 with 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.

### 26.4.1. 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 `extract-i18n` command provided by Angular CLI:

```
ng extract-i18n --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-US" datatype="plaintext" original="ng2.template">
    <body>
```

```
      <trans-unit id="7627914200888412251" datatype="html">
        <source>Welcome to Ponyracer</source>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app/app.html</context>
          <context context-type="linenumber">2,3</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" target-
language="fr">
    <body>
      <trans-unit id="7627914200888412251" 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="6888363845978110110" datatype="html">
        <source> Welcome to Ponyracer.
</source>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app/app.html</context>
          <context context-type="linenumber">6,8</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>
  <context-group purpose="location">
    <context context-type="sourcefile">src/app/app.html</context>
    <context context-type="linenumber">10,11</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>
  <context-group purpose="location">
    <context context-type="sourcefile">src/app/app.html</context>
    <context context-type="linenumber">13,15</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" equiv-text="{{ user().firstName
}}"/> <x id="INTERPOLATION_1" equiv-text="{{ user().lastName }}"/>!</source>
  <context-group purpose="location">
    <context context-type="sourcefile">src/app/app.html</context>
```

```
    <context context-type="linenumber">16,17</context>
  </context-group>
</trans-unit>
```

As you can see, this format has several interesting features:

- there is no way a translator could mess up the content of the angular expressions, since they are clearly indicated in the message;
- it's clear that these two interpolations are for the developers but it might also be a good idea to explain that in the description of the message for the translators;
- 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>
  <context-group purpose="location">
    <context context-type="sourcefile">src/app/app.html</context>
    <context context-type="linenumber">21,22</context>
  </context-group>
</trans-unit>
<trans-unit id="home.ponyImage.title" datatype="html">
  <source>Ponies are cool, aren&amp;apos;t they?</source>
  <context-group purpose="location">
    <context context-type="sourcefile">src/app/app.html</context>
    <context context-type="linenumber">23,24</context>
  </context-group>
</trans-unit>
```

Finally, how to translate the last snippet? There is no element where we could place an `i18n` attribute. There is a way to solve the problem: use an `ng-container` element, which won't be rendered in the DOM at runtime:

```
<ng-container i18n="@@home.startMessage">Let's start playing.</ng-container>
```

### 26.4.2. Translating, building and deploying the application

Now that you generated a complete `messages.xlf` file, someone needs to translate it.

A common mistake is to just replace the original source text (in English in our case) with 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" equiv-text="{{ user.firstName
}}"/> <x id="INTERPOLATION_1" equiv-text="{{ user.lastName }}"/>!</source>
  <target>Bienvenue dans Ponyracer <x id="INTERPOLATION" equiv-text="{{ user.firstName
}}"/> <x id="INTERPOLATION_1" equiv-text="{{ user.lastName }}"/> !</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` in `angular.json` by using the `i18n` property:

In versions before 9, to run or build the application in French, you had to pass various options to `ng serve` and `ng build`.

Since version 9, you must specify the locales you want to support, and their associated messages file, in the `angular.json` configuration file by using the `i18n` property:

```
"prefix": "ns",
"i18n": {
  "locales": {
    "fr": "src/locale/messages.fr.xlf"
  }
},
```

You can add as many locales as you want, each one linked to its translations.

You can then run:

```
ng build --localize
```

and the CLI builds the app, and generates as many versions as there are locales defined. In our case, it generates one version for the default locale `en-US`, in the directory `dist/i18n/en-US` (if your project is named `i18n`), and another one for the `fr` locale in the directory `dist/i18n/fr`.

If you want to run `ng serve`, you have to specify one locale, as you can only serve one version.

The easiest way to do so is to define a build `configuration` for this locale in `angular.json`, just like the `production` configuration, but much simpler:

```
"fr": {
  "localize": ["fr"]
}
```

and add a serve configuration:

```
"fr": {
  "buildTarget": "i18n:build:fr"
}
```

you can then run:

```
ng serve --configuration=fr
```

or build for a specific locale with:

```
ng build --configuration=production,fr
```

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 only once, and then the localization tool will generate 3 versions (once for each locale) of the application (this is very fast). Then, you need to 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.

## 26.5. 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. Since Angular 9.0, you can use `$localize` to do so. `$localize` is a tag function that can be applied to a template string (check the ECMAScript 2015 chapter if you need a refresher).

```
protected status = $localize`PENDING`;
```

Then, when you build your application with the CLI, the `$localize` calls are replaced with their translations!

You can also define an ID with the syntax we have in templates, and have dynamic parts in the string:

```
protected greetings = $localize`:@@home.greetings:Welcome ${this.user().firstName}!`;
```

`ng extract-i18n` supports the message extraction from `$localize` calls in the code since CLI v10.1, and it generates:

```
<trans-unit id="home.greetings" datatype="html">
  <source>Welcome <x id="PH" equiv-text="this.user().firstName"/>!</source>
  <context-group purpose="location">
    <context context-type="sourcefile">src/app/app.ts</context>
    <context context-type="linenumber">17</context>
  </context-group>
</trans-unit>
```

It can then be translated as usual (with `PH` the placeholder for the dynamic part):

```
<trans-unit id="home.greetings" datatype="html">
  <source>Welcome <x id="PH" equiv-text="{{ this.user.firstName }}"/>!</source>
  <target>Bonjour <x id="PH" equiv-text="{{ this.user.firstName }}"/> !</target>
</trans-unit>
```

## 26.6. 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" equiv-text="{racesPlanned(), plural, =0 {no race
is planned} =1 {only one race is planned} other
  {{{ racesPlanned() }} races are planned}}" xid="2482273160942454889"/>.
</source>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app/app.html</context>
          <context context-type="linenumber">32,35</context>
        </context-group>
      </trans-unit>
      <trans-unit id="5232470321856793506" 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>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app/app.html</context>
          <context context-type="linenumber">32,33</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" equiv-text="{racesPlanned, plural, =0 {...} =1 {...} other
{...}}"/>..
```

```
</source>
        <target>Bonjour, <x id="ICU" equiv-text="{racesPlanned, plural, =0 {...} =1
{...} other {...}}"/>.</target>
      </trans-unit>
      <trans-unit id="5232470321856793506" datatype="html">
        <source>{VAR_PLURAL, plural, =0 {no race is planned} =1 {only one race is
planned} other {<x id="INTERPOLATION" equiv-text="{{ racesPlanned }}"/> races are
planned} }</source>
        <target>{VAR_PLURAL, plural, =0 {aucune course n'est planifiée} =1 {seule une
course est planifiée} other {<x id="INTERPOLATION" equiv-text="{{ racesPlanned }}"/>
courses sont planifiées} }</target>
      </trans-unit>
```

## 26.7. Runtime i18n with Transloco

In our own experience as developers, and in our experience training hundreds of developers, we've learnt that runtime internationalization is often preferred.

There are several libraries allowing to do that. The most popular seems to be ngx-translate, but it's probably because it was the first one. It works well, but it's now in maintenance mode.

A more modern and complete alternative, which is also very popular, is Transloco.

The basic principles of both libraries are similar:

- configure the library with a set of supported languages, a default language and a loader, which is in charge of dynamically loading translation keys stored in a JSON file;
- use a directive, a pipe or a service to get the translation associated with a given key, from the template or from the TypeScript code.

Transloco is very well documented, so you should be able to learn everything there is to know about it from its documentation. You can also learn from our *I18n* exercise in the Pro Pack.

But here's the crux of it if you just want to see what it looks like.

First define a loader (here using HTTP, but there are other options):

```
@Injectable({ providedIn: 'root' })
export class TranslocoHttpLoader implements TranslocoLoader {
  private readonly http = inject(HttpClient);
  getTranslation(lang: string) {
    return this.http.get<Translation>(`./i18n/${lang}.json`);
  }
}
```

Then configure the library and add the required providers to the application:

```
export const appConfig: ApplicationConfig = {
```

```
  providers: [
    provideHttpClient(),
    provideTransloco({
      config: {
        availableLangs: ['en', 'fr'],
        defaultLang: 'en',
        prodMode: !isDevMode()
      },
      loader: TranslocoHttpLoader
    })
  ]
};
```

And finally use it to translate text in the template:

```
<ng-container *transloco="let t">
  <h1>{{ t('home.title') }}</h1>
  <p>{{ t('home.welcome-message') }}</p>
</ng-container>
```

## 26.8. 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 examples with the `home.` prefix) allows you to know where they are used, and to find them in the code easily. Relying on auto-generated IDs doesn't allow you to have 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 message files in your version control system. This makes sure a branch can have its own additions, which 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.

Try our exercise I18n 🦄 to learn how to configure and use Transloco, the LOCALE_ID, and date-fns to internationalize your applications.

# Chapter 27. Performances

Be careful with premature optimization. Always measure before and after. Beware of the benchmarks you find on the internets: it's pretty easy to make them say what the authors want.

Performances can mean a lot of things: speed, CPU usage (battery consumption), memory pressure… Everything is not important for everybody: you have different needs if you are programming for a mobile website, an e-commerce platform, or a classic CRUD application.

Performances can also be split into different categories, that, once more, won't all matter to you: first load, reload, and runtime performances.

First load is when you open an application for the first time. Reload is when you come back to that application. Runtime performances is what happens when the application is running. Some of the following recommendations are very generic, and could be applied to any framework. We wrote them down because we think it's worth knowing. And because when you talk about performances, the framework is sometimes the bottleneck, but really (really) often not.

## 27.1. First load (bundling, compression, lazy-loading, server side rendering)

When you load a modern Web application in your browser, a few things happen. First, the `index.html` is loaded and parsed by the browser. Then the JS scripts and other assets referenced are fetched. When one of the assets is received, the browser parses it, and executes it if it is a JS file.

### 27.1.1. Asset sizes

So the first tip is very obvious: be careful with your asset sizes!

The assets loading phase depends on how many assets you want to load. A lot will be slow. Big ones will be slow. Especially if the network is not that good, which happens more often than you think: you might test your application on an optical fiber connection, but some of your actual users might be in the middle of nowhere, using slow 3G. Here is what you can do.

### 27.1.2. Bundle your application

When you write your Angular application, you have imports all over the place, and your code is split across hundreds of files. But you don't want your users to load hundreds of files! So before shipping your application, you want to make a "bundle": group all the JavaScript files into one file.

esbuild's job is to take all your JavaScript files (and CSS, and template HTML files) and build bundles. It's not an easy tool to master, but the Angular CLI does a pretty good job at hiding its complexity.

### 27.1.3. Tree-shaking

esbuild (and other bundlers like Webpack) starts from the entry point of your application (the `main.ts` file that the CLI generated for you, and which you probably never touched), then resolves all the imports graph, and outputs the bundle. This is cool because the bundle will only contain the files from your codebase and your third party libraries that have been imported. The rest is not embedded. So even if you have a dependency in your `package.json` that you don't use anymore (so you don't import it anymore), it will not end up in the bundle.

It's even a bit smarter than that. If you have a file `models` exporting two classes, let's say `PonyModel` and `RaceModel`, and then only import `PonyModel` into the rest of the application, but never `RaceModel`, then the bundler only puts `PonyModel` in the final bundle, and drops `RaceModel`. This process is called **tree-shaking**. And every framework and library in the JavaScript ecosystem is fighting hard to be tree-shakable! In theory, it means that your final bundle contains only what is really needed! But in practice, bundlers are a bit conservative, and can't figure out some stuff. For example, if you have a class `Pony` with two methods `eat` and `run`, but you only use `run`, the code of the `eat` method will be in the final bundle. So it's not perfect, but it does a good job.

To avoid bundles that are too large, be wary of the external components and libraries you use, and of their transitive dependencies. What is their additional weight? Are they really useful? Isn't there a native way of doing the same thing, or isn't it simple enough to do it by yourself?

### 27.1.4. Minification and dead code elimination

When your bundle has been built, the code is usually minified and dead code will be eliminated. That means all variables, method names, class names… are renamed to use a one or two characters name through the entire codebase. This is a bit scary and sounds like it could break things, but the tools have been doing a great job.

### 27.1.5. Other assets

While the above sections were about JS specifically, your application also contains other assets, like styles, images, fonts… You should have the same concerns about them, and do your best to keep them at a reasonable size. Applying all kinds of crazy techniques to optimize your JS bundle sizes, but loading several MBs of images, wouldn't have a big impact on your page loading time and your bandwidth! As this is not really the scope of this ebook, I won't dig into this topic, but let me point out a great online resource by Addy Osmani about image optimization: Essential Image Optimization.

Angular has a directive `NgOptimizedImage` that enforces some of these best practices.

```
<img [ngSrc]="imageUrl" alt="Pony" width="100" height="100" />
```

### 27.1.6. Compression

All the modern browsers accept a compressed version of an asset when they ask the server for it. That means you can serve a compressed version to your users, and the browser will unzip it before parsing it. This is a must-do because it will save you tons of bandwidth and loading time.

Every server on the market gives the option of activating the compression of assets or even dynamic http responses. Generally the first user to request an asset will pay the cost of the compression on the fly, and then the following ones will receive the compressed asset directly.

The most common compression algorithm used is GZIP, but some others like Brotli are also popular.

### 27.1.7. Lazy-loading

Sometimes, despite doing your best to keep your JS bundle small, you end up with a big file because your app has grown to several dozens of components, using various third party libraries. And not only will this big bundle increase the time needed to fetch the JavaScript, it will also increase the time needed to parse it and execute it.

The solution to this problem is to use lazy-loading. It means that instead of having a big bundle of JavaScript, you split your application into several parts and tell the bundler to split it in several bundles.

The good news is that Angular (its router in particular) makes this task easy to achieve. You can read our chapter about the router if you want to learn more. The templates can also use the `@defer` instruction to lazy load components and their dependencies. Read the chapter about defer to learn everything about it.

Lazy-loading can vastly improve the loading time, as you can make the first bundle really small, with only what's needed to display the home page, and let Angular load additional bundles on demand, when your user navigates to another part. You can also use prefetching strategies to tell Angular to start loading the other bundles when it's idle.

### 27.1.8. Server side rendering

I'd like to start by saying that this technique is for 0.0001% of you. It's mainly useful for public web sites, but not so much for applications, especially intranet applications. Server side rendering (SSR) is the technique that consists of pre-rendering the pages on the server before serving them to the users. With this, when users ask for `/dashboard`, they will receive a pre-rendered version of the dashboard, instead of receiving the almost empty `index.html`.

It can lead to vast improvements in perceived startup time. Angular CLI offers an option `--ssr` for the `ng new` command which generates a project pre-configured for server side rendering. Angular can then pre-render the pages on a NodeJS server and serve them to your users. The page will display very fast and once Angular has started in the browser, it will do the "hydration" of the page and then run as usual. The hydration consists in taking the control of the DOM: add the necessary event listeners, start the change detection, and sometimes add some parts that can only be rendered on the client-side.

SSR also a big win if you want your website to be crawlable by search engines or social networks which don't execute JavaScript, since you can serve them pre-rendered pages, instead of a blank page.

The bad news is that it's not as easy as adding an option when you create your project. Your application needs to follow some best practices (no direct DOM manipulation for example, as the

server won't have a real DOM to manipulate). Then you need to set up your server and think about the strategy you want to adopt. Do you want to pre-render all pages or just a few? Do you want to pre-render the whole page, with the data fetching and authorization check it will need, or just some critical parts of the page? Do you want to pre-render them on build, or to pre-render them on demand and cache them? Do you want to do this for all the possible profiles and languages or just some? All these questions depend on the type of application you are building, and the effort can vary greatly depending on your goal.

So, again, I would advise you to use server side rendering only if it is critical for your application, and not based on the hype…

## 27.2. Reload (caching, service worker)

Once your user has opened the application once, it's possible to speed up the subsequent visits.

### 27.2.1. Caching

You should always cache the assets of your application (images, styles, JS bundles…). This is done by configuring your server and leveraging the `Cache-Control` and `ETag` headers. All the servers on the market allow you to do so. You can also use a CDN for this purpose, which will additionally allow the users to download the assets from a server close to their location. If you do so, the next time your users open the application, the browser won't have to send a request to fetch the assets because it will have them already.

But a cache is always tricky: you need to have a way to tell the browser "hey, I deployed a new version in production, please fetch the new assets!".

The easiest way to do this is to have a different name for the asset you updated. That means instead of deploying an asset named `main.js`, you deploy `main.xxxx.js` where `xxxx` is a unique identifier. This technique is called cache busting. And, again, the CLI is there for you: in production mode, it will name all your assets with a unique hash, derived from the content of the file. It also automatically updates the sources of the scripts in `index.html` to reflect the unique names, the sources of the fonts, the sources of the stylesheets, etc.

If you use the CLI, you can safely deploy a new version and cache everything, except the `index.html` (as this will contain the links to the fresh assets deployed)!

### 27.2.2. Service Worker

If you want to go a step further, you can use service workers.

Service Workers are an API that most modern browsers support, and to simplify, they act like a proxy in the browser. You can register a service worker in your application and every GET request will then go through it, allowing you to decide if you really want to fetch the requested resource, or if you want to serve it from cache. You can then cache everything, even your `index.html`, which guarantees the fastest startup time (no request to the server).

You may be wondering how a new version can be deployed if everything is cached, but you're covered: the service worker will serve from cache and then check if a new version is available. It

can then force the refresh, or ask the users if they want it immediately or later.

It even allows you to go offline, as everything is cached!

Angular offers a dedicated package called `@angular/service-worker`, which is relatively easy to setup, and can help you transform your Angular application into a Progressive Web App (PWA).

## 27.3. Profiling

Now that we have talked about first load and reload, we can start talking about runtime performances. But if you run into a performance issue, before trying any of the following tips, you should start by measuring and profiling the application.

Browsers nowadays offer nice developer tools, especially Chrome, which allows you to record your application, and analyze its behavior, the function call hierarchy, the time spent by each function, etc. You can also simulate some conditions, like using a slower processor, or using a 3G network.

In addition to the native browser developer tools, You can install the official Angular DevTools browser extension. It has profiling capabilities allowing for example to analyze its change detection.

But Angular also offers a precious, little-known tool: `ng.profiler`. It allows you to measure how long a change detection takes in the current page. Profiling the change detection is important when your application relies on the brute-force change detection mechanism which consists in evaluating all the expressions of all the components in the tree. Hopefully, when all your components rely on signals to signal a change and ZoneJS isn't used anymore, the time taken by change detection shouldn't be much of a concern anymore.

But before we get there, you can try to apply one of the tips which we'll look at, and measure again to see if there is any improvement.

In your `main.ts` file, replace the application bootstrapping code with the following:

```
bootstrapApplication(App, appConfig)
  .then(applicationRef => {
    const componentRef = applicationRef.components[0];
    // allows to run `ng.profiler.timeChangeDetection();`
    enableDebugTools(componentRef);
  })
  .catch(err => console.log(err));
```

Then go to the page you want to profile, open your browser console, and execute the following instruction:

```
> ng.profiler.timeChangeDetection()
ran 489 change detection cycles
1.02 ms per check
```

The Angular team recommends having a time per check below 3ms, to leave enough time for the application logic, the UI updates and the browser's rendering pipeline to fit within a 16ms frame (assuming a 60 FPS target frame rate).

When optimizing your code, using the techniques that we will explain in the following pages, it can help to use these tools to measure before and after, and see if it was actually worth it.

Let's discover these tips!

## 27.4. Runtime performances

Angular's magic relies on its change detection mechanism: the framework automatically detects changes in the state of the application and updates the DOM accordingly. So, as a general rule of thumb, you'll want to help Angular and limit the change detection triggering and the amount of DOM to update/create/delete.

Read the chapter about ZoneJS if you want a recap on how the Angular change detection works.

To be honest, most applications will be fine, even when displaying a lot of stuff.

But some of us will have to recode Excel in the browser for their enterprise, or will have a component with a tree displaying 10,000 customers, or another unreasonable thing to do in a browser. These things are tricky, whatever framework you use. They tend to update a lot of DOM, and have to check a lot of components. A few of the following tricks can help.

## 27.5. Production mode

While you're developing (using `ng serve` to serve your application), the CLI sets an internal variable called `ngDevMode` that the framework uses to know if it needs to do some extra work or not.

In this mode, error messages are more detailed. The generated DOM contains additional attributes useful for debugging. And more importantly, every change detection does two traversals of the component tree instead on just one. The second traversal ensures that all the expressions return the same thing as in the first traversal, and throws an error otherwise. This makes sure that you respect the one-way data flow principle.

Make sure to detect and fix these errors, because the production build (that you obtain when executing `ng build`) won't do these additional traversals, which will thus make the application faster. Such a problem will thus not cause any error being logged in the console.

## 27.6. `track` in for loops

If you use the control flow syntax, you know that `track` is mandatory in `@for` loops. To understand why, let me explain how modern JS frameworks (at least all major ones) handle collections. When you have a collection of 3 ponies and want to display them in a list, you'll write something like:

```
<ul>
  @for (pony of ponies(); track pony) {
    <li>{{ pony.name }}</li>
  }
</ul>
```

When you add a new pony, Angular will add a DOM node in the proper position. If you update the name of one of the ponies, Angular will change just the text content of the right `li`.

How does it do that? By keeping track of which DOM node references which object reference. Angular will have an internal representation looking like:

```
node li 1 -> pony #e435 // { id: 3, color: blue }
node li 2 -> pony #8fa4 // { id: 4, color: red }
```

It works great, and if you change an object for another one, Angular will destroy the node and build another one.

```
node li 1 (recreated) -> pony #c1ea // { id: 1, color: green }
node li 2 -> pony #8fa4 // { id: 4, color: red }
```

If the whole collection is updated with new objects, the complete DOM list will be destroyed and recreated. Which is fine, except when you just refresh a list with almost the same content: in that case, Angular destroys the complete node list and recreates it, even if there is no need to. For example, when you fetch the same results from the server, you will have the same content, but different references as your collection will have been recreated.

The solution for this use-case is to help Angular track the objects, not by their references, but by something that you know will uniquely identify the object, typically an ID:

```
<ul>
  @for (pony of ponies(); track pony.id) {
    <li>{{ pony.name }}</li>
  }
</ul>
```

With this `track` expression, Angular will only recreate a DOM node if the ID of the pony changes. On a very big list which doesn't change much, it can save a ton of DOM deletions/creations. Since this optimization technique is quite cheap to implement, Angular decided to force you to specify a track expression in `@for` loops. It isn't mandatory when the `*ngFor` directive is used, but a similar optimization exists: `trackBy`.

`track` is also necessary if you want to use animations. If a DOM element's style is supposed to be animated (by transitioning smoothly from the previous value to the new one), and the list of ponies is replaced by a new one when refreshed, then a correct `track` is a must: with a track by identity, the

animation will never happen, because the style of the element never changes. Instead, it's the element itself which is being replaced by Angular.

## 27.7. Change detection strategies

When we explained how Angular detects the changes in your application, we showed the tree of components and said that Angular starts by checking the root component, then its children, then its grand-children, until all components are checked. Then all the necessary DOM updates are applied in one batch.

But you may be wondering if it is a very good idea to check **every** component on **every** change. And you're right, that's often not really necessary.

Angular offers another change detection strategy: it's called `OnPush` and it can be defined on any component.

With this strategy, the template of the component will only be checked in 3 cases:

- one of the inputs of the component changed (to be more accurate, when the **reference** of one of the inputs changes);
- an event handler of the component was triggered;
- the value of a signal read by the template of the component changed.

This can be very convenient when the template of a component only depends on its inputs, and can give a serious boost to your application if you display a lot of components on screen! But once again, be very cautious before applying this optimization: if the preconditions end up not being respected, you will lose your hair wondering why the component (or any of its descendants) isn't always repainting itself after a change.

Let's take a small example to demonstrate.

Imagine that we have 3 components. A very simple `Image`:

```
@Component({
  selector: 'ns-img',
  template: `
    <p>{{ check() }}</p>
    <img [src]="src()" />
  `
})
export class Image {
  readonly src = input.required<string>();

  protected check(): void {
    console.log('image component view checked');
  }
}
```

used in a Pony:

```
@Component({
  selector: 'ns-pony',
  template: `
    <p>{{ check() }}</p>
    <ns-img [src]="ponyImageUrl()" />
  `,
  imports: [Image]
})
export class Pony {
  readonly ponyModel = input.required<PonyModel>();
  protected readonly ponyImageUrl = computed(() => `images/pony-${this.ponyModel
().color}-running.gif`);
  protected check(): void {
    console.log('pony component view checked');
  }
}
```

used itself in a Race:

```
@Component({
  selector: 'ns-race',
  template: `
    <h2>Race</h2>
    <p>{{ check() }}</p>
    @for (pony of ponies(); track pony.id) {
      <div>
        <ns-pony [ponyModel]="pony" />
      </div>
    }
    <button (click)="changeColor()">Change color</button>
  `,
  imports: [Pony]
})
export class Race {
  protected readonly ponies = signal<Array<PonyModel>>([
    { id: 1, color: 'green' },
    { id: 2, color: 'orange' }
  ]);
  protected readonly colors: Array<string> = ['green', 'orange', 'blue'];

  protected check(): void {
    console.log('race component view checked');
  }

  protected changeColor(): void {
    const ponies = this.ponies();
    ponies[0].color = this.randomColor();
```

```
    this.ponies.set([...ponies]);
  }

}
```

The `Race` displays two ponies, and the user can change the color of the first one by clicking on the `Change color` button.

With the current default change detection strategy, every time that we have a change in the application, all 3 components are checked.

We added a `check()` method in each component, called in each template: it allows us to track if the component is checked or not. And indeed in our example, we can see in our console:

```
pony component view checked
image component view checked
pony component view checked
image component view checked
race component view checked
```

(we can see that twice actually, because we are in development mode - see the section about the production mode above).

### 27.7.1. OnPush

But in this case, it's a waste of time: we know that if the pony doesn't change, the template of the `Pony` doesn't need to be checked. Same thing for the `Image`: if the `src` input is the same, there is no need to recompute the image URL. So let's switch these components to `OnPush`, by adding a `changeDetection` attribute in their `@Component` decorator:

```
@Component({
  selector: 'ns-img',
  template: `
    <p>{{ check() }}</p>
    <img [src]="src()" />
  `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class Image {
  readonly src = input.required<string>();

  protected check(): void {
    console.log('image component view checked');
  }
}
```

```
@Component({
```

```
  selector: 'ns-pony',
  template: `
    <p>{{ check() }}</p>
    <ns-img [src]="ponyImageUrl()" />
  `,
  imports: [Image],
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class Pony {
  readonly ponyModel = input.required<PonyModel>();
  protected readonly ponyImageUrl = computed(() => `images/pony-${this.ponyModel
().color}-running.gif`);

  protected check(): void {
    console.log('pony component view checked');
  }
}
```

When we click to change the color, we will only see in the console:

```
race component view checked
```

Which is awesome, because it means that we don't check the components that we don't need to check \o/.

### 27.7.2. OnPush and the mutability trap

But... there is a slight problem: the pony's color doesn't change anymore!

I picked this example on purpose: even if `OnPush` is really powerful, it can be tricky. Optimizing existing components is not only about adding a few `OnPush` here and there.

Why doesn't it work in our case?

Take a closer look at our `Race`, and its `changeColor` method:

```
protected changeColor(): void {
  const ponies = this.ponies();
  ponies[0].color = this.randomColor();
  this.ponies.set([...ponies]);
}
```

This method **mutates** the pony in the `ponies` collection, and this pony is the input of our `Pony`. Now that we shifted our component to be `OnPush`, Angular will only run the change detection if the **reference** of the `pony` input changes. And when you mutate an object, it's still the same object, so the reference doesn't change, and Angular thinks there is no need to run the change detection...

So, is this change detection strategy completely useless? Not really, but it does require you to be

more careful.

The simple way to fix our issue is to not mutate our pony in `changeColor`, but to create a new object:

```
protected changeColor(): void {
  const ponies = this.ponies();
  const pony = ponies[0];
  // create a new pony with the old attributes and the new color
  ponies[0] = { ...pony, color: this.randomColor() };
  this.ponies.set([...ponies]);
}
```

Once you've done that, the application is faster **and** correct. If the user clicks on the button, the `changeColor` method creates a new pony object with the old attributes and the new color. As this is a new object, Angular will run the change detection in the `Pony` (an input changed), and then the `src` input of the `Image` will also change, and the image will display the correct color. And, of course, if another event triggers the change detection in `Race`, the children component will not be checked (if their inputs did not change).

As you can see, you can quickly fall into a trap when migrating a component to an `OnPush` strategy, so be careful (unit tests are your friend).

There is a last topic we need to talk about: observables.

### 27.7.3. OnPush, Observables, signals, and the async pipe

Let's say we now have only one component, our well-known `Pony`. It subscribes to an observable from a `ColorService` that returns a new color every second. We obviously expect the image to change every second. The developer of this component thought that an `OnPush` change detection strategy couldn't hurt. What do you think?

```
@Component({
  selector: 'ns-pony',
  template: `
    <p>New color every 1s</p>
    @if (color) {
      <img [src]="'pony-' + color + '.gif'" [alt]="color" />
    }
  `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class Pony {
  protected color: string | undefined;

  constructor() {
    inject(ColorService)
      .get()
      .pipe(takeUntilDestroyed())
      .subscribe(color => (this.color = color));
```

```
  }
}
```

Sadly, this doesn't work. With the `OnPush` strategy, Angular only refreshes the template if one of the inputs changed (here, there is no input), or if an event was triggered (there is none either), or if a signal read by the template changed (no signal is used here). So the `color` field is updated every second, but the template is never refreshed…

This can be fixed by turning the `color` property into a signal and setting the signal value when the observable emits a new color. But there's an even simpler solution: using `toSignal` to transform the observable into a signal:

```
@Component({
  selector: 'ns-observable-on-push-with-signal',
  template: `@if (color(); as c) {
    <img [src]="'pony-' + c + '.gif'" [alt]="c" />
  }`,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class Pony {
  protected readonly color: Signal<string | undefined> = toSignal(inject(
ColorService).get());
}
```

Before signals were introduced, the solution to this problem was to use the `async` pipe to let the template subscribe (and unsubscribe) for you.

```
@Component({
  selector: 'ns-observable-on-push-with-async',
  template: `@if (color$ | async; as c) {
    <img [src]="'pony-' + c + '.gif'" [alt]="c" />
  }`,
  imports: [AsyncPipe],
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class Pony {
  protected color$: Observable<string> = inject(ColorService).get();
}
```

When a new value is received from the observable, the `async` pipe marks the component to be checked during the next change detection.

Note that `async` can lead to several HTTP requests if used several times in a template, It also makes it more difficult to access the color from inside the TypeScript code. So our preference goes to the `toSignal` solution, which has all the advantages of `async` without its disadvantages. Beware that `toSignal()` is still in developer preview though. But that won't last very long.

Those two examples show how setting the change detection strategy to `OnPush` shouldn't be done

lightly, especially with legacy components that don't use signals. However, `OnPush` components behave almost the same way now as all components will behave when ZoneJS won't be used anymore for change detection. So if you want to prepare for this zoneless future, you should consider, at least for new components, to use signals for their state and to use the `OnPush` strategy.

## 27.8. Get out of the zone

There is another way to avoid useless change detections: you can completely run some code outside of Zone.js. To do so, you can inject `NgZone`, and then use its `runOutsideAngular` method to execute code outside its watch.

This can be really useful to wrap something that handles a lot of events, typically something coming from a third-party library. For example, let's imagine you have a chart in your application, built with Chart.js. A user hovering over the chart will trigger hundreds of mouse events. The fact that hundreds of events are fired isn't really a problem. That also happens with any DOM element. What is a problem is that the library calls `addEventListener()` to handle these events, for example to display tooltips, and that's what triggers hundreds of change detections!

```
readonly canvas = viewChild.required<ElementRef<HTMLCanvasElement>>('chart');

constructor() {
  effect(() => {
    const ctx = this.canvas().nativeElement;
    new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Green', 'Red'],
        datasets: [{ label: 'Score', data: [12, 21] }]
      }
    });
  });
}
```

In that case, you can inject `NgZone` and use `runOutsideAngular` to ignore the events from Chart.js:

```
readonly canvas = viewChild.required<ElementRef<HTMLCanvasElement>>('chart');

constructor() {
  const ngZone = inject(NgZone);
  effect(() => {
    const ctx = this.canvas().nativeElement;
    ngZone.runOutsideAngular(() => {
      new Chart(ctx, {
        type: 'bar',
        data: {
          labels: ['Green', 'Red'],
          datasets: [{ label: 'Score', data: [12, 21] }]
        }
```

```
      });
    });
  });
}
```

This produces the same results, and the rest of the component would still be checked automatically by Angular, but we no longer trigger change detection when using the chart.

Having to do this to avoid performance problems when using Angular is one of the big problems caused by the usage of ZoneJS. It's one of the reasons signals have been introduced and Angular wants to move towards zoneless change detection.

## 27.9. Zoneless change detection

Since version 18, Angular uses a new `ChangeDetectionScheduler` to trigger change detections, or *synchronizations* as the Angular team now calls this process. This new scheduler no longer relies solely on ZoneJS to know when to synchronize the DOM with the application state. It's also triggered when a signal read by a template gets a new value, when a template or host listener is called or when an async pipe receives a new value (and on some other occasions that we won't mention here).

Thanks to this new scheduler, we can choose to go *zoneless*, i.e. to completely remove ZoneJS from our applications. This zoneless synchronization is experimental in v18 and v19, and is in developer preview in v20. We strongly encourage you to adopt it if you start a new project, because that's the way of the future.

To go zoneless, you will need to add `provideZonelessChangeDetection()` to the application providers, and to remove zone.js from the polyfills (inside the `angular.json` file).

There's a big catch though: your code, and the code of all the Angular libraries that you use, need to be ready for zoneless. This means that:

- all the components must properly use signals (or the async pipe) to manage their state;
- the code must not use some methods of NgZone like `NgZone.onStable`.

The good news is that most popular Angular component libraries have of course anticipated this change and are now ready for zoneless. All the instructions and examples in this book are zoneless-compatible too.

If all your components use `OnPush` and your application works fine, it's a good sign that your code is already ready for zoneless synchronization.

## 27.10. Pure pipes

As you know, you can build your own pipes to format and display your data. For example, to display the full name of a user, you can either write a method in your component:

```
@Component({
```

```
  selector: 'ns-menu',
  template: `
    <p>{{ userName() }}</p>
    <p>...</p>
    <p>{{ userName() }}</p>
  `
})
export class Menu {
  protected readonly user = signal<UserModel>({
    id: 1001,
    firstName: 'Jane',
    lastName: 'Doe',
    title: 'Miss'
  });

  protected userName(): string {
    return `${this.user().title} ${this.user().firstName} ${this.user().lastName}`;
  }
}
```

or better, define a computed signal:

```
@Component({
  selector: 'ns-menu',
  template: `
    <p>{{ userName() }}</p>
    <p>...</p>
    <p>{{ userName() }}</p>
  `
})
export class Menu {
  protected readonly user = signal<UserModel>({
    id: 1001,
    firstName: 'Jane',
    lastName: 'Doe',
    title: 'Miss'
  });
  protected readonly userName = computed(() => `${this.user().title} ${this.user
().firstName} ${this.user().lastName}`);
}
```

The computed version is more efficient, because the computation is memoized instead of being executed each time the expression is evaluated in the template.

This is fine when you have a signal containing a single user. But if you need have an array of objects each having a user as a property, it can be tedious to transform all these users into user names inside a computed.

A custom pipe is an easier solution to encapsulate this logic:

```
@Component({
  selector: 'ns-menu',
  template: `
    <p>{{ user | displayName }}</p>
    <p>...</p>
    <p>{{ user | displayName }}</p>
  `,
  imports: [DisplayNamePipe]
})
export class Menu {
  protected readonly user: UserModel = {
    id: 1001,
    firstName: 'Jane',
    lastName: 'Doe',
    title: 'Miss'
  };
}
```

```
@Pipe({
  name: 'displayName'
})
export class DisplayNamePipe implements PipeTransform {
  transform(user: UserModel): string {
    return `${user.title} ${user.firstName} ${user.lastName}`;
  }
}
```

This takes a little bit more work, but writing a pipe allows you to reuse it in any template.

What you may not know is that using a pipe also offers the same performance benefit as using a computed signal. By default, a pipe is "pure". In computer science, we call "pure" a function that has no side effect, and whose result only depends on its entries. A pure pipe is pretty much the same: the result of its `transform` method only depends on arguments. Knowing that, Angular applies the same optimization as for the computed signal: it memoizes the transformation. The `transform` method is only called if the reference of the value it transforms changes or if one of the other arguments changes (yes, a bit like the `OnPush` strategy for components).

By default, a custom pipe is pure, so that's great! But sometimes it's not a right fit.

In my example, if we mutate the user to set its `firstName` to a different value, the pipe never refreshes... It's pretty much the same issue that we had with the `OnPush` strategy: the reference of the value doesn't change, so the pipe does not run again.

Here you have two solutions:

- carefully use the pipe with immutable objects (do not mutate the user - create a new user with the new `firstName`);
- mark the pipe as "impure", and Angular will run it every time. You lose a tiny bit in

performance, but you are sure that the displayed value is refreshed.

To mark a pipe as impure, just add `pure: false` in its decorator:

```
@Pipe({
  name: 'displayName',
  pure: false
})
export class DisplayNameImpurePipe implements PipeTransform {
  transform(user: UserModel): string {
    return `${user.title} ${user.firstName} ${user.lastName}`;
  }
}
```

To sum up:

- a pure pipe is not called as often as a method in a component
- but it doesn't run again if the input value is mutated, so use it carefully.

## 27.11. Conclusion

This chapter hopefully taught you some techniques which can help solve performance problems. But remember the golden rules of performance optimization:

- don't
- don't... yet
- profile before optimizing.

As a famous computer scientist said:

> premature optimization is the root of all evil.
>
> — Donald Knuth

So strive to make the code as simple and correct and readable as possible, and only start thinking about profiling, then optimizing, if you have a proven performance problem.

For new components, relying on signals makes it easy to apply the `OnPush` strategy, and to be ready for zoneless change detection.

Try our exercise Performance 🦄! You will optimize our application and be able to measure every step of progress! We also have an exercise to switch to Zoneless 🦄!

# Chapter 28. Signals: advanced topics

## 28.1. Value equality

Signals are used to hold state, and you can react to their change. But how do they decide that they have changed? Calling `set` or `update` on a signal doesn't necessarily make it change: if you set a value that is *equal* to the value it already has, then the signal will consider that it hasn't changed.

This *equality* of two values, by default, is decided by calling `Object.is`. For objects, if the new value is `===` to the current value, no change happens. But you can define a custom equality in case referential equality is not what you want.

```
const status = signal(
  {
    dirty: false,
    touched: false
  },
  {
    equal: (previousStatus, newStatus) =>
      previousStatus.dirty === newStatus.dirty && previousStatus.touched ===
newStatus.touched
  }
);
// errorDisplayed will not re-evaluated if both values stay the same
const errorDisplayed = computed(() => status().dirty || status().touched);
```

## 28.2. `untracked`

`computed` and `effect` both take a function as argument. Angular calls this function at least once. This allows computing the initial value of the computed signal, or triggering the initial side-effect. But it's also necessary for Angular to know which signals are read inside the function, in order to update the dependency graph.

For example,

```
readonly price = signal(42);
readonly quantity = signal(2);

readonly total = computed(() => this.price() * this.quantity());
```

creates a dependency between `total` and the two other signals: when `price` or `quantity` change, Angular knows that `total` must be recomputed.

The dependencies on the above example are obvious. But that's not always the case, especially in effects.

Let's take the following example:

```
protected readonly visitedUrl = signal<string | undefined>(undefined);

constructor() {
  const analyticsService = inject(AnalyticsService);
  effect(() => {
    const url = this.visitedUrl();
    if (url) {
      analyticsService.sendHitForUrl(url).subscribe();
    }
  });
}
```

It sends a new hit, using HTTP, every time the `visitedUrl` signal changes. The dependency looks obvious: this effect only depends on `visitedUrl`. But does it, really?

If an interceptor has been configured for the HTTP client, and if this interceptor reads a `token` signal for example, then subscribing to the observable inside the effect indirectly reads the `token` signal. The effect is thus also dependent on the `token` signal, and a new hit will thus also be sent when the token changes.

To avoid such undesired dependencies, use `untracked`. A good rule of thumb to avoid surprises is to start by reading the signals you really want to depend on, and then run the code inside an `untracked` block:

```
protected readonly visitedUrl = signal<string | undefined>(undefined);

constructor() {
  const analyticsService = inject(AnalyticsService);
  effect(() => {
    const url = this.visitedUrl();
    untracked(() => {
      if (url) {
        analyticsService.sendHitForUrl(url).subscribe();
      }
    });
  });
}
```

## 28.3. Root and component effects

We learnt that `effect` needs an injection context. An effect can be created while constructing a component, directive, or service. Otherwise, an `Injector` must be passed as option.

But all effects aren't handled the same way. Angular distinguishes two kinds of effects:

- component effects, which are created using the injector of a component or directive;

- root effects, which are created in root services, or with the `forceRoot` option.

Component effects run just before the change detection of their owning component. They are destroyed when their owning component is destroyed.

Root effects on the other hand are never destroyed (since their owning service is never destroyed). They run as a micro task (i.e. at the same time a resolved promise would run). In unit tests, you must call `TestBed.tick()` to make them run.

If you want an effect, whatever its type, to stop running at a certain point, you're always free to destroy it yourself, by calling the `destroy` method of the `EffectRef` returned by `effect`.

## 28.4. `afterRenderEffect`

Since regular component effects run before the change detection, they're not a good fit to read or modify the DOM of the component: the DOM has not been rendered yet by Angular when the effect is run. If you want to read or modify the DOM inside an effect, you can use a different kind of effect, created by the function `afterRenderEffect`. Such an effect, as its name indicates, runs after the rendering.

## 28.5. Effect cleanup

Effects can also receive a cleanup function. This function is run when the effect runs again. This can be handy when you need to cancel a previous action before starting a new one. In the example below, we start an interval that runs every `count` seconds, and we want to stop it and start a new one when the count changes:

```
this.intervalEffect = effect(onCleanup => {
  const intervalId = setInterval(() => console.log(`count in intervalEffect ${this
.count()}`), this.count() * 1000);
  return onCleanup(() => clearInterval(intervalId));
});
```

## 28.6. Two-way binding with `model` inputs

Signals also allow a fresh take on existing patterns. As you probably know, Angular allows a "banana in a box" syntax for two-way binding. This is mostly used with `ngModel` to bind a form control to a component property:

*login.html*

```
<input name="login" [(ngModel)]="user.login" />
```

Under the hood, this is because the `ngModel` directive has a `ngModel` input and a `ngModelChange` output.

So the banana in a box syntax is just syntactic sugar for the following:

*login.html*

```
<input name="login" [ngModel]="user.login" (ngModelChange)="user.login = $event" />
```

The syntax is, in fact, general and can be used with any component or directive that has an input named `something` and an output named `somethingChange`.

You can leverage this in your own components and directives, for example to build a pagination component:

*pagination.ts*

```
readonly collectionSize = input.required<number>();
readonly pageSize = input.required<number>();
readonly page = input.required<number>();
readonly pageChange = output<number>();

protected readonly pages = computed(() => this.computePages());

protected goToPage(page: number) {
  this.pageChange.emit(page);
}

private computePages() {
  return Array.from({ length: Math.ceil(this.collectionSize() / this.pageSize()) }, (
_, i) => i + 1);
}
```

The component receives the collection, the page size, and the current page as inputs, and emits the new page when the user clicks on a button.

Every time an input changes, the component recomputes the buttons to display. The template uses a for loop to display the buttons:

*pagination.html*

```
@for (pageNumber of pages(); track pageNumber) {
  <button [class.active]="page() === pageNumber" (click)="goToPage(pageNumber)">
    {{ pageNumber }}
  </button>
}
```

The component can then be used like:

*Usage*

```
<ns-pagination [(page)]="page" [collectionSize]="collectionSize()"
```

```
[pageSize]="pageSize()" />
```

In the parent component, `page` can be a `number`, but it can also be a `WritableSignal<number>`. In the latter case, the framework will automatically pass the value of the signal as input to the pagination component, and will set the signal value to the new page when the pagination component emits one.

The pagination component can be rewritten using the `model()` function:

*pagination.ts*

```
readonly collectionSize = input.required<number>();
readonly pageSize = input.required<number>();
protected readonly pages = computed(() => this.computePages());

readonly page = model.required<number>();
// ^? ModelSignal<number>;
protected goToPage(page: number) {
  this.page.set(page);
}

private computePages() {
  return Array.from({ length: Math.ceil(this.collectionSize() / this.pageSize()) }, (
_, i) => i + 1);
}
```

As you can see, `model()` is used to define the input/output pair, and the output emission is done using the `set()` method of the signal (`ModelSignal` extends both `WritableSignal` and `OutputRef`).

A `model` can be required, or can have a default value, or can be aliased, as we saw for inputs. It can't be transformed though. If you use an alias, the output will be aliased as well.

## 28.7. Linked signals with `linkedSignal`

Angular v19 introduced a new (developer preview) concept called "linked signals". A linked signal is a *writable* signal, but it is also a *computed* signal, as its content can be reset thanks to a computation that depends on another signal (or several ones).

Imagine we have a component that displays a list of items received via an *input*, and we want our users to select one of them. By default, let's say we want to select the first item of the list. But every time the list of items changes, the selected item may no longer be valid, so we want to reset the selected item to the first one.

We can imagine a component like this:

```
export class ItemList {
  readonly items = input.required<Array<ItemModel>>();
  protected readonly selectedItem = signal<ItemModel | undefined>(undefined);
```

```
  protected pickItem(item: ItemModel) {
    this.selectedItem.set(item);
  }
}
```

Using an effect may come to mind to solve the selection problem:

```
constructor() {
  // ⚠️ This is not recommended
  effect(() => {
    this.selectedItem.set(this.items()[0]);
  });
}
```

Every time the list of items changes, the effect will be triggered and the first item will be selected. This works, but using effects is generally not recommended, except for some specific cases, like synchronizing something outside the application (like the local storage for example).

There is a nice trick that I can show you before we dive into the now-recommended solution: we can use a `computed` value that returns… a signal!

```
export class ItemList {
  readonly items = input.required<Array<ItemModel>>();
  protected readonly selectedItem = computed<WritableSignal<ItemModel | undefined>>(()
=> signal(this.items()[0]));

  protected pickItem(item: ItemModel) {
    this.selectedItem().set(item);
  }
}
```

As you can see, the `computed` value returns a signal that represents the selected item (whereas they usually return a value directly). Every time the list of items changes, the computed function is re-evaluated, and returns a new signal that represents the selected item. The downside of this solution is that we have to use `selectedItem()()` to read the value, or `selectedItem().set()` to update it, which is a bit ugly.

This is where we can use a `linkedSignal`:

```
export class ItemList {
  readonly items = input.required<Array<ItemModel>>();
  // ✅ This is recommended
  protected readonly selectedItem: WritableSignal<ItemModel> = linkedSignal(() =>
this.items()[0]);
```

A `linkedSignal` is a `WritableSignal`, but its value can be reset thanks to a computation. If the `items` change, then the computation will be re-executed and the value of the signal will be updated with the result.

This is a powerful concept, as it allows us to define a signal that depends on another signal, like a computed, but with the ability to write to it (a sort of "writable computed").

The computation can of course depend on several signals. Here `selectedItem` is reset when the `items` input changes, but also when the `enabled` input changes.

```
export class ItemList {
  readonly items = input.required<Array<ItemModel>>();
  readonly enabled = input.required<boolean>();
  // recomputes if `enabled` or `items` change
  protected readonly selectedItem = linkedSignal(() => (this.enabled() ? this.items
()[0] : undefined));
```

Note that you can use the previous value of the source signal in the computation function if you need to. For example, if you want to access the previous `items` value to compare it with the new one, you can declare the `linkedSignal` with the `source` and `computation` options. In that case, the computation function receives the current and previous values of the source as parameters.

```
export class ItemList {
  readonly items = input.required<Array<ItemModel>>();
  protected readonly selectedItem = linkedSignal</* source */ Array<ItemModel>, /*
value */ ItemModel>({
    source: this.items,
    computation: (items, previous) => {
      // pick the item the user selected if it's still in the new item list
      if (previous !== undefined) {
        const previousChoice = previous.value; // previous.source contains the
previous items
        if (items.map(item => item.name).includes(previousChoice.name)) {
          return previousChoice;
        }
      }
      return items[0];
    }
  });
```

You can also define a custom equality function to decide if the signal has changed with the `equal` option.

## 28.8. Async resources with `resource` and `rxResource`

In Angular v19, a new feature was added to help with asynchronous operations. Most applications need to fetch data from a server, depending on some parameters, and display the result in the UI:

`resource` aims to help with that.

This API is experimental, and will go through an RFC process soon: I would not advise you to use it yet.

The `resource()` function allows you to define a resource that represents an asynchronous operation. The function takes an object with a mandatory `loader` function that returns a promise:

```
list(): ResourceRef<Array<UserModel> | undefined> {
  return resource({
    loader: async () => {
      const response = await fetch('/users');
      return (await response.json()) as Array<UserModel>;
    }
  });
}
```

This example doesn't use the HTTP client, but the native `fetch()` function, which returns a promise. Indeed, the `resource()` function is not linked to RxJS, and can thus use any client that returns promises. `rxResource`, that we will discuss in a few seconds, is the alternative to resource that can be used with an Observable-based client. This is another example of Angular decoupling itself from RxJS, but still providing interoperability functions allowing you to use it smoothly.

You can also define a `defaultValue` option that will be used as the initial value of the resource (instead of `undefined` by default).

```
return resource({
  defaultValue: [],
  loader: async () => {
    const response = await fetch('/users');
    return (await response.json()) as Array<UserModel>;
  }
});
```

`resource()` returns a `ResourceRef`, an object containing:

- an `isLoading` signal that indicates if the resource is loading;
- a `value` signal that contains the result of the promise;
- an `error` signal that contains the error if the promise is rejected;
- a `status` signal that contains the status of the resource.

You can then use these signals in your template:

```
@if (usersResource.isLoading()) {
  <p>Loading...</p>
```

```
} @else {
  <ul>
    @for (user of usersResource.value(); track user.id) {
      <li>{{ user.name }}</li>
    }
  </ul>
}
```

The `status` signal can be:

- `'idle'`, the initial state;
- `'loading'`, when the promise is pending;
- `'error'`, when the promise is rejected;
- `'resolved'`, when the promise is resolved;
- `'reloading'`, when the resource is reloading;
- `'local'`, when the value is set locally.

The resource also has a `reload` method that allows you to reload the resource. In that case, its status will be set to `'reloading'`.

But the reloading can also be automatic, thanks to the `params` option. When provided, the resource will automatically reload if one of the signals used in the `params` changes. Here, for example, the component has a `sortOrder` option that is used in the request:

```
protected readonly sortOrder = signal<'asc' | 'desc'>('asc');
protected readonly usersResource = resource({
  // 👇 The `sortOrder` signal is used to trigger a reload
  params: () => ({ sort: this.sortOrder() }),
  loader: async loaderParams => {
    // 👇 loaderParams also contains the `abortSignal` to cancel the request
    // and the previous status of the resource
    // here we are only interested in the params
    const params = loaderParams.params;
    const response = await fetch(`/users?sort=${params.sort}`);
    return (await response.json()) as Array<UserModel>;
  }
});
```

If the `sortOrder` signal changes, the resource will automatically reload! You can also cancel the previous request if needed when the resource is reloaded using the `abortSignal` parameter of the loader (for example to implement a debounce). You can choose to ignore the reload request and thus keep the current value by returning `undefined` from the `params` function.

Last but not least, the returned `ResourceRef` is in fact writable. You can use its `set` or `update` methods to change the value of the resource (on the `value` or on the resource itself, both work). In that case, its status will be set to `'local'`. If you're only interested in reading the resource, you can use the

`asReadonly` method to get a read-only version of the resource.

Finally, the `ResourceRef` has a `destroy` method that can be used to stop the resource.

Angular v19.2 added the possibility to create resources with streamed response data. A streaming resource is defined with a `stream` option instead of a `loader` option. This `stream` function returns a promise of a signal (yes, I needed to read it twice as well). The signal value must be of type `ResourceStreamItem`: an object with a `value` or an `error` property. When the promise is resolved, the loader can continue to update that signal over time, and the resource will update its value and error every time the signal's item changes.

You can build this stream yourself, using a WebSocket for example. We can also imagine that some libraries such as Firebase could provide a stream function that would be directly usable:

```
list(): ResourceRef<Array<UserModel> | undefined> {
  return resource({
    // firebaseCollection does not exist in real-life
    stream: async ({ abortSignal }) => await firebaseCollection('users', abortSignal)
  });
}
```

Now, let's see how we can use an observable-based resource instead of a promised-based one.

You can use the `rxResource()` function in that case. This function is really similar to `resource()`, but its stream must return an observable instead of a promise. This allows you to use our good old `HttpClient` service to fetch data from a server, using all your interceptors, error handling, etc:

```
protected readonly sortOrder = signal<'asc' | 'desc'>('asc');
protected readonly usersResource = rxResource({
  params: () => ({ sort: this.sortOrder() }),
  // 👇 RxJS powered loader
  stream: ({ params }) => this.httpClient.get<Array<UserModel>>('/users', { params: {
sort: params.sort } })
});
```

Note that the `rxResource()` function is from the `@angular/core/rxjs-interop` package, where the `resource()` function is from `@angular/core`. You can have a stream of values by returning an observable that emits several times, and the resource will be updated every time a new value is emitted:

```
protected readonly sortOrder = signal<'asc' | 'desc'>('asc');
protected readonly usersResource = rxResource({
  params: () => ({ sort: this.sortOrder() }),
  // 👇 stream that fetches the value now and every 10s
  stream: ({ params }) =>
    timer(0, 10000).pipe(
      switchMap(() => this.httpClient.get<Array<UserModel>>('/users', { params: {
```

```
sort: params.sort } }))
    )
});
```

## 28.9. HTTP calls with `httpResource`

Angular v19.2 introduced a dedicated (and experimental) function to create resources that use HTTP requests: `httpResource()` in the `@angular/common/http` package.

This function uses `HttpClient` under the hood, allowing us to use our usual interceptors, testing utilities, etc.

The most basic usage is to call this function with a function that returns the URL from which you want to fetch data:

```
protected readonly usersResource = httpResource<Array<UserModel>>(() => '/users');
```

`httpResource()` returns an `HttpResourceRef` with the same properties as `ResourceRef`, the type returned by `resource()`, as it is built on top of it:

- `value` is a signal that contains the deserialized JSON response body;
- `status` is a signal that contains the resource status (idle, loading, error, resolved, etc.);
- `error` is a signal that contains the error if the request fails;
- `isLoading` is a signal that indicates if the resource is loading;
- `reload()` is a method that allows you to reload the resource;
- `update()` and `set()` are methods that allow you to change the value of the resource;
- `asReadonly()` is a method that allows you to get a read-only version of the resource;
- `hasValue()` is a method that allows you to know if the resource has a value;
- `destroy()` is a method that allows you to stop the resource.

It also contains a few more properties specific to HTTP resources:

- `statusCode` is a signal that contains the status code of the response as a number;
- `headers` is a signal that contains the headers of the response as `HttpHeaders`;
- `progress` is a signal that contains the download progress of the response as a `HttpProgressEvent`;

It is also possible to define a reactive resource by using a signal in the function that defines the URL. The resource will automatically reload when the signal changes:

```
protected readonly sortOrder = signal<'asc' | 'desc'>('asc');
protected readonly sortedUsersResource = httpResource<Array<UserModel>>(() =>
`/users?sort=${this.sortOrder()}`);
```

When using a reactive request, the resource will automatically reload when a signal used in the request changes. If you want to skip the reload, you can return `undefined` from the request function (as for `resource`).

If you need more fine-grained control over the request, you can also pass a function that returns an `HttpResourceRequest` object to the `httpResource()` function.

This object must have a `url` property and can have other options like `method` (`GET` by default), `params`, `headers`, `reportProgress`, etc. If you want to make the request reactive, you can use signals in the `url`, `params` or `headers` properties.

The above example would then look like:

```
protected readonly sortedUsersResource = httpResource<Array<UserModel>>(() => ({
  url: `/users`,
  params: { sort: this.sortOrder() },
  headers: new HttpHeaders({ 'X-Custom-Header': this.customHeader() })
}));
```

You can of course send a body with the request, for example for a `POST`/`PUT` request, using the `body` property of the request object.

```
protected readonly query = signal('');
protected readonly filterUsersResource: HttpResourceRef<Array<UserModel> | undefined>
= httpResource<
  Array<UserModel>
>(() => {
  const query = this.query();
  return query
    ? {
        url: `/users`,
        method: 'POST',
        body: { query }
      }
    : undefined;
});
```

You can pass additional options in a second argument, where you can define:

- `injector`, in case the resource is not created at construction time;
- `defaultValue`, a default value of the resource, to use when idle, loading, or in error;
- an `equal` function that defines the equality of two values;
- a `parse` function that allows you to transform the response before setting it in the resource.

It is also possible to request something else than JSON, by using the `httpResource.text()`, `httpResource.blob()` or `httpResource.arrayBuffer()` functions.

Some of you may get a feeling of déjà vu with all this, as it's quite similar to the TanStack Query library. I must insist that this is experimental and will probably evolve in the future. It will also probably be used by higher-level APIs or libraries. Let's see what the RFC process will bring us!

# Chapter 29. Deferrable Views with `@defer`

With the introduction of the Control flow syntax, the Angular team has also introduced a new way to load components lazily (as a developer preview for now). We already have lazy-loading in Angular, but it is mainly based on the router.

Angular v17 adds a new way to load components lazily, using the `@defer` syntax in your templates.

`@defer` lets you define a block of template that will be loaded lazily when a condition is met (with all the components, pipes, directives and libraries used in this block lazily loaded as well). Several conditions can be used. For example, it can be "as soon as possible (no condition)", "when the user scrolls to that section", "when the user clicks on that button" or "after 2 seconds".

Let's say your home page displays a "heavy" `Chart` that uses a charting library and some other dependencies, like a `FromNow` pipe:

*chart.ts*

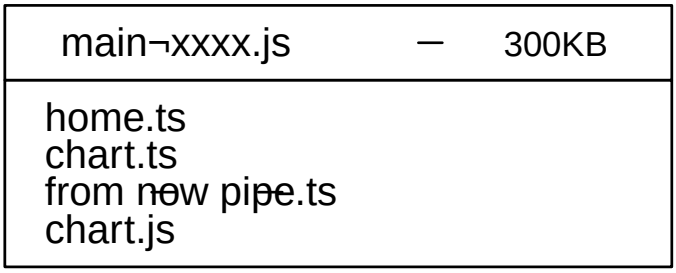
```
@Component({
  selector: 'ns-chart',
  template: '...',
  imports: [FromNowPipe],
})
export class Chart{
  // uses chart.js
}
```

*home.ts*

```
import { Chart} from './chart';

@Component({
  selector: 'ns-home',
  template: `
    <!-- some content -->
    <ns-chart />
  `,
  imports: [Chart]
})
export class Home{
  // ...
}
```

When the application is packaged, the `Chart` will be included in the main bundle:




Let's say that the component is not visible at first on the home page, maybe because it is at the bottom of the page, or because it is in a tab that is not active. It makes sense to avoid loading this component eagerly because it would slow down the initial loading of the page.

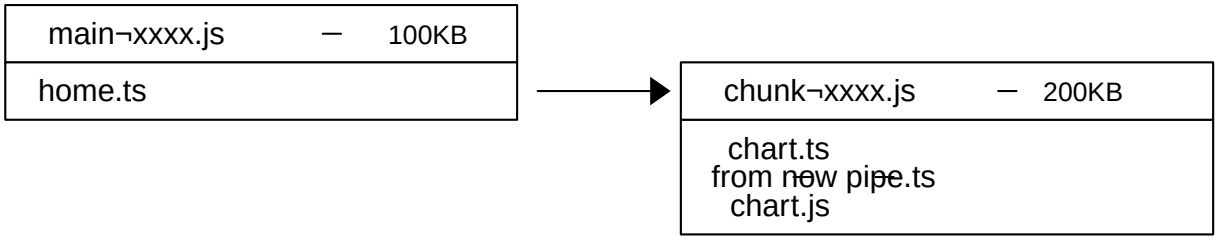
With `@defer`, you can load this component only when the user really needs it. Just wrapping the `Chart` in a `@defer` block will do the trick:

*home.ts*

```
import { Chart} from './chart';

@Component({
  selector: 'ns-home',
  template: `
    <!-- some content -->
    @defer (when isVisible) {
      <ns-chart />
    }
  `,
  imports: [Chart]
})
```

The Angular compiler will rewrite the static import of the `Chart` to a dynamic import (`() ⇒ import('./chart')`), and the component will be loaded only when the condition is met. As the component is now imported dynamically, it will not be included in the main bundle. The bundler will create a new chunk for it:




The `chunk-xxxx.js` file will only be loaded when the condition is met, and the `Chart` will be

displayed.

Before talking about the various kinds of conditions that can be used with `@defer`, let's see how to use another interesting feature: displaying a placeholder until the deferred block is loaded.

## 29.1. `@placeholder`, `@loading`, and `@error`

You can define a placeholder template with `@placeholder` that will be displayed until the loading condition is met. Then, while the block is loading, you can display a loading template with `@loading`. If no `@loading` block is defined, the placeholder stays there until the block is loaded. You can also define an error template with `@error` that will be displayed if the block fails to load.

```
@defer (when show()) {
  <ns-chart />
} @placeholder {
  <div>Something until the loading starts</div>
} @loading {
  <div>Loading...</div>
} @error {
  <div>Something went wrong</div>
}
```

When using server-side rendering, only the placeholder will be rendered on the server (the defer conditions will never trigger).

### 29.1.1. `after` and `minimum`

As the `@defer` block loading can be quite fast, there is a risk that the loading block is displayed and hidden too quickly, causing a "flickering" effect.

To avoid this, you can use the `after` option to specify after how many milliseconds the loading should be displayed.

If the block takes less than this delay to load, then the `@loading` block is never displayed.

You can also use the `minimum` option to specify a minimum duration for the loading. If the loading is faster than the minimum duration, then the loading will be displayed for the minimum duration (this only applies if the loading is ever displayed).

You can of course combine all these options:

```
@defer (when show()) {
  <ns-chart />
} @placeholder {
  <div>Something until the loading starts</div>
} @loading (after 500ms; minimum 500ms) {
  <div>Loading...</div>
}
```

You can also specify a `minimum` duration for the placeholder. It can be useful when the loading condition is immediate (for example, when no condition is specified). In that case, the placeholder will be displayed for the minimum duration, even if the block is loaded immediately, to avoid a "flickering" effect.

```
@defer (when show()) {
  <ns-chart />
} @placeholder (minimum 500ms) {
  <div>Something until the loading starts</div>
} @loading (after 500ms; minimum 500ms) {
  <div>Loading...</div>
}
```

## 29.2. Conditions

Several conditions can be used with `@defer`, let's see them one by one.

### 29.2.1. No condition or `on idle`

The simplest condition is to not specify any condition at all: in this case, the block will be loaded when the browser is idle (the loading is scheduled using `requestIdleCallback`).

```
@defer {
  <ns-chart />
}
```

This is equivalent to using the `on idle` condition:

```
@defer (on idle) {
  <ns-chart />
}
```

### 29.2.2. Simple boolean condition with `when`

You can also use a boolean condition to load a block of the template with `when`. Here, we display the defer block only when the `show` property of the component is true:

```
@defer (when show()) {
  <ns-chart />
}
```

Note that this is not the same as using `*ngIf` on the block, as the block will not be removed even if the condition becomes false later.

### 29.2.3. on `immediate`

The `on immediate` condition triggers the loading of the block immediately. It does not display a placeholder, even if one is defined.

### 29.2.4. on `timer`

The `on timer` condition triggers the loading of the block after a given duration, using `setTimeout` under the hood.

```
@defer (on timer(2s)) {
  <ns-chart />
}
```

### 29.2.5. on `hover`

Other conditions are based on user interactions. These conditions can specify the element of the interaction using a template reference variable, or none to use the placeholder element. In the latter case, the placeholder element must exist and have a single child element that will be used as the element of the interaction.

The `on hover` condition triggers the loading of the block when the user hovers the element. Under the hood, it listens to the `mouseenter` and `focusin` events.

```
<span #trigger>Hover me</span>

@defer (on hover(trigger)) {
  <ns-chart />
}
```

or using the placeholder element:

```
@defer (on hover) {
  <ns-chart />
} @placeholder {
  <span>Hover me</span>
}
```

### 29.2.6. on `interaction`

The `on interaction` condition triggers the loading of the block when the user interacts with the element. Under the hood, it listens to the `click` and `keydown` events.

### 29.2.7. on `viewport`

The `on viewport` condition triggers the loading of the block when the element becomes visible in the

viewport. Under the hood, it uses an intersection observer.

### 29.2.8. Multiple conditions

You can also combine multiple conditions using a comma-separated list:

```
<!-- Loads if the user hovers the placeholder, or after 1 minute -->
@defer (on hover, timer(60s)) {
  <ns-chart />
} @placeholder {
  <span>Something until the loading starts</span>
}
```

## 29.3. Prefetching

`@defer` allows you to separate the loading of a component from its display. You can use the same conditions we previously saw to load a component using `prefetch`, and then display it with another condition.

For example, you can prefetch the lazy-loaded content `on idle` and then display it `on interaction`:

```
@defer (on interaction; prefetch on idle) {
  <ns-chart />
} @placeholder {
  <button>Show me</button>
}
```

Note that the `@loading` block will not be displayed if the deferred block is already prefetched when the loading condition is met.

## 29.4. How to test deferred loading?

When a component uses defer blocks in its template, you'll have to do some extra work to test it.

The `TestBed` API has been extended to help you with that. The `configureTestingModule` method now accepts a `deferBlockBehavior` option. By default, this option is set to `DeferBlockBehavior.Playthrough`, which means that the defer blocks will be displayed automatically when a condition is met, as they would when the application runs in the browser.

You can change this behavior by using `DeferBlockBehavior.Manual`. `Manual` means that you'll have to manually trigger the display of the defer blocks. But let's start with the default option, `Playthrough`.

In that case, the defer blocks will be displayed automatically when a condition is met, after calling `await fixture.whenStable()`.

So if we test a component with a deferred block that is visible after clicking on a button, we can use:

```
// Click the button to trigger the deferred block
(fixture.nativeElement as HTMLElement).querySelector('button')!.click();
fixture.detectChanges();

// Wait for the deferred block to render
await fixture.whenStable();

// Check its content
const loadedBlock = (fixture.nativeElement as HTMLElement).querySelector('div')!;
expect(loadedBlock.textContent).toContain('Some lazy-loaded content');
```

If you want to use the `DeferBlockBehavior.Manual` behavior, you'll have to manually trigger the display of the defer blocks.

```
await TestBed.configureTestingModule({
  deferBlockBehavior: DeferBlockBehavior.Manual
}).compileComponents();
```

To do so, the fixture returned by `TestBed.createComponent` now has an async `getDeferBlocks` method that returns an array of `DeferBlockFixture` objects. Each of these fixtures has a `render` method that you can call to display the block in a specific state, by providing a `DeferBlockState` parameter.

`DeferBlockState` is an enum with the following values:

- `DeferBlockState.Placeholder`: display the placeholder state of the block
- `DeferBlockState.Loading`: display the loading state of the block
- `DeferBlockState.Error`: display the error state of the block
- `DeferBlockState.Complete`: display the defer block as if the loading was complete

This allows a fine-grained control of the state of the defer blocks. If we want to test the same component as before, we can do:

```
const deferBlocks = await fixture.getDeferBlocks();
// only one defer block should be found
expect(deferBlocks.length).toBe(1);

// Render the defer block
await deferBlocks[0].render(DeferBlockState.Complete);

// Check its content
const loadedBlock = (fixture.nativeElement as HTMLElement).querySelector('div')!;
expect(loadedBlock.textContent).toContain('Some lazy-loaded content');
```

# Chapter 30. Going to production

So now you've built an application, and you are seriously thinking about showing it to the world. Let's have a look about what you need to do to go to production!

## 30.1. Environments and configurations

If you use Angular CLI, you can define several environments.

To do this, the CLI offers a schematic called `environment`:

```
ng generate environments
```

This generates files named `environment.ts` and `environment.development.ts`.

These files contain an empty object called `environment`, in which you can add as many properties as you want.

*environment.ts*

```
export const environment = {};
```

You'll then only import `environment.ts` in your application. It's a bit weird, but the CLI will then use the right file according to the environment.

When serving (with `ng serve`) your application, the CLI (Webpack, to be more accurate) will use `environment.development.ts`.

But you can also serve your application with a specified configuration. By default, the CLI has another configuration named `production`.

So you can also run `ng serve --configuration=production`. The difference between these configurations can be found in the `angular.json` files:

*angular.json*

```
"configurations": {
  "production": {
    "budgets": [
      {
        "type": "initial",
        "maximumWarning": "500kB",
        "maximumError": "1MB"
      },
      {
        "type": "anyComponentStyle",
        "maximumWarning": "4kB",
        "maximumError": "8kB"
```

```
      }
    ],
    "outputHashing": "all"
  },
  "development": {
    "optimization": false,
    "extractLicenses": false,
    "sourceMap": true,
    "fileReplacements": [
      {
        "replace": "src/environments/environment.ts",
        "with": "src/environments/environment.development.ts"
      }
    ]
  },
```

As you can see, there is a `production` configuration with a few properties. The `budgets` one, for example, checks that your initial loading and component styles are not too heavy.

There is a super useful property in the development configuration that the schematic added: `fileReplacements`.

*angular.json*

```
"fileReplacements": [
  {
    "replace": "src/environments/environment.ts",
    "with": "src/environments/environment.development.ts"
  }
]
```

You can see that the `environment.ts` file is replaced by `environment.development.ts`: this is how Webpack knows which file to use.

You then always import from `environment.ts` in your code, and, during the build, the CLI will pick the proper environment file for the configuration.

This means that you can define as many configurations as you want. For example, you could add a `preprod` configuration with a dedicated `environment.preprod.ts` file.

It also means that you can replace as many files as you want in your application. You can imagine doing crazy things like replacing `pony.ts` with a different version (I don't see why you would do that though ^^).

An environment file can contain whatever you want. But as its name indicates, it's supposed to contain code that is specific to a given environment (development, production, pre-production, etc.). For example, you may have a different API location in development than in the live version.

As the `production` environment is often the one you want when you build the application with `ng build`, the CLI uses it by default since version 12.0 (instead of having to specify

`--configuration=production`).

Since the CLI version 9.0, it is now possible to specify several configurations at once:

```
ng build --configuration=production,preprod
```

The command then uses the `production` configuration, merged with the `preprod` configuration. The `preprod` configuration can re-declare a property of the `production` configuration, to overwrite it.

Note that this replacement mechanism is also available for assets and styles in the CLI, giving you the possibility to theme your applications differently by just using configurations.

## 30.2. strictTemplates

When you compile your application in AoT (`aot: true`, the default value since Angular 9.0), the templates are checked by the Angular compiler.

By default, only a light check is run. To go further, you can use the `strictTemplates` compiler option:

*tsconfig.json*

```
"angularCompilerOptions": {
  "strictTemplates": true,
}
```

With this option, the compiler checks that the input values, DOM events used, references in templates, etc. are all of the correct type. For example, trying to feed a number into an input that expects a boolean, results in a compilation error.

Angular 13.2 introduced a new option called `extendedDiagnostics`, which runs additional checks on your application. For example, it checks that you use the two-way binding syntax correctly (the "banana syntax" `[(ngModel)]`, and not `([ngModel])`). This option logs warnings by default, but you can configure it to throw errors:

*tsconfig.json*

```
"angularCompilerOptions": {
  "strictTemplates": true,
  "extendedDiagnostics": {
    "defaultCategory": "error"
  }
}
```

Angular 20 also introduced a new option called `typeCheckHostBindings` which checks that the expressions in your host bindings are correct.

## 30.3. Package your application

I slightly spoiled the next step in the previous sections. If you want to package your application for production using the CLI, you simply have to run:

```
ng build
```

That will create a `dist` folder containing the result of the build. The `prod` flag uses the file replacement we just talked about, but also adds a bunch of other options. Some of them are really interesting.

It also activates tree-shaking and dead code elimination thanks to `optimization: true`. To make this process even more efficient, the Angular CLI team has written a tool named "build optimizer", which is activated by the `buildOptimizer: true` option.

To further reduce the volume of JavaScript code generated, the third party libraries are not bundled in a separate file (`vendorChunk: false`) - the licences are removed (`extractLicences: true`).

You typically don't want to debug in production. And you probably don't want to provide the non-minified source code to any visitor of your application either. So `sourceMap: false` disables the time-consuming generation of the source maps.

A last interesting option is `outputHashing`: it tells Webpack that the generated files should not be called `main.js` but `main.xxxxxx.js` where `xxxxxx` is a cryptographic hash of the content of the file. This is done to be sure that you can cache these files without worrying about it: see the section about "cache busting" in the Performances chapter just above.

As you can see, the CLI does plenty of nice optimizations for you. If you choose not to use the CLI, you'll have to figure out a way to do the same things by using Webpack or another tool.

## 30.4. Server configuration

The last thing to do is to take the result of the packaging step (in the `dist` folder) and to deploy it on your favorite server. That can be a static server like Apache or Nginx for example.

Do not use `ng serve` in production, even with the `--configuration=production` option. It's only a development tool, and is neither optimized nor secured for production.

You still have a few things to do though. The precise way to do these things depends on your web server.

First you'll want to make sure you are serving all your assets compressed (probably using gzip). Then you want the assets to be cached for a long time. Don't worry about potential cache issues, as we generated the assets with a hash in their names.

The last step is less obvious and often forgotten (been there, done that...): you need to configure your server in a way that ensures that each route will serve `index.html`.

Think about it: you deployed your application on `https://ng-ponyracer.ninja-squad.com`, tweeted it to the world, and people are starting to visit it. Your server will serve the `index.html` file to them and everything is fine. They are navigating in the application, maybe going to the list of races at `https://ng-ponyracer.ninja-squad.com/races`.

But what happens if someone hits `F5`/`Cmd` + `R`? The next request to the server will be for `/races`, and if you did not plan for it, your server will return a `404`…

So you have to make sure that, one way or another, all requests to routes of your application will serve `index.html`. The Angular application will restart from scratch, the router will analyze the URL, and it will navigate to the proper route immediately.

## 30.5. Conclusion

I think we covered the important parts of going to production. As you can see this is fairly easy if you use Angular CLI, so I strongly encourage you to do so.

Try our exercise Going to production 🦄! You'll learn a few tricks that'll be useful in the future.

# Chapter 31. This is the end

Thanks for reading!

There are some other chapters that will be added in the following releases on more 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.

Current versions:

- Angular: `20.0.1`
- Angular CLI: `20.0.1`

## A.1. v20.0.0 - 2025-05-28

**Testing your app**

- We now use Playwright for e2e tests instead of Cypress. (2025-05-07)

**Advanced components and directives**

- `afterRender` has been renamed to `afterEveryRender` in Angular v20. (2025-05-07)

**Signals: advanced topics**

- The resource status is now a string instead of an enum in Angular v20. (2025-05-13)
- `request` has been renamed to `params` in `resource()` in Angular v20. (2025-05-07)

**Performances**

- The zoneless section has been updated. (2025-05-28)

**Going to production**

- Mention the new `typeCheckHostBindings` option introduced in v20. (2025-03-27)

## A.2. v19.2.0 - 2025-02-26

**Signals: advanced topics**

- Add a section about `httpResource()`, introduced in v19.2 (2025-02-20)
- Add a section about `resource` with a `stream` and update the `rxResource` section according to changes in v19.2. (2025-02-09)
- We can define a `defaultValue` in a `resource` in v19.2. (2025-02-07)

## A.3. v19.1.0 - 2025-01-16

# A.4. v19.0.0 - 2024-11-19

**Signals basics**

- New chapter to introduce signal basics at the beginning of the ebook! (2024-09-26)

**Reactive Programming**

- Rewrite and introduce the interoperability with signals. (2024-11-07)

**Building components and directives**

- Chapter updated to explain `input()` and `output()`. (2024-10-24)

**Reacting to signal changes**

- New chapter about `computed` and `effect` and how to use them. (2024-10-24)

**Send and receive data with Http**

- Showcase how to use `toSignal` to subscribe. (2024-11-19)

**Advanced components and directives**

- Use `viewChild()`/`contentChild()` instead of decorators. (2024-11-19)

**Signals: advanced topics**

- New chapter about advanced topics with Signals! (2024-11-19)
- Add a section about the `resource` and `rxResource` functions introduced in v19. (2024-11-19)
- Add a section about the experimental `linkedSignal` introduced in v19. (2024-11-19)

**Performances**

- Rewrite the chapter to use signals and solve a lot of problems our of the box! (2024-11-19)

# A.5. v18.2.0 - 2024-08-15

# A.6. v18.1.0 - 2024-07-10

**The templating syntax**

- Add a section about `@let` variables, as introduced in Angular v18.1. (2024-07-08)

**Building components and directives**

- Add a section about `afterRender` and `afterNextRender`. (2024-06-21)

**Performances**

- Refresh the performances chapter regarding the control flow syntax, signals, and the

experimental zoneless detection (2024-05-31)

- Mention the `--ssr` option of the CLI for Server Side Rendering. (2024-05-23)

# A.7. v18.0.0 - 2024-05-22

**The templating syntax**

- The chapter now introduces `@if`/`@for`/`@switch` from the control flow syntax as the recommended way to write templates. We kept a section about `*ngIf`/`*ngFor`/`*ngSwitch` as they are not deprecated and can still be used. All template examples across the ebook now use the control flow syntax. (2024-04-22)

**Building components and directives**

- Add a section about fallback content for `ng-content`, as introduced in Angular v18. (2024-05-02)

**Forms**

- Mention the `events` observable on `FormControl`, introduced in Angular v18. (2024-05-03)

**Send and receive data with Http**

- Mention that `HttpClientModule` is deprecated in Angular v18. (2024-04-27)

**Internationalization**

- Add a section about i18n with Transloco and new exercice to go along with it! (2024-03-16)

# A.8. v17.3.0 - 2024-03-14

**Signals**

- Add a section about the `output()` function introduced in v17.3 (2024-03-08)

**Advanced observables**

- Add a section about using subjects as triggers. (2024-02-27)

# A.9. v17.2.0 - 2024-02-15

**Signals**

- Add a section about the `model()` function introduced in v17.2 (2024-02-13)
- Add a section about the queries as signals functions (`viewChild()`/`viewChildren()`/`contentChild()`/`contentChildren()`) introduced in v17.2 (2024-02-12)

**Deferred loading with `@defer`**

- The defer block fixture default behavior switched to `Playthrough`. (2024-02-01)

## A.10. v17.1.0 - 2024-01-18

**Signals**

- Add a section about input as signals, as introduced in v17.1 (2024-01-17)
- Mention the new `Signals` exercise added to the Pro Pack! (2023-12-23)
- Add a section about how to handle nullable values in signals. (2023-12-23)

## A.11. v17.0.0 - 2023-11-08

**Styling components and encapsulation**

- We now use `styleUrl` when possible, as introduced in Angular v17. (2023-11-08)

**Signals**

- Remove the `mutate` method from examples,a s it has been remove in Angular v17. (2023-10-12)

**Control flow syntax**

- New chapter about the control flow syntax introduced in Angular v17! (2023-10-08)

**Deferred loading with `@defer`**

- New chapter about deferred loading with `@defer` as introduced in Angular v17! (2023-10-30)

## A.12. v16.2.0 - 2023-08-10

**Building components and directives**

- Add a section about the `transform` option of `@Input`, introduced in Angular v16.1. (2023-06-24)

## A.13. v16.1.0 - 2023-06-14

## A.14. v16.0.0 - 2023-05-17

**Building components and directives**

- Introduce required inputs, as added in Angular v16 (2023-05-03)

**Router**

- Add a section about `withComponentInputBinding` to get router parameters and data as component inputs, as introduced in Angular v16 (2023-05-03)

**Signals**

- New chapter about Signals! (2023-05-17)

### Advanced observables

- Use the `takeUntilDestroyed` RxJS operator introduced in Angular v16 (2023-05-03)

## A.15. v15.2.0 - 2023-02-23

### Router

- As Angular v15.2 deprecates class-based resolvers and guards, we now use functional resolvers and guards in all examples. (2023-02-23)

## A.16. v15.1.0 - 2023-01-11

### Dependency Injection

- Use a better example for DI configuration, with a logging service that logs to the console in development and calls an API in production. (2023-01-05)
- Add a section about the `inject()` function. (2022-12-01)

### Router

- Remove the section about the `CanLoad` guard as it is now deprecated (use `CanMatch` instead). (2023-01-11)

### Standalone components

- Add a section about HTTP with `provideHttpClient` and functional interceptors. (2022-11-30)

### Going to production

- Explains how to use `ng generate environments`. (2023-01-11)

## A.17. v15.0.0 - 2022-11-16

### Dependency Injection

- Remove the `providedIn: NgModule` syntax now that it is deprecated in Angular v15 (2022-11-16)

### Router

- The router automatically unwraps default module exports in lazy-loading routes in Angular v15 (2022-11-16)
- Showcases an example of a functional resolver (2022-11-14)
- Showcases an example of functional guard (2022-11-14)

### Standalone components

- Use the `NgFor` alias introduced in Angular v15 for the `NgForOf` directive (2022-11-16)

- The router now automatically unwraps default component exports in lazy-loading routes (2022-11-16)

**Going to production**

- Replace the explanation of `enableProdMode` by a section about production mode and mention the `ngDevMode` variable. (2022-11-16)
- We now explain how to use `fileReplacements` as it is no longer included by default in CLI v15. (2022-11-16)

# A.18. v14.2.0 - 2022-08-26

**Standalone components**

- Mention `provideRouter(routes)` (2022-08-26)

**Performances**

- Mention the experimental `NgOptimizedImage` directive introduced in v14.2 (2022-08-26)

# A.19. v14.1.0 - 2022-07-21

**Router**

- Add a section on the new `CanMatch` guard introduced in v14.1 (2022-07-21)

# A.20. v14.0.0 - 2022-06-03

**Forms**

- Add a section about `FormArray` and `FormRecord` (2022-06-03)
- Add a section about typed forms (2022-06-03)
- We nows use and explain the new "strictly typed forms API" 🚀 (2022-06-03)

**Standalone components**

- New chapter about standalone APIs! (2022-06-03)

**Performances**

- Better example of `NgZone.runOutsideAngular` usage (2022-05-11)

# A.21. v13.3.0 - 2022-03-16

# A.22. v13.2.0 - 2022-01-27

**Forms**

- The forms chapter has a new section about control value accessors, explaining how to create custom form controls (2021-12-14)

**Advanced components and directives**

- The advanced components chapter has a new section about ng-template, explaining how to create customizable components using conditional, contextual content projection (2021-12-17)

**Going to production**

- Section about the new `extendedDiagnostics` option introduced in v13.2 (2022-01-27)

# A.23. v13.1.0 - 2021-12-10

# A.24. v13.0.0 - 2021-11-04

**The templating syntax**

- Remove the canonical `bind-`, `on-`, `ref-` syntax that has been deprecated in Angular v13 (2021-11-04)

**Going to production**

- Remove the section about differential loading as it has been removed in Angular v13 (2021-11-04)
- Remove the `fullTemplateTypeCheck` explanation, as it is deprecated in Angular v13, and only keep its remplacement `strictTemplates`. (2021-11-04)

# A.25. v12.2.0 - 2021-08-05

**Global**

- Add links to our quizzes! (2021-07-29)

**Reactive Programming**

- RxJS v7.2 allows to import operators directly from `rxjs`, so all imports have been simplified. (2021-08-05)

# A.26. v12.1.0 - 2021-06-25

# A.27. v12.0.0 - 2021-05-13

**Global**

- All examples now use strict null checks. (2021-05-13)

**From zero to something**

- The ebook now uses ESLint as its linter. (2021-05-13)

### Testing your app

- The e2e tests section now introduces Cypress (2021-05-13)

### Send and receive data with Http

- Section about the new `HttpContext` introduced in Angular v12. (2021-05-13)
- The HTTP examples now use the human-readable `HttpStatusCode` enum. (2021-05-13)

### Going to production

- The CLI uses the production configuration by default for `ng build` since v12, and the `--prod` flag is deprecated. (2021-05-13)

## A.28. v11.2.0 - 2021-02-12

## A.29. v11.1.0 - 2021-01-21

## A.30. v11.0.0 - 2020-11-12

### Internationalization

- `ng xi18n` has been renamed `ng extract-i18n` in CLI v11 (2020-11-12)

## A.31. v10.2.0 - 2020-10-22

### Internationalization

- `xi18N` now extracts messages from the `$localize` calls in TypeScript code (2020-09-10)

## A.32. v10.1.0 - 2020-09-03

### Testing your app

- `async` has been deprecated and renamed `waitForAsync` (2020-09-01)

### Internationalization

- Import the global variants of the locale data. It's simpler, supports all formatting options, and doesn't trigger an optimization bailout warning when building the app with the CLI. (2020-07-01)

## A.33. v10.0.0 - 2020-06-25

### Global

- Bump to ng `10.0.0` (2020-06-25)

### The wonderful world of Web Components

- Use `customElements.define` instead of the deprecated `document.registerElement`. (2020-06-17)

**Reactive Programming**

- Pass an object as argument to the `Observable.subscribe()` method when an error or a completion must be handled, instead of 2 or 3 functions, because passing several functions will be deprecated in RxJS 7. (2020-06-05)

# A.34. v9.1.0 - 2020-03-26

**Global**

- Bump to ng `9.1.0` (2020-03-26)

**From zero to something**

- Bump to cli `9.1.0` (2020-03-26)

# A.35. v9.0.0 - 2020-02-07

**Global**

- Bump to ng `9.0.0` (2020-02-07)
- Bump to ng `9.0.0-next.5` (2020-02-06)

**A gentle introduction to ECMAScript 2015+**

- Add a section about tagged template strings. (2019-08-02)

**Diving into TypeScript**

- Showcase interface usage for modeling entities (2019-08-10)
- Improve the `enum` section with examples of how to use union types (2019-08-10)

**Advanced TypeScript**

- Introduce a new chapter about advanced TypeScript patterns, like `keyof`, mapped types, type guards, and other things! (2019-08-10)

**From zero to something**

- Bump to cli `9.0.1` (2020-02-07)
- Bump to cli `9.0.0-next.3` (2020-02-06)
- Bump to cli `8.3.2` (2019-08-30)
- Bump to cli `8.3.0` (2019-08-22)

**Testing your app**

- Use `TestBed.inject` instead of the deprecated `TestBed.get` in ng `9.0.0` (2020-02-06)

### Internationalization

- Explains how to configure the default currency code (2020-02-07)
- Introduce `@angular/localize` usage in ng `9.0.0` (2020-02-07)

### Going to production

- Mention the multiple configurations support introduced in CLI v9.0 (2020-02-07)
- Explain the `fullTemplateTypeCheck` and `strictTemplates` options (2020-02-07)

## A.36. v8.2.0 - 2019-08-01

### Global

- Bump to ng `8.2.0` (2019-08-01)

### From zero to something

- Bump to cli `8.2.0` (2019-08-01)

### Testing your app

- Use a more strictly typed `createSpyObj` syntax. (2019-07-31)

## A.37. v8.1.0 - 2019-07-02

### Global

- Bump to ng `8.1.0` (2019-07-02)

### The wonderful world of Web Components

- Mention more recent alternatives to Polymer, remove the dead HTML import spec and mention Angular Elements (2019-06-01)

### From zero to something

- Bump to cli `8.1.0` (2019-07-02)

## A.38. v8.0.0 - 2019-05-29

### Global

- Bump to ng `8.0.0` (2019-05-29)

### A gentle introduction to ECMAScript 2015+

- How to use `async/await` with promises (2019-05-19)

### From zero to something

- Bump to cli `8.0.0` (2019-05-29)
- Bump cli to `7.3.0` (2019-02-28)

**Testing your app**

- Showcase the awesome `ngx-speculoos` library for cleaner unit tests (2019-05-20)

**Forms**

- Showcase the awesome `ngx-valdemort` library for better validation error messages (2019-05-19)

**Router**

- Use `import` for lazy-loading routes as introduced by ng `8.0.0` (2019-05-20)

**Angular compiler**

- Update the AoT explanation and generated code for Angular 8.0.0 (Ivy) (2019-05-20)

**Advanced components and directives**

- Add and explain the `static` flag for `ViewChild` and `ContentChild` introduced by Angular `8.0.0` (2019-05-27)

**Going to production**

- Differential loading using `browserslist` as introduced by the cli `8.0.0`. (2019-05-20)

# A.39. v7.2.0 - 2019-01-09

**Global**

- Bump to ng `7.2.0` (2019-01-07)
- Bump to ng `7.2.0-rc.0` (2019-01-03)
- Bump to ng `7.2.0-beta.2` (2018-12-14)

**From zero to something**

- Bump to cli `7.2.0` (2019-01-09)
- Bump to cli `7.2.0-rc.0` (2019-01-07)
- Bump to cli `7.2.0-beta.2` (2019-01-07)

# A.40. v7.1.0 - 2018-11-27

**Global**

- Bump to ng `7.1.0` (2018-11-22)
- Bump to ng `7.1.0-rc.0` (2018-11-20)

- Bump to ng `7.0.2` (2018-11-05)

**From zero to something**

- Bump to cli `7.1.0` (2018-11-27)
- Bump to cli `7.0.4` (2018-11-05)

**Router**

- Use `UrlTree` in `CanActivate` guard, as introduced by 7.1 (2018-11-22)

# A.41. v7.0.0 - 2018-10-25

**Global**

- Bump to ng `7.0.0` (2018-10-18)
- Bump to ng `7.0.0-rc.1` (2018-10-18)
- Bump to ng `7.0.0-rc.0` (2018-10-18)
- Bump to ng `7.0.0-beta.6` (2018-10-18)
- Bump to ng `7.0.0-beta.4` (2018-10-18)
- Bump to ng `7.0.0-beta.0` (2018-10-18)

**From zero to something**

- Bump to cli `7.0.2` (2018-10-24)
- Bump to cli `7.0.1` (2018-10-18)
- Bump to cli `6.2.1` (2018-09-07)
- Bump to cli `6.2.0-rc.0` (2018-09-07)

**Performances**

- Adds a performances chapter! (2018-08-30)

**Going to production**

- Adds a new chapter about Going to production! (2018-10-25)

# A.42. v6.1.0 - 2018-07-26

**Global**

- Bump to ng `6.1.0` (2018-07-26)
- Bump to ng `6.1.0-rx.0` (2018-07-26)
- Bump to ng `6.1.0-beta.1` (2018-07-26)
- Bump to ng `6.0.7` (2018-07-06)

### From zero to something

- Bump to cli `6.1.0` (2018-07-26)
- Bump to cli `6.0.8` (2018-07-06)
- Bump cli to `6.0.7` (2018-05-30)

### Pipes

- Add the `keyvalue` pipe introduced in Angular 6.1 (2018-07-26)
- Show usage of formatting functions available since Angular 6.0 (2018-06-15)

### Styling components and encapsulation

- New `ShadowDom` encapsulation option with Shadow DOM v1 support (the old and soon deprecated `Native` option uses Shadow DOM v0) (2018-07-26)

### Send and receive data with Http

- HTTP tests now use `verify` every time (2018-07-06)

### Router

- Adds the `Scroll` event and `scrollPositionRestoration` option introduced in 6.1 (2018-07-26)

### Advanced observables

- Use `shareReplay` instead of `publishReplay` and `refCount` (2018-07-20)

### Internationalization

- Update for CLI 6.0 and use a dedicated configuration (2018-05-09)

## A.43. v6.0.0 - 2018-05-04

### Global

- Bump to ng `6.0.0` (2018-05-04)
- Bump to ng `6.0.0-rc.4` (2018-04-13)
- Bump to ng `6.0.0-rc0` (2018-04-05)
- Bump to ng `6.0.0-beta.7` (2018-04-05)
- Bump to ng `6.0.0-beta.6` (2018-04-05)
- Bump to ng `6.0.0-beta.1` (2018-04-05)

### The wonderful world of Web Components

- Replace customelements.io by webcomponents.org (2018-01-19)

### From zero to something

- Bump to cli `6.0.0` (2018-05-04)
- The chapter now uses Angular CLI from the start! (2018-03-19)

**Dependency Injection**

- Use `providedIn` to register services, as recommended for Angular 6.0 (2018-04-15)
- Updates the dependency injection via token section with a better example (2018-03-19)

**Reactive Programming**

- We now use the pipeable operators introduced in RxJS 5.5 (2018-01-28)

**Services**

- Use `providedIn` to register the service, as recommended for Angular 6.0 (2018-04-15)

**Testing your app**

- Simplify service unit tests now that they use `providedIn` from ng 6.0 (2018-04-15)

**Advanced components and directives**

- Angular 6.0+ allows to type ElementRef<T> (2018-04-05)

**Advanced observables**

- We now use the imports introduced in RxJS 6.0 (`import { Observable, of } from 'rxjs'`) (2018-04-05)
- We now use the pipeable operators introduced with RxJS 5.5 (2018-01-28)

# A.44. v5.2.0 - 2018-01-10

**Global**

- Bump to ng `5.2.0` (2018-01-10)
- Bump to ng `5.1.0` (2017-12-07)

**Building components and directives**

- Better lifecycle explanation (2017-12-13)

**Forms**

- Reintroduce the `min` and `max` validators from version 4.2.0, even if they are not available as directives. (2017-12-13)

**Send and receive data with Http**

- Remove remaining mentions to the deprecated `HttpModule` and `Http` (2017-12-08)

# A.45. v5.0.0 - 2017-11-02

### Global

- Bump to ng `5.0.0` (2017-11-02)
- Bump to ng `5.0.0-rc.5` (2017-11-02)
- Bump to ng `5.0.0-rc.3` (2017-11-02)
- Bump to ng `5.0.0-rc.2` (2017-11-02)
- Bump to ng `5.0.0-rc.0` (2017-11-02)
- Bump to ng `5.0.0-beta.6` (2017-11-02)
- Bump to ng `5.0.0-beta.5` (2017-11-02)
- Bump to ng `5.0.0-beta.4` (2017-11-02)
- Bump to ng `5.0.0-beta.1` (2017-11-02)
- Bump to ng `4.4.1` (2017-09-16)

### Pipes

- Use the new i18n pipes introduced in ng 5.0.0 (2017-11-02)

### Forms

- Add a section on the `updateOn: 'blur'` option for controls and groups introduced in 5.0 (2017-11-02)
- Remove the section about combining template-based and code-based approaches (2017-09-01)

### Send and receive data with Http

- Use object literals for headers and params for the new http client, introduced in 5.0.0 (2017-11-02)

### Router

- Adds ng 5.0 `ChildActivationStart`/`ChildActivationEnd` to the router events (2017-11-02)

### Internationalization

- Remove deprecated i18n comment with ng 5.0.0 (2017-11-02)
- Show how to load the locale data as required in ng 5.0.0 and uses the new i18n pipes (2017-11-02)
- Placeholders now displays the interpolation in translation files to help translators (2017-11-02)

# A.46. v4.3.0 - 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.47. v4.2.0 - 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.48. v4.0.0 - 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.49. v2.4.4 - 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.50. v2.2.0 - 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.51. v2.0.0 - 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.52. v2.0.0-rc.5 - 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.53. v2.0.0-rc.0 - 2016-05-06

### 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` (2016-03-19)
- 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 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)

# A.54. v2.0.0-alpha.47 - 2016-01-15

**Global**

- First public release of the ebook! (2016-01-15)