Switching to Angular 2

AngularJS is the modern framework you need to build performant and robust web applications. Switching to Angular 2 is the quickest way to get to grips with Angular 2 and will help you transition into the brave new world of Angular 2.

We'll start with an overview which puts the changes of the framework into context with version 1.x. After that, you will be taken on a TypeScript crash course so we can take advantage of Angular 2 in its native, statically-typed environment. We'll look at the new change detection method in detail, how Directives and Components change how you create websites with Angular, the new Angular 2 router, and much more.

By the end of the book, you'll be ready to start building quick and efficient Angular 2 applications that take advantage of all the new features on offer.

Who this book is written for

Do you want to jump in at the deep end of Angular 2? Or perhaps you're interested assessing the changes before moving over? If so, then Switching to Angular 2 is the book for you.

To get the most out of the book, you'll need to have basic understanding of AngularJS 1.x and have a good understanding of JavaScript. No knowledge of the changes made to Angular 2 is required.

What you will learn from this book

- Understand the changes made from AngularJS 1.x with side-by-side code samples to help demystify the Angular 2 learning curve
- Start working with Angular 2's new method of implementing directives
- Use TypeScript to write modern, powerful Angular 2 applications
- Dig in to the change detection method and other architectural changes to make sure you know what's going on under the hood of Angular 2
- Get to work with the new router in Angular 2
- Use the new features of Angular 2, including pipes, and the updated features, such as forms, services, and the DI module
- Understand server-side rendering in Angular 2 to keep your new applications SEO friendly

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 'The Building Blocks of an Angular 2 Application'
- A synopsis of the book’s content
- More information on Switching to Angular 2
Minko Gechev is a software engineer who strongly believes in open source software. He has developed numerous such projects, including AngularJS 1.x and Angular 2 style guides, angular2-seed, a static code analyzer for Angular 2 projects, aspect.js, angular-aop, and many others. He runs training courses in JavaScript, Angular, and other web technologies.

Minko loves to experiment with theoretical concepts from computer science and apply them in practice. He has spoken about Angular and software development at worldwide conferences and meetups, including ng-vegas, AngularConnect, ITWeekend Kiev, AngularJS-SF, and Angular Berlin.
AngularJS is a JavaScript development framework that makes building web applications easier. It is used today in large-scale, high-traffic websites that struggle with underperformance and portability issues, as well as SEO unfriendliness and complexity at scale. Angular 2 changes these.

It is the modern framework you need to build performant and robust web applications. *Switching to Angular 2* is the quickest way to get to grips with Angular 2, and it will help you transition into the brave new world of Angular 2.

By the end of the book, you’ll be ready to start building quick and efficient Angular 2 applications that take advantage of all the new features on offer.

**What this book covers**

*Chapter 1, Getting started with Angular 2*, kicks off our journey into the world of Angular 2. It describes the main reasons behind the design decisions of the framework. We will look into the two main drivers behind the shape of the framework—the current state of the Web and the evolution of frontend development.

*Chapter 2, The Building Blocks of an Angular 2 Application*, gives an overview of the core concepts introduced by Angular 2. We’ll explore how the foundational building blocks for the development of applications provided by AngularJS 1.x differ from the ones in the last major version of the framework.

*Chapter 3, TypeScript Crash Course*, explains that although Angular 2 is language agnostic, Google’s recommendation is to take advantage of the static typing of TypeScript. In this chapter, you’ll learn all the essential syntax you need to develop Angular 2 applications in TypeScript!
Preface

Chapter 4, Getting Started with Angular 2 Components and Directives, describes the core building blocks for developing the user interface of our applications—directives and components. We will dive into concepts such as view encapsulation, content projection, inputs and outputs, change detection strategies, and more. We’ll discuss advanced topics such as template references and speeding up our applications using immutable data.

Chapter 5, Dependency Injection in Angular 2, covers one of the most powerful features in the framework, which was initially introduced by AngularJS 1.x: its dependency injection mechanism. It allows us to write more maintainable, testable, and understandable code. By the end of this chapter, we will know how to define the business logic in services and glue them together with the UI through the DI mechanism. We will also look into some more advanced concepts, such as the injectors hierarchy, configuring providers, and more.

Chapter 6, Working with the Angular 2 Router and Forms, explores the new module for managing forms in the process of developing a real-life application. We will also implement a page that shows the entered through the form data. In the end, we will glue the individual pages together into an application by using the component-based router.

Chapter 7, Explaining Pipes and Communicating with RESTful services, dives into the router and the forms modules in detail. Here, we will explore how we can develop model-driven forms and define parameterized and child routes. We will also explain the HTTP module and see how we can develop pure and impure pipes.

Chapter 8, SEO and Angular 2 in the Real World, explores some advanced topics in the Angular 2 application development, such as running an application in Web Workers and server-side rendering. In the second part of the chapter, we will explore tools that can ease our daily life as developers, such as angular-cli and angular2-seed, explain the concept of hot reloading, and more.
The Building Blocks of an Angular 2 Application

In the previous chapter, we looked at the drivers for the design decisions behind Angular 2. We described the main reasons that led to the development of a brand new framework; Angular 2 takes advantage of web standards while keeping the past lessons in mind. Although we are familiar with the main drivers, we still haven't described the core Angular 2 concepts. The last major release of the framework took a different path from AngularJS 1.x and introduced a lot of changes in the fundamental building blocks used for the development of single-page applications.

In this chapter, we'll look at the framework's core and make a brief introduction to the main components of Angular 2. Another important purpose of this chapter is to take an overview of how these concepts can be put together to help us build professional user interfaces for our web applications. The following sections will give us an overview of everything that we are going to take a look at in more detail later in the book.

In this chapter, we're going to look at:

- A conceptual overview of the framework, showing how different concepts relate to each other.
- How we can build a user interface as a composition of components.
- What path the directives took in Angular 2, and how their interface changed compared to the previous major version of the framework.
- The reasons for the enforced separation of concerns, which led to the decomposition of the directives into two different components. In order to get better sense of these two concepts, we're going to demonstrate basic syntax for their definition.
The Building Blocks of an Angular 2 Application

- An overview of the improved change detection, and how it involves the context that directives provide.
- What zones are, and why they can make our daily development process easier.
- What pipes are, and how are they related to the AngularJS 1.x filters.
- The brand-new dependency injection (DI) mechanism in Angular 2 and how it is related to the service component.

A conceptual overview of Angular 2

Before we dive into different parts of Angular 2, let's get a conceptual overview of how everything fits together. Let's have a look at the following diagram:

Fig. 1 to Fig. 4 shows the main Angular 2 concepts and the connections between them. The main purpose of these diagrams is to illustrate the core blocks for building single-page applications with Angular 2, and their relations.

The Component is the main building block we're going to use to create the user interface of our applications with Angular 2. The Component is a direct successor of the Directive, which is the primitive for attaching behavior to the DOM. Components extend Directives by providing further features, such as a view with an attached template, which can be used for rendering composition of directives. Inside the template of the view can reside different expressions.
The preceding diagram illustrates conceptually the **Change Detection** mechanism of Angular 2. It runs the digest loop, which evaluates the registered expressions in the context of specific UI components. Since the concept of scope has been removed from Angular 2, the execution context of the expressions are the controllers of the components associated with them.

The **Change Detection** mechanism can be enhanced using **Differs**; that's why there's a direct relation between these two elements on the diagram.

**Pipes** are another component of Angular 2. We can think of the Pipes as the filters from AngularJS 1.x. Pipes can be used together with components. We can include them in the expressions, which are defined in the context of any component:
Now let's take a look at the preceding diagram. **Directives** and **Components** delegate the business logic to **Services**. This enforces better separation of concerns, maintainability, and reusability of code. **Directives** receive references to instances of specific services declared as dependencies using the **DI** mechanism of the framework, and delegate the execution of the business related logic to them. Both **Directives** and **Components** may use the **DI** mechanism, not only to inject services, but also to inject DOM elements and/or other **Components** or **Directives**.

![Diagram of components and router](image)

**Fig. 4**

Lastly, the component-based router is used for defining the routes in our application. Since **Directives** do not own a template, only the **Components** can be rendered by the router, representing the different views in our application. The router also uses the predefined directives, which allow us to define hyperlinks between the different views and the container where they should be rendered.

Now we're going to look more closely at these concepts, see how they work together to make Angular 2 applications, and how they've changed from their AngularJS 1.x predecessors.

### Changing directives

AngularJS 1.x introduced the concept of directives in the development of single-page applications. The purpose of directives is to encapsulate the DOM-related logic, and allow us to build user interfaces as compositions of such components by extending the syntax and the semantics of HTML. Initially, like most innovative concepts, directives were viewed controversially because they predispose us to write invalid HTML when using custom elements or attributes without the `data-` prefix. However, over time, this concept has gradually been accepted, and has proved that it is here to stay.
Another drawback of the implementation of directives in AngularJS 1.x are the different ways we can use them. This requires understanding of the attribute values, which can be literals, expressions, callbacks, or microsyntax. This makes tooling essentially impossible.

Angular 2 keeps the concept of directives, but takes the best parts from AngularJS 1.x and adds some new ideas and syntax. The main purpose of Angular 2's directives is to attach behavior to the DOM by extending it with custom logic defined in an ES2015 class. We can think of these classes as controllers associated to the directives, and think of their constructors as similar to the linking function of the directives from AngularJS 1.x. However, the new directives have limited configurability. They do not allow for the definition of a template, which makes most of the already known properties for defining directives unnecessary. The simplicity of the directives API does not limit their behavior, but only enforces stronger separation of concerns. To complement this more simple directive API, Angular 2 has introduced a richer interface for the definition of UI elements, called components. Components extend the functionality of directives by allowing them to own a template, through the Component metadata. We're going to take a further look at components later.

The syntax used for Angular 2 directives involves ES2016 decorators. However, we can also use TypeScript, ES2015, or even ECMAScript 5 (ES5) in order to achieve the same result with a little bit more typing. The following code defines a simple directive, written in TypeScript:

```typescript
@Directive({
  selector: '[tooltip]
})
export class Tooltip {
  private overlay: Overlay;
  @Input()
  private tooltip: string;
  constructor(private el: ElementRef, manager: OverlayManager) {
    this.overlay = manager.get();
  }
  @HostListener('mouseenter')
onMouseEnter() {
    this.overlay.open(this.el.nativeElement, this.tooltip);
  }
  @HostListener('mouseleave')
onMouseLeave() {
    this.overlay.close();
  }
}
```
The directive can be used with the following markup in our template:

```html
<div tooltip="42">Tell me the answer!</div>
```

Once the user points over the label, *Tell me the answer!*, Angular will invoke the method, defined under the `@HostListener` decorator in the directive's definition. In the end, the open method of the overlay manager will be executed. Since we can have multiple directives on a single element, the best practices state that we should use an attribute as a selector.

An alternative ECMAScript 5 syntax for the definition of this directive is:

```javascript
var Tooltip = ng.core.Directive({
  selector: ' [tooltip]',
  inputs: [ 'tooltip' ],
  host: {
    '(mouseenter)': 'onMouseEnter()',
    '(mouseleave)': 'onMouseLeave()'
  }
}).Class({
  constructor: [ng.core.ElementRef, Overlay, function (tooltip, el, manager) {
    this.el = el;
    this.overlay = manager.get();
  }],
  onMouseEnter() {
    this.overlay.open(this.el.nativeElement, this.tooltip);
  },
  onMouseLeave() {
    this.overlay.close();
  }
});
```

The preceding ES5 syntax demonstrates the internal JavaScript **Domain Specific Language (DSL)** that Angular 2 provides in order to allow us to write our code without the syntax, which is not yet supported by modern browsers.

We can summarize that Angular 2 has kept the concept of directives by maintaining the idea of attaching behavior to the DOM. The core differences between 1.x and 2 are the new syntax, and the further separation of concerns introduced by bringing the components. In Chapter 4, *Getting Started with Angular 2 Components and Directives*, we will take a further look at directives' API. We'll also compare the directives' definition syntax using ES2016 and ES5. Now let's have a look at the big change to Angular 2 components.
Getting to know Angular 2 components

Model View Controller (MVC) is a micro-architectural pattern initially introduced for the implementation of user interfaces. As AngularJS developers, we use different variations of this pattern on a daily basis, most often Model View ViewModel (MVVM). In MVC, we have the model, which encapsulates the business logic of our application, and the view, which is responsible for rendering the user interface, accepting user input, as well as delegating the user interaction logic to the controller. The view is represented as the composition of components, which is formally known as the composite design pattern.

Let's take a look at the following structural diagram, which shows the composite design pattern:

Here we have three classes:

- An abstract class called Component.
- Two concrete classes called Leaf and Composite. The Leaf class is a simple terminal component in the component tree that we're going to build soon.
The Component class defines an abstract operation called operation. Both Leaf and Composite inherit from the Component class. However, the Composite class also owns references to it. We can take this even further and allow Composite to own a list of references to Component instances, as shown in the diagram. The components list inside Composite can hold references to different Composite or Leaf instances, or instances of other classes, which extend the Component class or any of its successors. In the implementation of the method, operation, inside Composite, the invoked operation of the different instances inside the loop can behave differently. This is because of the late binding mechanism used for the implementation of the polymorphism in object-oriented programming languages.

Components in action

Enough of theory! Let's build a component tree based on the class hierarchy illustrated in the diagram. This way, we're going to demonstrate how we can take advantage of the composite pattern for building, user interface by using simplified syntax. We're going to take a look at a similar example in the context of Angular 2 in Chapter 4, Getting Started with Angular 2 Components and Directives:

```java
Composite c1 = new Composite();
Composite c2 = new Composite();
Composite c3 = new Composite();

  c1.components.push(c2);
  c1.components.push(c3);

  Leaf l1 = new Leaf();
  Leaf l2 = new Leaf();
  Leaf l3 = new Leaf();

  c2.components.push(l1);
  c2.components.push(l2);

  c3.components.push(l3);
```

The preceding pseudo-code creates three instances of the Composite class and three instances of the Leaf class. The instance, c1, holds references to c2 and c3 inside the components list. The instance, c2, holds references to l1 and l2, and c3 holds reference to l3:
The preceding diagram is a graphical representation of the component tree we built in the snippet. This is a quite simplified version of what the view in the modern JavaScript frameworks looks similar to. However, it illustrates the very basics of how we can compose directives and components. For instance, in the context of Angular 2 we can think of directives as instances of the preceding Leaf class (since they don't own view and thus cannot compose other directives and components), and components as instances of the Composite class.

If we think more abstractly for the user interface in AngularJS 1.x, we can notice that we use quite a similar approach. The templates of our views are composing different directives together in order to deliver fully a functional user interface to the end user of our application.

**Components in Angular 2**

Angular 2 took this approach by introducing new building blocks called **components**. Components extend the directive concept we described in the previous section, and provide broader functionality. Here is the definition of a basic hello-world component:

```typescript
@Component({
  selector: 'hello-world',
  template: '<h1>Hello, {{this.target}}!</h1>
})
class HelloWorld {
  target: string;
  constructor() {
    this.target = 'world';
  }
}
```
We can use it by inserting the following markup in our view:

```html
<hello-world></hello-world>
```

According to the best practices, we should use an element as a selector for our components since we may have only a single component per DOM element.

The alternative ES5 syntax using the DSL Angular provides is:

```javascript
var HelloWorld = ng.core.
  .Component({
    selector: 'hello-world',
    template: '<h1>Hello, {{target}}!</h1>'
  }).Class({
    constructor: function () {
      this.target = 'world';
    }
  });
```

We will take a look at the preceding syntax in more detail later in the book. However, let's briefly describe the functionality, which this component provides. Once the Angular 2 application has been bootstrapped, it will look at all the elements of our DOM tree and process them. Once it finds the element called `hello-world`, it will invoke the logic associated with its definition, which means that the template of the component will be rendered and the expression between the curly brackets will be evaluated. This will result to the markup, `<h1>Hello, world!</h1>`.

So to summarize, the Angular core team separated out the directives from AngularJS 1.x into two different parts—Components and Directives. Directives provide an easy way to attach behavior to DOM elements without defining a view. Components in Angular 2 provide a powerful, and yet simple-to-learn API, which makes it easier to define the user interface of our applications. Angular 2 Components allow us to do the same amazing things as AngularJS 1.x directives, but with less typing and fewer things to learn. Components extend the Angular 2 directive concept by adding a view to it. We can think of the relation between Angular 2 components and directives the same way as the relation between Composite and Leaf from the diagram we saw in Fig. 5.

If we start illustrating the conceptual model of the building blocks Angular 2 provides, we can present the relation between Directive and Component as inheritance. Chapter 4, Getting Started with Angular 2 Components and Directives describes these two concepts in further details.
Pipes

In business applications, we often need to have different visual representations of the same piece of data. For example, if we have the number 100,000 and we want to format it as currency, most likely we won't want to display it as plain data; more likely, we'll want something like $100,000.

The responsibility for formatting data in AngularJS 1.x was assigned to filters. Another example for a data formatting requirement is when we use collections of items. For instance, if we have a list of items, we may want to filter it based on a predicate (a boolean function); in a list of numbers, we may want to display only prime numbers. AngularJS 1.x has a filter called filter, which allows us to do this. However, the duplication of the names often leads to confusion. That's another reason the core team renamed the filter component to pipe.

The motivation behind the new name is the syntax used for pipes and filters:

```
{{expression | decimal | currency}}
```

In the preceding example, we apply the pipes, decimal and currency, to the value returned by expression. The entire expression between the curly braces looks like Unix pipe syntax.

Defining pipes

The syntax for defining pipes is similar to the one used for the definition of directives and components. In order to create a new pipe, we can use the ES2015 decorator, @Pipe. It allows us to add metadata to a class, declaring it as a pipe. All we need to do is to provide a name for the pipe and define the data formatting logic. There's also an alternative ES5 syntax, which can be used if we want to skip the process of transpilation.

During runtime, once the Angular 2 expression interpreter finds out that a given expression includes a call of a pipe, it will retrieve it out of the pipes collection allocated within the component and invoke it with the appropriate arguments.

The following example illustrates how we can define a simple pipe called lowercase1, which transforms the given string, passed as argument to its lowercase representation:

```typescript
@Pipe({ name: 'lowercase1' })
class LowerCasePipe1 implements PipeTransform {
    transform(value: string): string {
        if (!value) return value;
```

```typescript
    return value.toLowerCase();
```
The Building Blocks of an Angular 2 Application

```javascript
if (typeof value !== 'string') {
    throw new Error('Invalid pipe value', value);
}
return value.toLowerCase();
```

In order to be consistent, let's show the ECMAScript 5 syntax for defining pipes:

```javascript
var LowercasePipe1 = ng.core.Pipe(
    name: 'lowercase');
.Class({
    constructor: function () {},
    transform: function (value) {
        if (!value) return value;
        if (typeof value === 'string') {
            throw new Error('Invalid pipe value', value);
        }
        return value.toLowerCase();
    }
});
```

In the TypeScript syntax, we implement the `PipeTransform` interface and define the `transform` method declared inside it. However, in ECMAScript 5, we do not have support for interfaces, but we still need to implement the `transform` method in order to define a valid Angular 2 pipe. We are going to explain the TypeScript interfaces in the next chapter.

Now let's demonstrate how we can use the `lowercase1` pipe inside a component:

```javascript
@Component({
    selector: 'app',
    pipes: [LowercasePipe1],
    template: '<h1>{{"SAMPLE" | lowercase1}}</h1>'
})
class App {}
```

And, the alternative ECMAScript 5 syntax for this is:

```javascript
var App = ng.core.Component({
    selector: 'app',
    pipes: [LowercasePipe1],
    template: '<h1>{{"SAMPLE" | lowercase1}}</h1>'
})
```
We can use the `App` component with the following markup:

```html
<app></app>
```

The result we are going to see on the screen is the text sample within an `h1` element.

By keeping the data formatting logic as a separate component, Angular 2 keeps the strong separation of concerns that can be seen throughout. We will take a look at how we can define stateful and stateless pipes for our application in Chapter 7, *Building a real-life application while exploring pipes and http*.

### Change detection

As we saw earlier, the view in MVC updates itself, based on change events it receives from the model. A number of *Model View Whatever (MVW)* frameworks took this approach, and embedded the observer pattern in the core of their change detection mechanism.

#### Classical change detection

Let's take a look at a simple example, which doesn't use any framework. Suppose we have a model called `User`, which has a property called `name`:

```typescript
class User extends EventEmitter {
  private name: string;
  setName(name: string) {
    this.name = name;
    this.emit('change');
  }
  getName(): string {
    return this.name;
  }
}
```

The preceding snippet uses TypeScript. Do not worry if the syntax does not look familiar to you, we're going to make an introduction to the language in the next chapter.

The `user` class extends the class, `EventEmitter`. This provides primitives for emitting and subscribing to events.
Now let's define a view, which displays the name of an instance of the User class, passed as an argument to its constructor:

```javascript
class View {
    constructor(user: User, el: Element /* a DOM element */) {
        el.innerHTML = user.getName();
    }
}
```

We can initialize the view element by:

```javascript
let user = new User();
user.setName('foo');
let view = new View(user, document.getElementById('label'));
```

As the end result, the user will see a label with the content, `foo`. However, changes in user will not be reflected by the view. In order to update the view when the name of the user changes, we need to subscribe to the change event and then update the content of the DOM element. We need to update the View definition in the following way:

```javascript
class View {
    constructor(user: User, el: any /* a DOM element */) {
        el.innerHTML = user.getName();
        user.on('change', () => {
            el.innerHTML = user.getName();
        });
    }
}
```

This is how most frameworks used to implement their change detection before the era of AngularJS 1.x.

**AngularJS 1.x change detection**

Most beginners are fascinated by the data-binding mechanism in AngularJS 1.x. The basic Hello World example looks similar to this:

```javascript
function MainCtrl($scope) {
    $scope.label = 'Hello world!';
}
```

```html
<body ng-app ng-controller="MainCtrl">
    {{label}}
</body>
```
If you run this, Hello world! magically appears onto the screen. However, that is not even the most impressive thing! If we add a text input and we bind it to the label property of the scope, each change will reflect the content displayed by the interpolation directive:

```html
<body ng-controller="MainCtrl">
  <input ng-model="label">
  {{label}}
</body>
```

How awesome is that! This is one of the main selling points of AngularJS 1.x—the extreme ease of achieving data-binding. We add two (four if we count ng-controller and ng-app) attributes in our markup, add property to a mystical object called $scope, which is magically passed to a custom function we define, and everything simply works!

However, the more experienced Angular developer has a better understanding of what is actually going on behind the scene. In the preceding example, inside the directives, ng-model and ng-bind (in our case, the interpolation directive, {{}}), Angular adds watchers with different behavior associated to the same expression—label. These watchers are quite similar to the observers in the classical MVC pattern. On some specific events (in our case, change of the content of the text input), AngularJS will loop over all such watchers, evaluate the expressions associated to them in the context of a given scope, and store their results. This loop is known as the digest loop.

In the preceding examples, the evaluation of the expression, label, in the context of the scope will return the text, Hello world!. On each iteration, AngularJS will compare the current result of the evaluation with the previous result, and will invoke the associated callback in case the values differ. For instance, the callback added by the interpolation directive will set the content of the element to be the new result of the expression's evaluation. This is an example of the dependency between the callbacks of the watchers of two directives. The callback of the watcher added by ng-model modifies the result of the expression associated to the watcher added by the interpolation directive.

However, this approach has its own drawbacks. We said that the digest loop will be invoked on some specific events, but what if these events happen outside the framework, for example? What if we use setTimeout and inside the callback, passed as the first argument, we change properties attached to the scope that we're watching? AngularJS will be unaware of the change and won't invoke the digest loop, so we need to do that explicitly using $scope.$apply. But what if the framework knew about all the asynchronous events happening in the browser, such as user events, XMLHttpRequest events, WebSockets related events, and others? In such a case, AngularJS would be able to intercept the event's handling and could invoke the digest loop without forcing us to do so!
The Building Blocks of an Angular 2 Application

In the zone.js
That's exactly the case in Angular 2. This functionality is implemented with zones using `zone.js`.

At ng-conf in 2014, Brian Ford gave a talk about zones. Brian presented zones as meta-monkey patching of browser APIs. Recently Miško Hevery proposed to TC39 more mature zones API for standardization. `Zone.js` is a library developed by the Angular team, which implements zones in JavaScript. They represent an execution context, which allow us to intercept asynchronous browser calls. Basically, by using zones, we are able to invoke a piece of logic just after the given `XMLHttpRequest` completes or when we receive a new `WebSocket` event. Angular 2 took advantage of `zone.js` by intercepting asynchronous browser events and invoking the digest loop just at the right time. This totally eliminates the need of explicit calls of the digest loop by the developer using Angular.

Simplified data flow
The cross-watcher dependencies may create a tangled data flow in our application, which is hard to follow. This may lead to unpredictable behavior and bugs, which are hard to find. Although Angular 2 kept the dirty checking as a way for achieving change detection, it enforced unidirectional data flow. This happened by disallowing dependencies between the different watchers, which allows the digest loop to be run only once. This strategy increases the performance of our applications dramatically, and reduces the complexity of the data flow. Angular 2 also made improvements to memory efficiency and the performance of the digest loop. Further details on Angular 2's change detection and the different strategies used for its implementation can be found in Chapter 4, Getting Started with Angular 2 Components and Directives.

Enhancing AngularJS 1.x's change detection
Now let's take a step back and again think about the change detection mechanism of the framework.

We said that inside the digest loop, Angular evaluates registered expressions and compares the evaluated values with the values associated with the same expressions in the previous iteration of the loop.
The most optimal algorithm used for the comparison may differ depending on the type of the value returned from the expression's evaluation. For instance, if we get a mutable list of items, we need to loop over the entire collection and compare the items in the collections one by one in order to verify that there is a change or not. However, if we have an immutable list, we can perform a check with a constant complexity, only by comparing references. This is the case because the instances of immutable data structures cannot change. Instead of applying an operation, which intends to modify such instances, we'll get a new reference with the modification applied.

In AngularJS 1.x, we can add watchers using a few methods. Two of them are $watch(exp, fn, deep) or $watchCollection(exp, fn). These methods give us some level of control over the way the change detection will perform the equality check. For example, adding a watcher by using $watch and passing a false value as a third argument will make AngularJS perform a reference check (that is compare the current value with the previous one using ===). However, if we pass a truthy (any true value), the check will be deep (that is using angular.equals). This way, depending on the expected type of the returned by the expression value, we can add listeners in the most appropriate way in order to allow the framework to perform equality checks with the most optimal algorithm available. This API has two limitations:

- It does not allow you to choose the most appropriate equality check algorithm at runtime.
- It does not allow you to extend the change detection to third-parties for their specific data structures.

The Angular core team assigned this responsibility to differs, allowing them to extend the change detection mechanism and optimize it, based on the data we use in our applications. Angular 2 defines two base classes, which we can extend in order to define custom algorithms:

- KeyValueDiffer: This allows us to perform advanced diffing over key-value-based data structures.
- IterableDiffer: This allows us to perform advanced diffing over list-like data structures.

Angular 2 allows us to take full control over the change detection mechanism by extending it with custom algorithms, which wasn't possible in the previous version of the framework. We'll take a further look into the change detection and how we can configure it in Chapter 4, Getting Started with Angular 2 Components and Directives.
Understanding services

Services are the building blocks that Angular provides for the definition of the business logic of our applications. In AngularJS 1.x, we had three different ways for defining services:

```javascript
// The Factory method
module.factory('ServiceName', function (dep1, dep2, ...) {
  return {
    // public API
  }
});

// The Service method
module.service('ServiceName', function (dep1, dep2, ...) {
  // public API
  this.publicProp = val;
});

// The Provider method
module.provider('ServiceName', function () {
  return {
    $get: function (dep1, dep2, ...) {
      return {
        // public API
      }
    }
  }
});
```

Although the first two syntactical variations provide similar functionality, they differ in the way the registered directive will be instantiated. The third syntax allows further configuration of the registered provider during configuration time.

Having three different methods for defining services is quite confusing for the AngularJS 1.x beginners. Let’s think for a second what necessitated the introduction of these methods for registering services. Why can’t we simply use JavaScript constructor functions, object literals, or ES2015 classes instead, which Angular will not be aware of? We could encapsulate our business logic inside a custom JavaScript constructor function like this:

```javascript
function UserTransactions(id) {
  this.userId = id;
}
UserTransactions.prototype.makeTransaction = function (amount) {
```
This code is completely valid. However, it doesn't take advantage of one of the key features that AngularJS 1.x provides—the DI mechanism. The `MainCtrl` function uses the constructor function, `UserTransactions`, which is visible in its body. The preceding code has two main pitfalls:

- We're coupled with the logic used for the service's instantiation.
- The code is not testable. In order to mock `UserTransactions`, we need to monkey patch it.

How does AngularJS deal with these two things? When a given service is required, through the DI mechanism of the framework, AngularJS resolves all of its dependencies and instantiates it by passing them to the `factory` function, which encapsulates the logic for its creation. The `factory` function is passed as the second argument to the `factory` and `service` methods. The `provider` method allows definition of a service on lower level; the `factory` method there is the one under the `$get` property of the provider.

Just like AngularJS 1.x, Angular 2 tolerates this separation of concerns as well, so the core team kept the services. In contrast to AngularJS 1.x, the last major version of the framework provides a much simpler interface for the definition of services by allowing us to use plain ES2015 classes or ES5 constructor functions. We cannot escape from the fact that we need to explicitly state which services should be available for injection and somehow specify instructions for their instantiation. However, Angular 2 uses the ES2016 decorator's syntax for this purpose instead of the methods familiar to us from AngularJS 1.x. This allows us to define the services in our applications as simple as ES2015 classes, with decorators for configuration of the DI:

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

@Injectable()
class HttpService {
    constructor() { /* ... */ }
}

@Injectable()
class User {
```
The Building Blocks of an Angular 2 Application

```typescript
constructor(private service: HttpService) {

save() {
    return this.service.post('/users')
        .then(res => {
            this.id = res.id;
            return this;
        });
}
}
```

The alternative ECMAScript 5 syntax is:

```javascript
var HttpService = ng.core.Class({
    constructor: function () {}
});

var User = ng.core.Class({
    constructor: [HttpGetService, function (service) {
        this.service = service;
    }],
    save: function () {
        return this.service.post('/users')
            .then(function (res) {
                this.id = res.id;
                return this;
            });
    }
});
```

Services are related to the components and the directives described in the previous sections. For developing highly coherent and reusable UI components, we need to move all the business-related logic to inside our services. And, in order to develop testable components, we need to take advantage of the DI mechanism for resolving all their dependencies.

A core difference between the services in Angular 2 and AngularJS 1.x is the way their dependencies are being resolved and represented internally. AngularJS 1.x is using strings for identifying the different services and the associated factories used for their instantiation. However, Angular 2 uses keys instead. Usually the keys are the types of the distinct services. Another core difference in the instantiation is the hierarchical structure of injectors, which encapsulate different dependency providers with different visibility.
Another distinction between the services in the last two major versions of the framework is the simplified syntax. Although Angular 2 uses ES2015 classes for the definition of our business logic, you can use ECMAScript 5 constructor functions as well or use the DSL provided by the framework. The DI in Angular 2 has a completely different syntax and has improved behavior by providing a consistent way of injecting dependencies. The syntax used in the preceding example uses ES2016 decorators, and in Chapter 5, Dependency Injection in Angular 2, we’ll take a look at alternative syntax, which uses ECMAScript 5. You can also find more detailed explanation of Angular 2 services and DI in Chapter 5, Dependency Injection in Angular 2.

Understanding the new component-based router

In traditional web applications, all the page changes are associated with a full-page reload, which fetches all of the referenced resources and data and renders the entire page onto the screen. However, requirements for web applications have evolved over time.

Single-page applications (SPAs) that we build with Angular simulate desktop user experiences. This often involve incremental loading of the resources and data required by the application, and no full-page reloads after the initial page load. Often the different pages or views in SPAs are represented by different templates, which are loaded asynchronously and rendered on a specific position on the screen. Later, when the template with all the required resources is loaded and the route is changed, the logic attached to the selected page is invoked and populates the template with data. If the user presses the refresh button after the given page in our SPA is loaded, the same page needs to be re-rendered after the refresh of the view completes. This involves similar behavior—finding the requested view, fetching the required template with all referenced resources, and invoking the logic associated with that view.

What template needs to be fetched, and the logic which should be invoked after the page reloads successfully, depends on the selected view before the user pressed the refresh button. The framework determines this by parsing the page URL, which contains the identifier of the currently selected page, represented in the hierarchical structure.

All the responsibilities related to the navigation, changing the URL, loading the appropriate template, and invoking specific logic when the view is loaded are assigned to the router component. These are some quite challenging tasks, and support for different navigation APIs required for cross-browser compatibility makes the implementation of routing in modern SPAs a non-trivial problem.
AngularJS 1.x introduced the router in its core, which was later externalized into the `ngRoute` component. It allows a declarative way for defining the different views in our SPA, by providing a template for each page and a piece of logic that needs to be invoked when a page is selected. However, the functionality of the router is limited. It does not support essential features such as nested view routing. That’s one of the reasons most developers preferred to use `ui-router`, developed by the community. Both AngularJS 1.x’s router, and `ui-router`, route-definitions include a route configuration object, which defines a template and a controller associated with the page.

As described in the previous sections, Angular 2 changed the building blocks it provides for the development of SPAs. Angular 2 removes the floating controllers, and instead represents views as a composition of components. This necessitates the development of a brand new router, which empowers these new concepts.

The core differences between the AngularJS 1.x router and the Angular 2 router are:

- The Angular 2 router is component based, `ngRoute` is not.
- There is now support for nested views.
- Different syntax empowered by ES2016 decorators.

### Angular 2 route definition syntax

Let’s take a brief look at the new syntax used by the Angular 2 router to define routes in our applications:

```typescript
import {Component} from 'angular2/core';
import {bootstrap} from 'angular2/platform/browser';
import {RouteConfig, ROUTER_DIRECTIVES, ROUTER_BINDINGS} from 'angular2/router';

import {Home} from './components/home/home';
import {About} from './components/about/about';

@Component({
  selector: 'app',
  templateUrl: './app.html',
  directives: [ROUTER_DIRECTIVES]
})
@RouteConfig([{
  path: '/', component: Home, name: 'home' },
  { path: '/about', component: About, name: 'about' }
])
class App {}

bootstrap(App, [ROUTER_PROVIDERS]);
```
We won't go into too much detail here since Chapter 6, Angular 2 forms and the new component-based router and Chapter 7, Building a real-life application while exploring pipes and http, are dedicated to the new router, but let's mention the main points in the preceding code snippet.

The router lives in the module, angular2/router. There, we can find the directives it defines, the decorator used for the configuration of the routes and ROUTER_PROVIDERS.

```
We'll take a further look at ROUTER_PROVIDERS in Chapter 7, Building a real-life application while exploring pipes and http.
```

The parameter passed to the @RouteConfig decorator shows how we define the routes in our application. We use an array with objects, which defines the mappings between routes and the components associated with them. Inside the Component decorator, we explicitly state that we want to use the directives contained within ROUTER_DIRECTIVES, which are related to the router's usage inside the templates.

### Summary

In this chapter, we took a quick overview of the main building blocks for developing SPAs provided by Angular 2. We pointed out the core differences between these components in AngularJS 1.x and Angular 2.

Although we can use ES2015, or even ES5, for building Angular 2 applications, the recommendation from Google is to take advantage of the language used for the development of the framework—TypeScript.

In the next chapter, we'll take a look at TypeScript and how we can start using it in your next application. We will also explain how we can take advantage of the static typing in the JavaScript libraries and frameworks written in vanilla JavaScript, with ambient type annotations.
Where to buy this book

You can buy Switching to Angular 2 from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.