1 Introducing Angular

This chapter covers

- A high-level overview of the Angular framework
- Generating a new project with Angular CLI
- Getting started with Angular modules and components
- Introducing the sample application ngAuction

Angular is an open source JavaScript framework maintained by Google. It's a complete rewrite of its popular predecessor, AngularJS. The first version of Angular was released in September 2016 under the name Angular 2. Shortly after, the digit *2* was removed from the name, and now it's just *Angular*. Twice a year, the Angular team make major releases of this framework. Future releases will include new features, perform better, and generate smaller code bundles, but the architecture of the framework most likely will remain the same.

Angular applications can be developed in JavaScript (using the syntax of ECMAScript 5 or later versions) or TypeScript. In this book, we use TypeScript; we explain our reasons for this in appendix B.

> **NOTE** In this book, we expect you to know the syntax of JavaScript and HTML and to understand what web applications consist of. We also assume that you know what CSS is. If you're not familiar with the syntax of

TypeScript and the latest versions of ECMAScript, we suggest you read appendixes A and B first, and then continue reading from this chapter on. If you're new to developing using Node.js tooling, read appendix C.

NOTE All code samples in this book are tested with Angular 6 and should work with Angular 7 without any changes. You can download the code samples from https://github.com/Farata/angulartypescript. We provide instructions on how to run each code sample starting in chapter 2.

This chapter begins with a brief overview of the Angular framework. Then we'll start coding—we'll generate our first project using the Angular CLI tool. Finally, we'll introduce the sample application ngAuction that you'll build while reading this book.

1.1 Why select Angular for web development?

Web developers use different JavaScript frameworks and libraries, and the most popular are Angular, React, and Vue.js. You can find lots of articles and blog posts comparing them, but such comparisons aren't justified, because React and Vue.js are libraries that don't offer a full solution for developing and deploying a complete web application, whereas Angular does offer that full solution.

If you pick React or Vue.js for your project, you'll also need to select other products that support routing, dependency injection, forms, bundling and deploying the app, and more. In the end, your app will consist of multiple libraries and tools picked by a senior developer or an architect. If this developer decides to leave the project, finding a replacement won't be easy because the new hire may not be familiar with all the libraries and tools used in the project.

The Angular framework is a platform that includes all you need for developing and deploying a web app, batteries included. Replacing one Angular developer with another is easy, as long as the new person knows Angular.

From a technical perspective, we like Angular because it's a feature-complete framework that you can use to do the following right out of the box:

- Generate a new single-page web app in seconds using Angular CLI
- Create a web app that consists of a set of components that can communicate with each other in a loosely coupled manner
- Arrange the client-side navigation using the powerful router
- Inject and easily replace *services*, classes where you implement data communication or other business logic
- Arrange state management via injectable singleton services
- Cleanly separate the UI and business logic
- Modularize your app so only the core functionality is loaded on app startup, and other modules are loaded on demand
- Creating modern-looking UIs using the Angular Material library

- Implement reactive programming where your app components don't pull data that may not be ready yet, but subscribe to a data source and get notifications when data is available

Having said that, we need to admit that there is one advantage that React and Vue.js have over Angular. Although Angular is a good fit for creating single-page apps, where the entire app is developed in this framework, the code written in React and Vue.js can be included into any web app, regardless of what other frameworks were used for development of any single-page or multipage web app.

This advantage will disappear when the Angular team releases a new module currently known as `@angular/elements` (see https://github.com/angular/angular/tree/master/packages/elements). Then you'll be able to package your Angular components as custom elements (see https://developer.mozilla.org/en-US/docs/Web/Web_Components/Custom_Elements) that can be embedded into any existing web app written in JavaScript, with or without any other libraries.

1.2 Why develop in TypeScript and not in JavaScript?

You may be wondering, why not develop in JavaScript? Why do we need to use another programming language if JavaScript is already a language? You wouldn't find articles about additional languages for developing Java or C# applications, would you?

The reason is that developing in JavaScript isn't overly productive. Say a function expects a `string` value as an argument, but the developer mistakenly invokes it by passing a numeric value. With JavaScript, this error can be caught only at runtime. Java or C# compilers won't even compile code that has mismatching types, but JavaScript is a dynamically typed language and the type of a variable can be changed during runtime.

Although JavaScript engines do a decent job of guessing the types of variables by their values, development tools have a limited ability to help you without knowing the types. In mid- and large-size applications, this JavaScript shortcoming lowers the productivity of software developers.

On larger projects, good IDE context-sensitive help and support for refactoring are important. Renaming all occurrences of a variable or function name in statically typed languages is done by IDEs in a split second, but this isn't the case in JavaScript, which doesn't support types. If you make a mistake in a function or a variable name, it's displayed in red. If you pass the wrong number of parameters (or wrong types) to a function, again, the errors are displayed in red. IDEs also offer great context-sensitive help. TypeScript code can be refactored by IDEs.

TypeScript follows the latest ECMAScript specifications and adds to them types, interfaces, decorators, class member variables (fields), generics, enums, the keywords `public`, `protected`, and `private`, and more. Check the TypeScript roadmap on GitHub at https://github.com/Microsoft/TypeScript/wiki/Roadmap to see what's coming in future releases of TypeScript.

TypeScript interfaces allow you to declare custom types. Interfaces help prevent compile-time errors caused by using objects of the wrong type in your application.

The generated JavaScript code is easy to read and looks like hand-written code. The Angular framework itself is written in TypeScript, and most of the code samples in the Angular documentation (see https://angular.io), articles, and blogs are use TypeScript. In 2018, a Stack Overflow developer survey (https://insights.stackoverflow.com/survey/2018) showed TypeScript as the fourth-most-loved language. If you prefer to see more scientific proof that TypeScript is more productive compared to JavaScript, read the study "To Type or Not to Type: Quantifying Detectable Bugs in JavaScript," (Zheng Gao et al., ICSE 2017) available at http://earlbarr.com/publications/typestudy.pdf.

From the authors' real-world experience

We work for a company, Farata Systems, that over the years developed pretty complex software using the Adobe Flex (currently Apache Flex) framework. Flex is a productive framework built on top of the strongly typed, compiled ActionScript language, and the applications are deployed in the Flash Player browser plugin (a virtual machine).

When the web community started moving away from using browser plugins, we spent two years trying to find a replacement for the Flex framework. We experimented with different JavaScript-based frameworks, but the productivity of our developers seriously suffered. Finally, we saw a light at the end of the tunnel with a combination of the TypeScript language, the Angular framework, and the Angular Material UI library.

1.3 Overview of Angular

Angular is a component-based framework, and any Angular app is a tree of components (think views). Each view is represented by instances of component classes. An Angular app has one root component, which may have child components. Each child component may have its own children, and so on.

Imagine you need to rewrite the Twitter app in Angular. You could take a prototype from your web designer and start by splitting it into components, as shown in figure 1.1. The top-level component with the thick border encompasses multiple child components. In the middle, you can see a New Tweet component above two instances of the Tweet component, which in turn has child components for reply, retweet, like, and direct messaging.

A parent component can pass data to its child by binding the values to the child's component property. A child component has no knowledge of where the data came from. A child component can pass data to its parent (without knowing who the parent is) by emitting events. This architecture makes components self-contained and reusable.

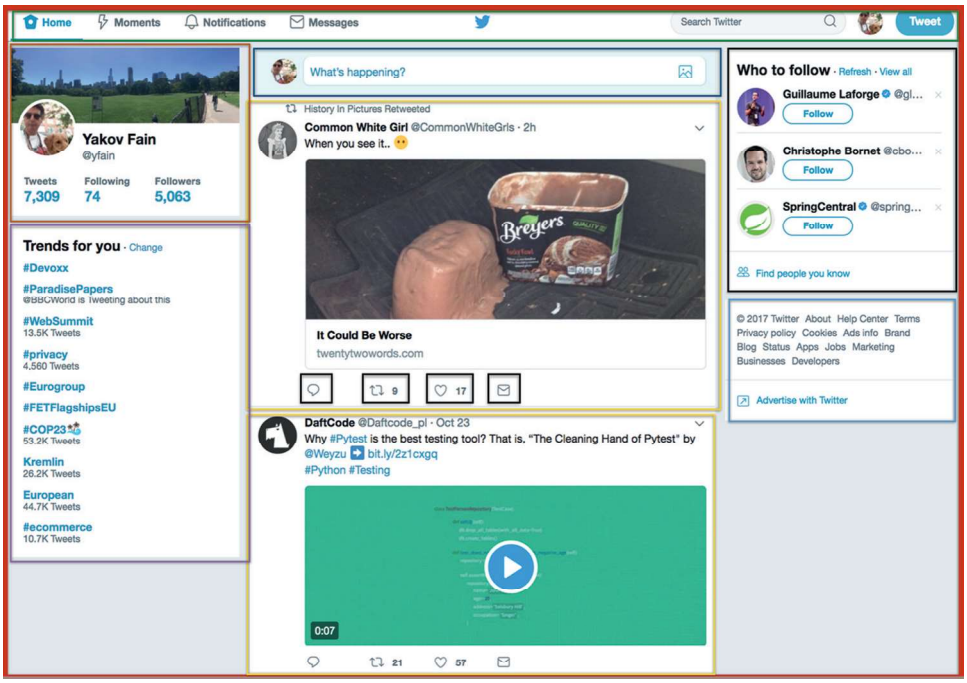


Figure 1.1 Splitting a prototype into components

When writing in TypeScript, a *component* is a class annotated with a decorator, `@Component()`, where you specify the component's UI (we explain decorators in section B.10, "Decorators," in appendix B).

```
@Component({
    ...
}
export class AppComponent {
    ...
})
```

Most of the business logic of your app is implemented in services, which are classes without a UI. Angular will create instances of your service classes and will inject them into your components. Your component may depend on services, and your services may depend on other services. A *service* is a class that implements some business logic. Angular injects services into your components or other services using the dependency injection (DI) mechanism we talk about in chapter 5.

Components are grouped into Angular modules. A *module* is a class decorated with `@NgModule()`. A typical Angular module is a small class that has an empty body, unless you want to write code that manually bootstraps the application—for example, if an app includes a legacy AngularJS app. The `@NgModule()` decorator lists all components

and other artifacts (services, directives, and so on) that should be included in this module. The following listing shows an example.

Listing 1.1 A module with one component

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

Declares that AppComponent belongs to this module

Declares that AppComponent is a root component

To write a minimalistic Angular app, you can create one `AppComponent` and list it in the `declarations` and `bootstrap` properties of `@NgModule()`. A typical module lists several components, and the root component is specified in the `bootstrap` property of the module. Listing 1.1 also lists `BrowserModule`, which is a must for apps that run in a browser.

Components are the centerpiece of the Angular architecture. Figure 1.2 shows a high-level diagram of a sample Angular application that consists of four components and two services, all packaged inside a module. Angular injects its `HttpClient` service into your app's `Service1`, which in turn is injected into the `GrandChild1` component.

The HTML template of each component is inlined either inside the component (the `template` property of `@Component()`) or in the file referenced from the component using the `templateUrl` property. The latter option offers a clean separation between the code and the UI. The same applies to styling components. You can either

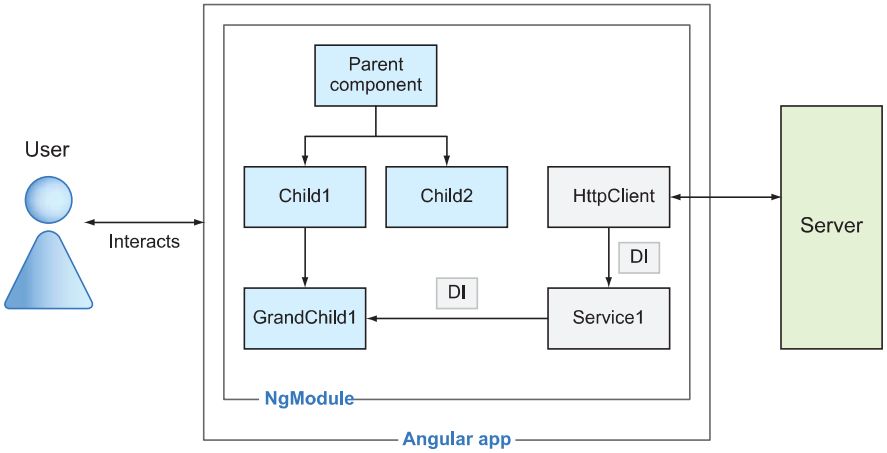


Figure 1.2 Sample architecture of an Angular app

inline the styles using the `styles` property, or provide the location of your CSS file(s) in `styleURLs`. The following listing shows the structure of some search component.

Listing 1.2 Structure of a sample component

```
@Component({
  selector: 'app-search',
   templateUrl: './search.component.html',
   styleUrls: ['./search.component.css']
 })
export class SearchComponent {
  // Component's properties and methods go here
}
```

Other components' templates can use the tag <app-search>.

The template's code is in this file.

The component's styles are in this file (there could be more than one).

The value in the `selector` property defines the name of the tag that can be used in the other component's template. For example, the root app component can include a child search component, as in the following listing.

Listing 1.3 Using the search component in the app component

```
@Component({
  selector: 'app-root',
  template: `<div>
              <app-search></app-search>
            </div>`,
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
    ...
}
```

The UI of the AppComponent includes the UI of the SearchComponent.

Listing 1.3 uses an inline template. Note the use of the backtick symbols instead of quotes for a multiline template (see section A.3 in appendix A).

The Angular framework is a great fit for developing single-page applications (SPAs), where the entire browser's page is not being refreshed and only a certain portion of the page (view) may be replacing another as the user navigates through your app. Such client-side navigation is arranged with the help of the Angular router. If you want to allocate an area within a component's UI for rendering its child components, you use a special tag, `<router-outlet>`. For example, on app start, you may display the home component in this outlet, and if the user clicks the Products link, the outlet content will be replaced by the product component.

To arrange navigation within a child component, you can allocate the `<router-outlet>` area in the child as well. Chapters 3 and 4 explain how the router works.

UI components for Angular apps

The Angular team has released a library of UI components called Angular Material (see https://material.angular.io). At the time of this writing, it has more than 30 well-designed UI components based on the Material Design guidelines (see https://material.io/guidelines). We recommend using Angular Material components in your projects, and if you need more components in addition to Angular Material, use one of the third-party libraries like PrimeNG, Kendo UI, DevExtreme, or others. You can also use the popular Bootstrap library with Angular applications, and we show how to do this in the ngAuction example in chapter 2. Starting in chapter 7, you'll rewrite ngAuction, replacing Bootstrap components with Angular Material components.

Angular for mobile devices

Angular's rendering engine is a separate module, which allows third-party vendors to create their own rendering engine that targets non-browser-based platforms. The TypeScript portion of the components remains the same, but the content of the `template` property of the `@Component` decorator may contain XML or another language for rendering native components.

For example, you can write a component's template using XML tags from the NativeScript framework, which serves as a bridge between JavaScript and native iOS and Android UI components. Another custom UI renderer allows you to use Angular with React Native, which is an alternative way of creating native (not hybrid) UIs for iOS and Android.

We stated earlier that a new Angular app can be generated in seconds. Let's see how the Angular CLI tool does it.

1.4 Introducing Angular CLI

Angular CLI is a tool for managing Angular projects throughout the entire life-cycle of an application. It serves as a code generator that greatly simplifies the process of new-project creation as well as the process of generating new components, services, and routes in an existing app. You can also use Angular CLI for building code bundles for development and production deployment. Angular CLI will not only generate a boilerplate project for you, it will also install Angular framework and all its dependencies.

Angular CLI has become a de facto way of starting new Angular projects. You'll install Angular CLI using the package manager npm. If you're not familiar with package managers, read appendix C. To install Angular CLI globally on your computer so it can be used for multiple projects, run the following command in the Terminal window:

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

After the installation is complete, Angular CLI is ready to generate a new Angular project.

1.4.1 Generating a new Angular project

CLI stands for *command-line interface*, and after installing Angular CLI, you can run the `ng` command from the Terminal window. Angular CLI understands many command-line options, and you can see all of them by running the `ng help` command . You'll start by generating a new Angular project with the `ng new` command. Create a new project called hello-cli:

```
ng new hello-cli
```

This command will create a directory, hello-cli, and will generate a project with one module, one component, and all required configuration files including the package.json file, which includes all project dependencies (see appendix C for details). After generating these files, Angular CLI will start npm to install all dependencies specified in package.json. When this command completes, you'll see a new directory, hello-cli, as shown in figure 1.3.

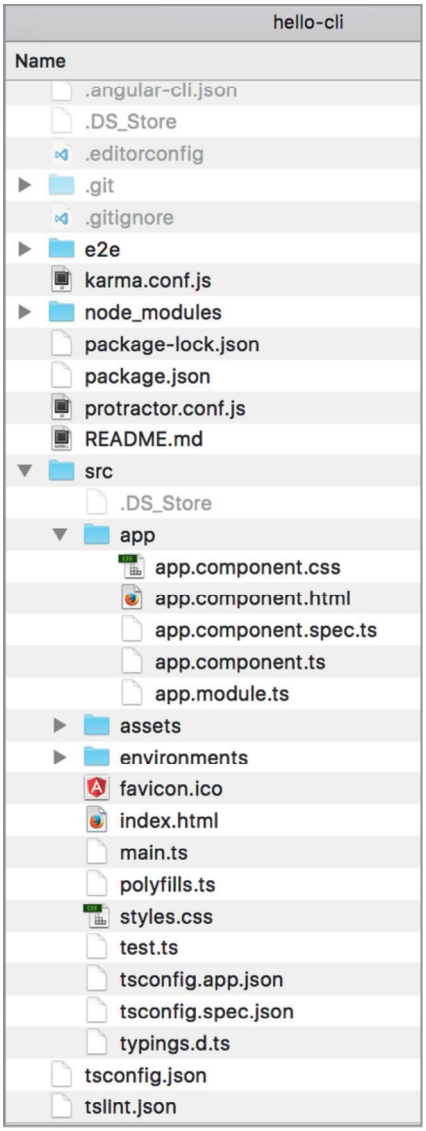


Figure 1.3 A newly generated Angular project

TIP Say you have an Angular 5 project and want to switch to the latest version of Angular. You don't need to modify dependencies in the package.json file manually. Run the `ng update` command, and all dependencies in package.json will be updated, assuming you have the latest version of Angular CLI installed. The process of updating your apps from one Angular version to another is described at https://update.angular.io.

We'll review the content of the hello-cli directory in chapter 2, but let's build and run this project. In the Terminal window, change to the hello-cli directory and run the following command:

```
ng serve
```

Angular CLI will spend about 10–15 seconds to compile TypeScript into JavaScript and build the application bundles. Then Angular CLI will start its dev server, ready to serve this app on port 4200. Your terminal output may look like figure 1.4.

```
/Users/yfain11/hello-cli
MacBook-Pro-8:hello-cli yfain11$ ng serve
** NG Live Development Server is listening on localhost:4200, open your browser
on http://localhost:4200/ **
Date: 2017-11-02T10:11:19.984Z                                              -
Hash: 6a0410d7576a15d5375e
Time: 5746ms
chunk {inline} inline.bundle.js (inline) 5.79 kB [entry] [rendered]
chunk {main} main.bundle.js (main) 20.2 kB [initial] [rendered]
chunk {polyfills} polyfills.bundle.js (polyfills) 548 kB [initial] [rendered]
chunk {styles} styles.bundle.js (styles) 33.5 kB [initial] [rendered]
chunk {vendor} vendor.bundle.js (vendor) 7.02 MB [initial] [rendered]

webpack: Compiled successfully.
```

Figure 1.4 Building the bundles with `ng serve`

Now, point your Web browser at http://localhost:4200, and you'll see the landing page of your app, as shown in figure 1.5.

Congratulations! You created, configured, built, and ran your first Angular app without writing a single line of code!

The `ng serve` command builds the bundles in memory without generating files. While working on the project, you run `ng serve` once, and then keep working on

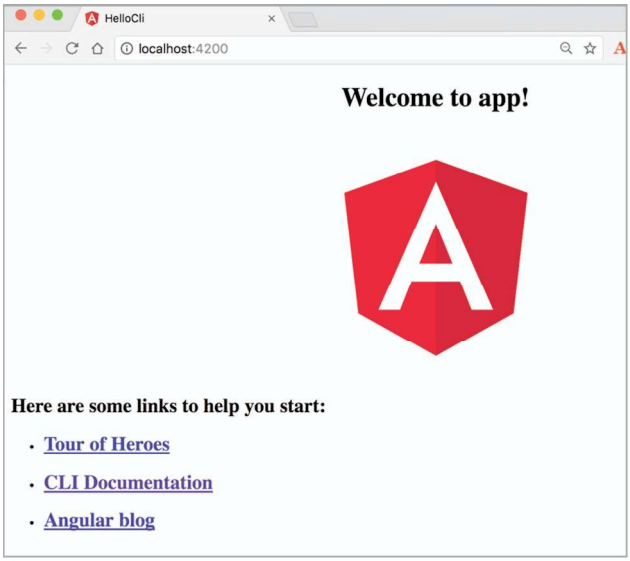


Figure 1.5 Running the app in the browser

your code. Every time you modify and save a file, Angular CLI will rebuild the bundles in memory (it takes a couple of seconds), and you'll see the results of your code modifications right away. The following JavaScript bundles were generated:

- inline.bundle.js is a file used by the Webpack loader to load other files.
- main.bundle.js includes your own code (components, services, and so on).
- polyfills.bundle.js includes polyfills needed by Angular so it can run in older browsers.
- styles.bundle.js includes CSS styles from your app.
- vendor.bundle.js includes the code of the Angular framework and its dependencies.

For each bundle, Angular CLI generates a source map file to allow debugging the original TypeScript, even though the browser will run the generated JavaScript. Don't be scared by the large size of vendor.bundle.js—it's a dev build, and the size will be substantially reduced when you build the production bundles.

Webpack and Angular CLI

Currently, Angular CLI uses Webpack (see http://webpack.js.org) to build the bundles and `webpack-dev-server` to serve the app. When you run `ng serve`, Angular CLI runs `webpack-dev-server`. Starting with Angular 7, Angular CLI offers an option to use Bazel for bundling. After the initial project build, if a developer continues working on the project, Bazel can rebuild the bundles a lot faster than Webpack.

Some useful options of ng new

When you generate a new project with the `ng new` command, you can specify an option that can change what's being generated. If you don't want to generate a separate CSS file for the application component styles, specify the inline-style option:

```
ng new hello-cli --inline-style
```

If you don't want to generate a separate HTML file for the application component template, use the inline-template option:

```
ng new hello-cli --inline-template
```

If you don't want to generate a file for unit tests, use the skip-tests option:

```
ng new hello-cli --skip-tests
```

If you're planning to implement navigation in your app, use the routing option to generate an additional module where you'll configure routes:

```
ng new hello-cli --routing
```

For the complete list of available options, run the `ng help new` command or read the Angular CLI Wiki page at https://github.com/angular/angular-cli/wiki.

1.4.2 Development and production builds

The `ng serve` command bundled the app in memory but didn't generate files and didn't optimize your Hello CLI application. You'll use the `ng build` command for file generation, but now let's start discussing bundle-size optimization and two modes of compilation.

Open the Network tab in the dev tools of your browser and you'll see that the browser had to load several megabytes of code to render this simple app. In dev mode, the size of the app is not a concern, because you run the server locally, and it takes the browser a little more than a second to load this app, as shown in figure 1.6.

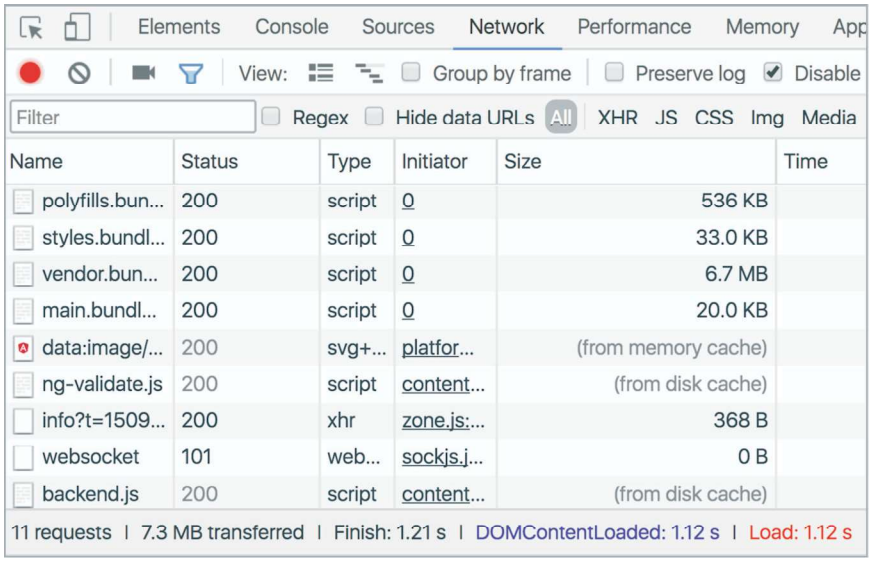


Figure 1.6 Running the non-optimized app

Now visualize a user with a mobile device browsing the internet over a regular 3G connection. It'll take 20 seconds to load the same Hello CLI app. Many people can't tolerate waiting 20 seconds for any app except Facebook (30% of the earth's population lives on Facebook). You need to reduce the size of the bundles before going live.

Applying the `--prod` option while building the bundles will produce much smaller bundles (as shown in figure 1.6) by optimizing your code. It'll rename your variables as single letters, remove comments and empty lines, and remove the majority of the unused code. Another piece of code that can be removed from app bundles is the Angular compiler. Yes, the `ng serve` command included the compiler into the vendor.bundle.js. But how are you going to remove the Angular compiler from your deployed app when you build it for production?

1.5 JIT vs. AOT compilation

Let's revisit the code of app.component.html. For the most part, it consists of standard HTML tags, but there's one line that browsers won't understand:

```
Welcome to {{title}}!
```

These double curly braces represent binding a value into a string in Angular, but this line has to be compiled by the Angular compiler (it's called `ngc`) to replace the binding with something that browsers understand. A component template can include other Angular-specific syntax (for example, structural directives `*ngIf` and `*ngFor`) that needs to be compiled before asking the browser to render the template.

When you run the `ng serve` command, the template compilation is performed inside the browser. After the browser loads your app bundles, the Angular compiler (packaged inside vendor.bundle.js) performs the compilation of the templates from main.bundle.js. This is called *just-in-time* (JIT) compilation. This term means that the compilation happens when the bundles arrive at the browser.

The drawbacks of JIT compilation include the following:

- There's an interval of time between loading bundles and rendering the UI. This time is spent on JIT compilation. For a small app like Hello CLI, this time is minimal, but in real-world apps, JIT compilation can take a couple of seconds, so the user needs to wait longer before seeing your app.
- The Angular compiler has to be included in vendor.bundle.js, which adds to the size of your app.

Using JIT compilation in production is discouraged, and you want templates to be precompiled into JavaScript before the bundles are created. This is what *ahead-of-time* (AOT) compilation is about.

The advantages of AOT compilation are as follows:

- The browser can render the UI as soon as your app is loaded. There's no need to wait for code compilation.
- The `ngc` compiler isn't included in vendor.bundle.js, and the resulting size of your app might be smaller.

Why use the word *might* and not *will*? Removing the `ngc` compiler from the bundles should always result in smaller app size, right? Not always. The compiled templates are larger than those that use a concise Angular syntax. The size of Hello CLI will definitely be smaller, as there's only one line to compile. But in larger apps with lots of views, the compiled templates may increase the size of your app so that it's even larger than the JIT-compiled app with `ngc` included in the bundle. You should use the AOT mode anyway, because the user will see the initial landing page of your app sooner.

NOTE You may be surprised by seeing `ngc` compiler errors in an app that was compiling fine with `tsc`. The reason is that AOT requires your code to be statically analyzable. For example, you can't use the keyword `private` with properties

that are used in the template, and no default exports are allowed. Fix the errors reported by the `ngc` compiler and enjoy the benefits of AOT compilation.

No matter whether you choose JIT or AOT compilation, at some point you'll decide to do an optimized production build. How do you do this?

1.5.1 Creating bundles with the –prod option

When you build bundles with the `--prod` option, Angular CLI performs code optimization and AOT compilation. See it in action by running the following command in your Hello CLI project:

```
ng serve --prod
```

Open the app in your browser and check the Network tab, as shown in figure 1.7. Now the size of the same app is only 108 KB gzipped.

Expand the column with the bundle sizes—the dev server even did the gzip compression for you. The filenames of the bundles include a hash code of each bundle. Angular CLI calculates a new hash code on each production build to prevent browsers from using the cached version if a new app version is deployed in prod.

Shouldn't you always use AOT? Ideally, you should unless you use some third-party JavaScript libraries that produce errors during AOT compilation. If you run into this problem, turn AOT compilation off by building the bundles with the following command:

```
ng serve --prod --aot false
```

Figure 1.8 shows that both the size and the load time increased compared to the AOT-compiled app in figure 1.7.

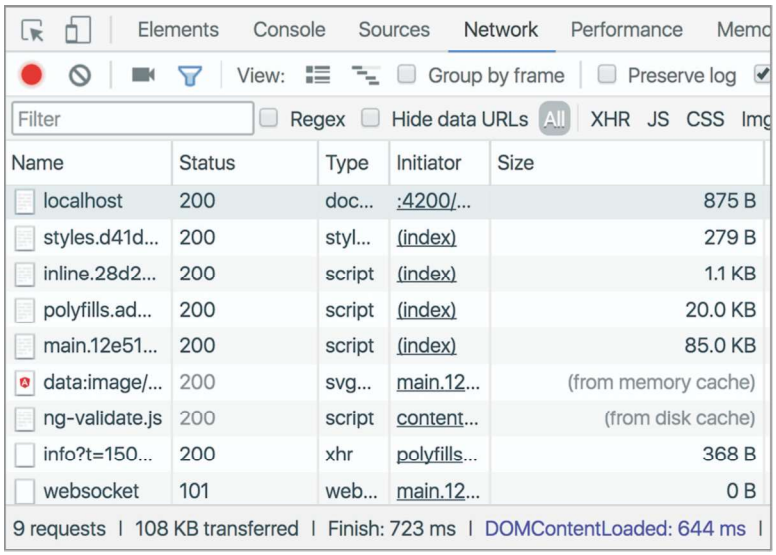


Figure 1.7 Running the optimized app with AOT

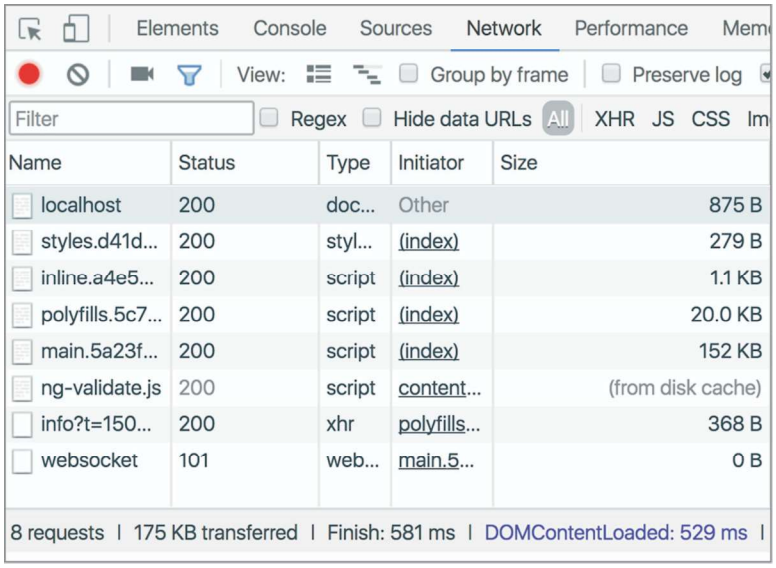


Figure 1.8 Running the optimized app without AOT

1.5.2 Generating bundles on the disk

You were using the `ng serve` command, which was building the bundles in memory. When you're ready to generate production files, use the `ng build` command instead. The `ng build` command generates files in the dist directory (by default), but the bundle sizes won't be optimized.

With `ng build --prod`, the generated files will be optimized but not compressed, so you'd need to apply the gzip compression to the bundles afterward. We'll go over the process of building production bundles and deploying the app on the Node.js server in section 12.5.3 of chapter 12.

After the files are built in the dist directory, you can copy them to whatever web server you use. Read the product documentation for your web server, and if you know where to deploy an index.html file in your server, this would be the place for the Angular app bundles as well.

The goal of this section was to get you started with Angular CLI, and we'll continue its coverage in chapter 2. The first generated app is rather simple and doesn't illustrate all the features of Angular; the next section will give you some ideas of how things are done in Angular.

1.6 Introducing the sample ngAuction app

To make this book more practical, we start every chapter by showing you small applications that illustrate Angular syntax or techniques, and at the end of most of the chapters you'll use the new concepts in a working application. You'll see how components and services are combined into a working application.

Imagine an online auction (let's call it ngAuction) where people can browse and search for products. When the results are displayed, the user can select a product and bid on it. The information on the latest bids will be pushed by the server to all users subscribed to such notifications.

The functionality of browsing, searching, and placing bids will be implemented by making requests to the RESTful endpoints, implemented in the server developed with Node.js. The server will use WebSockets to push notifications about the user's bid and about the bids placed by other users. Figure 1.9 depicts sample workflows for ngAuction.

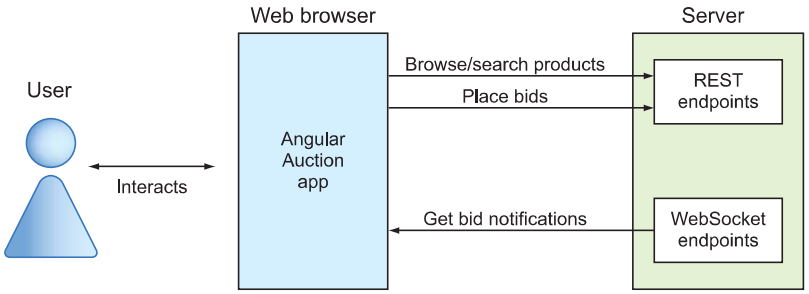


Figure 1.9 The ngAuction workflows

Figure 1.10 shows how the first version of the ngAuction home page will be rendered on desktop computers. Initially, you'll use gray placeholders instead of product images.

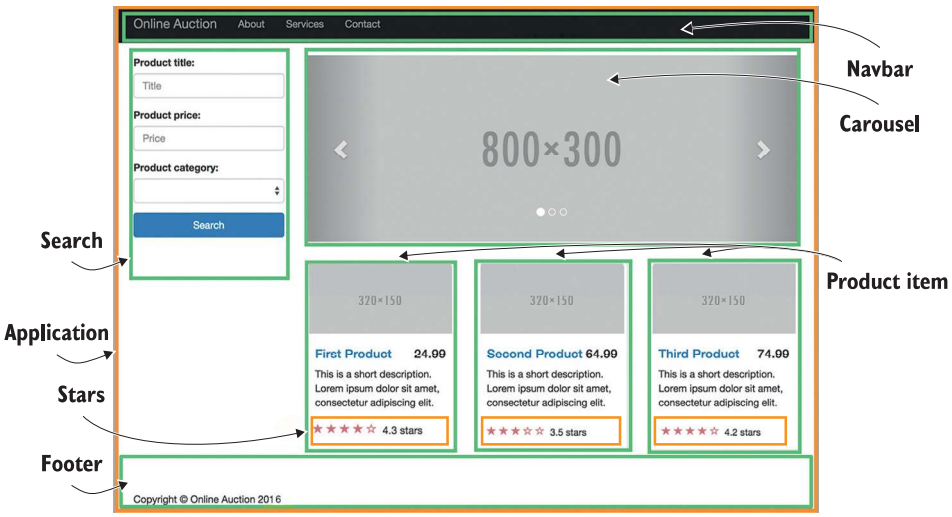


Figure 1.10 The ngAuction home page with highlighted components

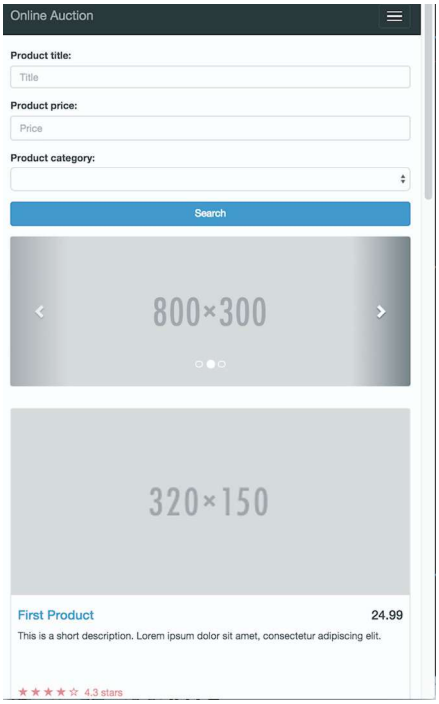


Figure 1.11 The online auction home page on smartphones

You'll use responsive UI components offered by the Bootstrap library (see http://getbootstrap.com), so on mobile devices the home page may be rendered as in figure 1.11.

Starting in chapter 7, you'll redesign ngAuction to completely remove the Bootstrap framework, replacing it with the Angular Material and Flex Layout libraries. The home page of the refactored version of ngAuction will look like figure 1.12.

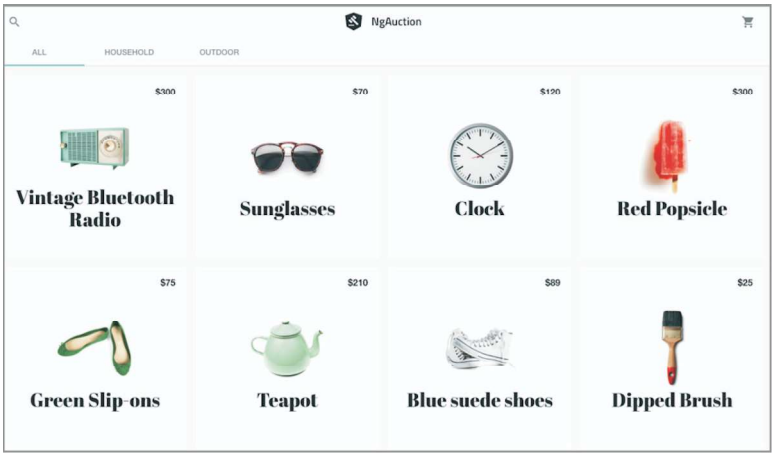


Figure 1.12 The redesigned ngAuction

The development of an Angular application comes down to creating and composing components. In chapter 2 you'll generate this project and its components and services using Angular CLI, and in chapter 7, you'll refactor its code. Figure 1.13 shows the project structure for the ngAuction app.

In chapter 2, you'll start by creating an initial version of the landing page of ngAuction, and in subsequent chapters, you'll keep adding functionality that illustrates various Angular features and techniques

NOTE We recommend that you develop Angular applications using an IDE like WebStorm (inexpensive) or Visual Studio Code (free). They offer the autocomplete feature, provide convenient search, and have integrated Terminal windows so you can do all your work inside the IDE.

Summary

- Angular applications can be developed in TypeScript or JavaScript.
- Angular is a component-based framework.
- The TypeScript source code has to be transpiled into JavaScript before deployment.
- Angular CLI is a great tool that helps in jump-starting your project. It supports bundling and serving your apps in development and preparing production builds.

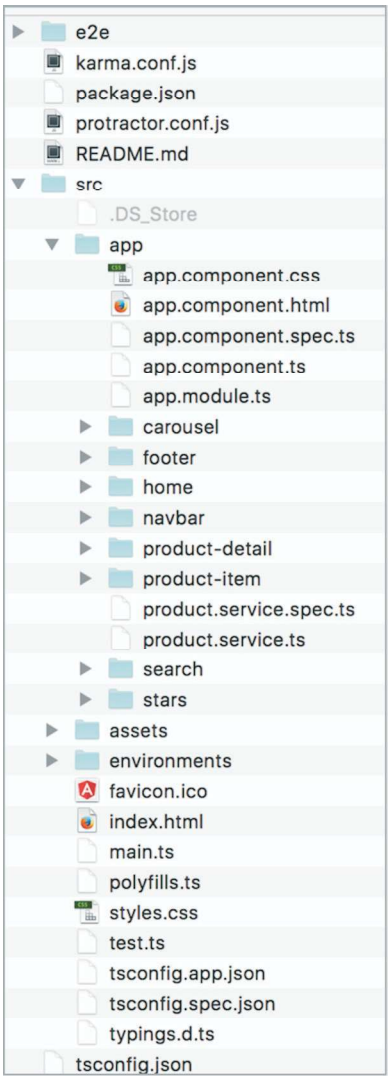


Figure 1.13 The project structure for the online auction app

2 The main artifacts of an Angular app

This chapter covers

- Understanding components, directives, services, modules, and pipes
- Generating components, directives, services, and routes with Angular CLI
- Looking at Angular data bindings
- Building the first version of the ngAuction app

In this chapter, we'll start by explaining the roles of the main artifacts of Angular applications. We'll introduce you to each one and show how Angular CLI can generate these artifacts. Then we'll do an overview of the Angular CLI configuration file where you can modify your project settings.

After that, we'll discuss how data binding is implemented in Angular. At the end of the chapter, you'll develop the initial version of the ngAuction application that you'll continue working on throughout the book.

2.1 Components

A component, the main artifact of any Angular app, is a class with a view (UI). To turn a class into component, you need to decorate it with the `@Component()` decorator. A component can consist of one or more files—for example, a file with the extension .ts with the component class, a .css file with styles, an .html file with a template, and a .spec.ts file with test code for the component.

You don't have to split the code of each component into these files. For example, you can have a component in just one file with inline styles and templates and no files with tests. No matter how many files will represent your component, they all should be located in the same directory.

Any component belongs to exactly one module of an app, and you have to include the name of the component's class into the `declarations` property of the `@NgModule()` decorator in the module file. In chapter 1, we already had an `AppComponent` listed in the `AppModule`.

The ng generate command of Angular CLI

Even after generating a new project, you can keep using Angular CLI for generating artifacts by using the `ng generate` command or its alias `ng g`. Here are some options you can use with the `ng g` command:

- `ng g c`—Generates a new component.
- `ng g s`—Generates a new service.
- `ng g d`—Generates a new directive.
- `ng g m`—Generates a new module.
- `ng g application`—Generates a new app within the same project. This command was introduced in Angular 6.
- `ng g library`—Starting with Angular 6, you can generate a library project. This is not an app, but can include services and components as well.

Each of these commands requires an argument, such as the name of the item, to generate. For a complete list of available options and arguments, run the `ng help generate` command or refer to the Angular CLI documentation.

Here are some examples of using the `ng g` command:

- `ng g c product` will generate four files for a new product component in the src/app/product directory and add the `ProductComponent` class to the `declarations` property of `@NgModule`.
- `ng g c product -is --it --spec false` will generate a single file, product.component.ts, with inlined styles and template and no test in the src/app/product directory, and add `ProductComponent` to the `declarations` property of `@NgModule`.

- `ng g s product` will generate the file product.service.ts containing a class decorated with `@Injectable` and the file product.service.spec.ts in the src/app directory.

`ng g s product -m app.module` will generate the same files as the preceding command and will also add `ProductService` to the `providers` property of `@NgModule`.

Let's add a product component to the Hello CLI project you created in chapter 1 by running the following command in the root directory of that project:

```
ng g c product --is --it --spec false
```

This command will create the src/app/product directory with the product.component .ts file.

Listing 2.1 product.component.ts

```
@Component({
  selector: 'app-product',
  template: `
  <p>
  product Works!  <------ A default text to render
   </p>
  `,
  styles: []
})
export class ProductComponent implements OnInit {   <------ Implementing OnInit requires the ngOnInit() method in the class.

  constructor() { }

  ngOnInit() {}  <------ A lifecycle method
 }
```

The generated `@Component()` decorator has the `app-product` selector, the `template` property with inlined HTML, and the `styles` property for inline CSS. Other components can include your product component in their templates by using the `<app-product>` tag.

The class in listing 2.1 has an empty constructor and one method, `ngOnInit ()`, which is one of the component lifecycle methods. If implemented, `ngOnInit()` is invoked after the code in the constructor. `OnInit` is one of the lifecycle interfaces that require the implementation of `ngOnInit()`. We'll cover a component lifecycle in section 9.2 in chapter 9.

NOTE The `@Component()` decorator has a few more properties that we'll discuss when it comes time to use them. All properties of the `@Component()` decorator are described at https://angular.io/api/core/Component.

Using selector prefixes

The selector of the component in listing 2.1 has the prefix `app-`, which is a default prefix for apps. For library projects, the default prefix is `lib-`. A good practice is to come up with a more specific prefix that would better identify your application. Your project is called Hello CLI, so you may want to give the `hello-` prefix to all your components. To do this, use the `-prefix` option while generating your components:

```
ng g c product -prefix hello
```

That command would generate a component with the `hello-product` selector . An easier way to ensure that all generated components have a specific prefix is to specify the prefix in the file .angular-cli.json (or in angular.json, starting with Angular 6), discussed later in this chapter.

If you open the app.module.ts file, you'll see that `ProductComponent` has been imported and added to the `declarations` section by your `ng g c` command:

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

The newly generated `ProductComponent` class was added to the `declarations` property of `@NgModule()`. We'll keep using components in each chapter of this book so you'll get a chance to learn the various features of Angular components.

What's metadata?

TypeScript decorators allow you to modify the behavior of a class, property, or method (or its arguments) without changing their code by enabling you to provide metadata about the decorated artifact. In general, *metadata* is additional information about data. For example, in an MP3 file, the audio is the data, but the name of the artist, song title, and album cover are metadata. MP3 players include metadata processors that read the metadata and display some of it while playing the song.

In the case of classes, metadata refers to additional information about the class. For example, the `@Component()` decorator tells Angular (the metadata processor) that this is not a regular class, but a component. Angular generates additional JavaScript code based on the information provided in the properties of the `@Component()`

nsdecorator to turn a class into a UI component. The `@Component()` decorator doesn't change the internals of the decorated class but adds some data describing the class so the Angular compiler can properly generate the final code of the component.

In the case of class properties, the `@Input()` decorator tells Angular that this class property should support binding and be able to receive data from the parent component. You'll learn about the `@Input()` decorator in section 8.2.1 in chapter 8.

Under the hood, a decorator is a function that attaches some data to the decorated element. See section B.10 in appendix B for more details.

A component is a class with a UI, and a service is a class where you implement the business logic of your app. Let's get familiar with services.

2.2 Services

For cleaner code separation, we usually don't use a component for code that fetches or manipulates data. An injectable service is the right place for handling data. A component may depend on one or more services. Chapter 5 covers how dependency injection works in Angular. Here, we'll give you an idea of how services and components work together.

Let's start with generating a service in a shared folder, assuming that this service will be used by multiple components. To ensure that the `providers` property of `@NgModule()` will be updated with the newly generated service, use the following option:

```
ng g s shared/product -m app.module
```

The new file product.service.ts will be generated in the src/app/shared directory:

```
@Injectable()
export class ProductService {
  constructor() { }
}
```

Accordingly, the app.module.ts file will be updated to include the provider for this service:

```
@NgModule({
...
  providers: [ProductService]
})
export class AppModule { }
```

Next, implement some methods in `ProductService` with the required business logic. Note that the generated `ProductService` class is annotated with the `@Injectable()` decorator. To have Angular instantiate and inject this service into any component, add the following argument to the component's constructor:

```
constructor(productService: ProductService){
  // start using the service, e.g. productService.getMyData();
}
```

A service isn't the only artifact without a UI. Directives also don't have their own UI, but they can be attached to the UI of components.

> **What's new in Angular 6**
>
> Starting from Angular 6, the `ng g s` command generates a class with the `Injectable()` decorator:
>
> ```
> @Injectable({
>   provideIn: 'root'
> })
> ```
>
> `provideIn: 'root'` allows you to skip the step of specifying the service in the `providers` property of the `NgModule()` decorator.

2.3 Directives

Think of an Angular directive as an HTML enhancer. Directives allow you to teach an old HTML element new tricks. A *directive* is a class annotated with the `@Directive()` decorator. You'll see the `@Directive()` decorator used in section 11.7 in chapter 11.

A directive can't have its own UI, but can be attached to a component or a regular HTML element to change their visual representation. There are two types of directives in Angular:

- *Structural*—The directive changes the structure of the component's template.
- *Attribute*—The directive changes the behavior or visual representation of an individual component or HTML element.

For example, with the structural `*ngFor` directive, you can iterate through an array (or other collection) and render an HTML element for each item of the array. The following listing uses the `*ngFor` directive to loop through the `products` array and render an `<li>` element for each product (assuming there's an interface or `Product` class with a `title` property).

Listing 2.2 Iterating through products with `*ngFor`

```
@Component({
  selector: 'app-root',
  template: `<h1>All Products</h1>
  <ul>
    <li *ngFor="let product of products">
      {{product.title}}
    </li>
  </ul>
  `})
```

Renders <li> for each product

Each <li> element shows the product title.

```
export class AppComponent {
  products: Product[] = [];

 // the code to populate products is removed for brevity
}
```

*ngFor iterates through this array.

The following element uses the structural `*ngIf` directive to either show or hide the `<mat-error>` element, depending on the return (true or false) of the `hasError()` method, which checks whether the value in a form field `title` has invalid minimum length:

```
<mat-error *ngIf="formModel.hasError('minlength', 'title')" >Enter at least 3
     characters</mat-error>
```

Later in this chapter when we talk about two-way binding, we'll use the attribute `ngModel` directive to bind the value in the `<input>` element to a class variable, `shippingAddress`:

```
<input type='text' placeholder="Enter shipping address"
   [(ngModel)]="shippingAddress">
```

You can create custom attribute directives as well, described in the product documentation at https://angular.io/guide/attribute-directives.

Yet another artifact that doesn't have its own UI but that can transform values in the component template is a pipe.

2.4 Pipes

A *pipe* is a template element that allows you to transform a value into a desired output. A pipe is specified by adding the vertical bar (|) and the pipe name right after the value to be transformed:

```
template: `<p>Your birthday is {{ birthday | date }}</p>`
```

In the preceding example, the value of the `birthday` variable will be transformed into a date of a default format. Angular comes with a number of built-in pipes, and each pipe has a class that implements its functionality (for example, `DatePipe`) as well as the name that you can use in the template (such as `date`):

- An `UpperCasePipe` allows you to convert an input string to uppercase by using | `uppercase` in the template.
- A `LowerCasePipe` allows you to convert an input string to lowercase by using | `lowercase` in the template.
- A `DatePipe` allows you to display a date in different formats by using | `date`.
- A `CurrencyPipe` transforms a number into a desired currency by using | `currency`.
- An `AsyncPipe` unwraps the data from the provided `Observable` stream by using | `async`. You'll see a code sample that uses `async` in section 6.5 in chapter 6.

Some pipes don't require input parameters (such as `uppercase`), and some do (such as `date:'medium'`). You can chain as many pipes as you want. The following code snippet shows how you can display the value of the `birthday` variable in a medium date format and in uppercase (for example, JUN 15, 2001, 9:43:11 PM):

```
template=
  `<p>{{ birthday | date: 'medium' | uppercase}}</p>`
```

As you can see, with literally no coding you can convert a date into the required format as well as show it uppercase (see the date formats in the Angular `DatePipe` documentation, http://mng.bz/78lD).

In addition to predefined pipes, you can create custom pipes that can include code specific to your application. The process of creating custom pipes is described at https://angular.io/guide/pipes. Code samples for this chapter include an app demonstrating a custom pipe that can convert the temperature from Fahrenheit to Celsius and back.

Now you know that your app can include components, services, directives, and pipes. All these artifacts must be declared in your app module(s).

2.5 Modules

An Angular module is a container for a group of related components, services, directives, and pipes. You can think of a module as a package that implements certain functionality from the business domain of your application, such as a shipping or billing module. All elements of a small application can be located in one module (the root module), whereas larger apps may have more than one module (feature modules). All apps must have at least a root module that's bootstrapped during app launch.

From a syntax perspective, an Angular module is a class annotated with the `@NgModule()` decorator. To load the root module on application startup, invoke the `bootstrapModule()` method in the main.ts file of your app:

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

The Angular framework itself is split into modules. Including some of the Angular modules is a must (for example, `@angular/core`), whereas some modules are optional. For example, if you're planning to use the Angular Forms API and make HTTP requests, you should add `@angular/forms` and `@angular/common/http` in your package.json file and should include `FormsModule` and `HttpClientModule` in the root module of your app, as shown in the following listing.

Listing 2.3 A sample root module

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
     BrowserModule,
```

The only component included in this module

Other modules that are needed for this app

```
    FormsModule,
    HttpClientModule
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

The top-level component to be loaded on app startup

If you decide to split your app into several modules, in addition to the root module you'll need to create *feature modules*, covered next.

2.5.1 Feature modules

An Angular app may consist of a root module and feature modules. You can implement a certain feature of your app (for example, shipping) in a feature module. Whereas the `@NgModule()` decorator of the root module of a web application must include the `BrowserModule`, feature modules include the `CommonModule` instead. Feature modules can be imported by other modules. The `@NgModule()` decorator of feature modules doesn't include the `bootstrap` property, because bootstrapping the entire app is the responsibility of the root module.

The following listing generates a small app called Hello Modules and adds a feature module called `ShippingModule` to it.

Listing 2.4 Generating a project and a feature module

```
ng new hello-modules
 cd hello-modules
ng g m shipping
```

Generates a new project called hello-modules

Generates a new feature module called shipping

This app will have a feature module with the following listing's content in the file src/app/shipping/shipping.module.ts.

Listing 2.5 Generated feature module

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

Feature module imports CommonModule instead of the BrowserModule

Now let's generate a new shipping component, instructing Angular CLI to include it into `ShippingModule`:

```
ng g c shipping -it -is -m shipping
```

This command generates the file shipping/shipping.component.ts with the decorated class `ShippingComponent` with an inline template and an empty `styles` property. The

command also adds it to the `declarations` section of the `ShippingModule`. The code for the shipping component is shown in the following listing.

Listing 2.6 Generated shipping component

```
@Component({
  selector: 'app-shipping',
  template: `
  <p>
  shipping Works!  <-- A default template
   </p>
  `,
  styles: []
})
export class ShippingComponent implements OnInit {  <-- Implementing the lifecycle interface OnInit

  constructor() { }

  ngOnInit() {}      <-- This lifecycle hook is invoked after the constructor.
 }
```

Note the selector of the shipping component: `app-shipping`. You'll be using this name in the template of the `AppComponent`.

The code for your shipping module will include the shipping component and will look like the following listing.

Listing 2.7 Generated shipping module

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

A feature module may declare its own components and services, but to make all or some of them visible to other modules, you need to export them. In this case, you need to add an `exports` section to the shipping module so it looks as follows.

Listing 2.8 Exporting a shipping component

```
@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [ShippingComponent],
  exports: [ShippingComponent]      <-- Exporting a component from the module
 })
export class ShippingModule { }
```

External modules will see only those members of the shipping module that were explicitly mentioned in `exports`. The shipping module may include other members, like classes, directives, and pipes. If you don't list them in the `exports` section, these members will remain private to the shipping module and will be hidden from the rest of the app. Now you should include the shipping module in the root module.

Listing 2.9 Adding the shipping module to the root module

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

Adds the shipping module to the root module

To have the browser render the shipping component in the root component, you can add the `<app-shipping>` tag to the template of the `AppComponent`.

Listing 2.10 Adding a shipping component

```
@Component({
  selector: 'app-root',
  template: `
  <h1>Welcome to {{title}}!!</h1>
  <app-shipping></app-shipping>
   `,
  styles: []
})
export class AppComponent {
  title = 'app';
}
```

Adds the ShippingComponent to the root component's template

Run this app with `ng serve` and open the browser at http://localhost:4200. You'll see the window that renders the `AppComponent` from the root module and the `ShippingComponent` from the shipping module, as shown in figure 2.1.

"Welcome to app!!" is the greeting from the `AppComponent` located in the root module, whereas "shipping Works!" comes from the `ShippingComponent` located in your feature module. This was a rather simple example,

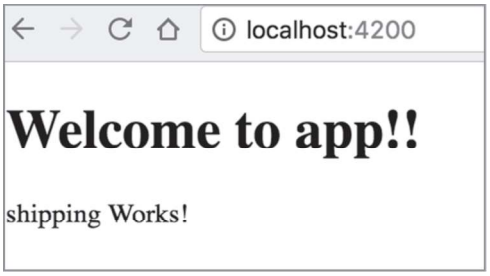


Figure 2.1 Running the two-module app

but it illustrates how you can *modularize* an app so its specific features are located in a separate reusable module and can be developed by a separate developer(s).

Your app modules can be loaded either eagerly on application startup, as was done with the Hello Modules project, or lazily, such as when the user clicks a specific link. In section 4.3 in chapter 4, you'll see a sample app with a module that's lazy loaded.

You know by now that a component consists of TypeScript code and the UI (the template). The next concept to learn is how the code and the UI can be synchronized as the data changes either programmatically or as the result of the user's interaction with the app.

2.6 Understanding data binding

Angular has a mechanism called *data binding* that allows you to keep a component's properties in sync with the view. In this section, we'll explain how data binding works with properties and events.

Angular supports two types of data binding: *unidirectional* (default) and *two-way*. With unidirectional data binding , the data is synchronized in one direction: either from the class member variable (property) to the view or from the view event to the class variable or a method that handles the event. Angular updates the binding during its change detection cycle, explained in section 9.1 in chapter 9.

2.6.1 Binding properties and events

To display a value of a class variable in a template's string, use double curly braces. If a class has a variable called `name`, you can show its value like this:

```
<h1>Hello {{name}}!</h1>
```

This is also known as *interpolation* , and you can use any valid expression inside these double curly braces.

Use square brackets to bind the value from a class variable to a property of an HTML element or an Angular component. The following binds the value of the class variable `isValid` to the `hidden` property of the HTML `<span>` element:

```
<span [hidden]="isValid">This field is required</span>
```

Note that the square brackets are used to the left of the equals sign. If the value of the `isValid` variable is false, the text of the `span` element isn't hidden, and the user will see the message "This field is required." As soon as the value of the `isValid` variable becomes true, the text "This field is required" becomes hidden.

The preceding examples illustrate unidirectional binding from the class variable to the view. The next listings will illustrate the unidirectional binding from the view to a class member, such as a method.

To assign an event-handler function to an event, put the event name in parentheses in the component's template. The following listing shows how to bind the `onClickEvent()` function to the `click` event, and the `onInputEvent()` function to the `input` event.

Listing 2.11 Two events with handlers

```
<button (click)="onClickEvent()">Get Products</button>
<input placeholder="Product name" (input)="onInputEvent()">
```

If the button is clicked, invoke the method onClickEvent().

As soon as the value of the input field changes, invoke the method onInputEvent().

When the event specified in parentheses is triggered, the expression in double quotes is reevaluated. In listing 2.11, the expressions are functions, so they're invoked each time the corresponding event is triggered.

If you're interested in analyzing the properties of the event object, add the `$event` argument to the method handler. In particular, the target `property` of the event object represents the DOM node where the event occurred. The instance of the event object will be available only within the binding scope (that is, in the event-handler method). Figure 2.2 shows how to read the event-binding syntax.

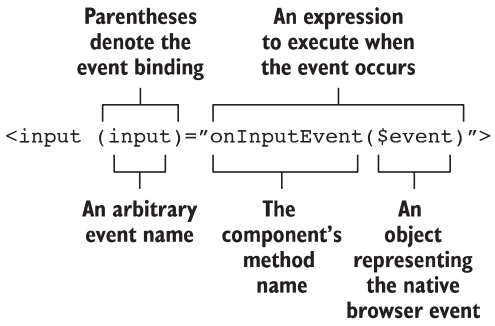


Figure 2.2 Event-binding syntax

The event in parentheses is called the *target of binding*. You can bind functions to any standard DOM events that exist today (see "Event reference" in the Mozilla Developer Network documentation, http://mzl.la/1JcBR22) or that will be introduced in the future.

Parentheses are used for binding to the standard DOM events as well as the custom events of a component. For example, say you have a price-quoter component that can emit a custom event, `lastPrice`. The following code snippet shows how to bind the `lastPrice` event to the class method `priceQuoteHandler()`:

```
<price-quoter (lastPrice)="priceQuoteHandler($event)"> </price-quoter>
```

You'll learn how to create components that emit custom events in chapter 8, section 8.2.2.

2.6.2 One- and two-way data binding in action

Let's run and review two simple apps from the project bindings that come with this chapter. If you use the Angular 5 code samples, the two apps `oneway` and `twoway` are configured by creating two elements in the `apps` array in the file .angular-cli.json. If you use the Angular 6 code samples, these two apps are configured in the file angular.json.

> **Configuring Angular CLI projects**
>
> Prior to Angular 6, the generated projects include the configuration file .angular-cli.json, which allows you to specify where the source code is located, which directory will contain the compiled code, where to find the assets of your project, where the code and styles required by third-party libraries (if any) are, and more. Angular CLI uses properties of this file during generation of the your app artifacts, during builds, and while running tests.
>
> You can find the complete and current description of each config property in the document "Angular CLI Config Schema," available at https://github.com/angular/angular-cli/wiki/angular-cli. You'll use the `apps` config property in this section and the `styles` and `scripts` properties in section 2.7.
>
> Starting in Angular 6, projects are configured in the angular.json file, and its schema is described at https://github.com/angular/angular-cli/wiki/angular-workspace. Now the project is treated as a workspace, which can contain one or more apps and libraries with their own configurations, but all of them share the dependencies located in a single directory: node_modules.

These two apps will be configured similarly—only the app names and the names of the files that boot these apps will differ, as shown in the following listing.

Listing 2.12 Angular 5: Configuring two apps in .angular-cli.json

```
"apps": [
  {
    "name": "oneway",   <---- The name of the first app
     ...
    "main": "main-one-way-binding.ts",   <---- The bootstrap file of the first app
   ...
  },
  {
    "name": "twoway",   <---- The name of the second app
     ...
    "main": "main-two-way-binding.ts",   <---- The bootstrap file of the second app
   ...
  }
  ]
```

Because both apps are located in the same project, you need to run `npm install` once. In Angular 5 and earlier versions, you can bundle and run any of these app by

specifying the `--app` option with the `ng serve` or `ng build` command . The file main-one-way-binding.ts contains the code to bootstrap the app module from the directory named one-way, and the file main-two-way-binding.ts bootstraps the app module from the two-way directory of this project.

In Angular 5, if you want to build the bundles in memory and start the dev server with the app configured under the name `oneway`, the following command will do it:

```
ng serve --app oneway
```

NOTE If you use the Angular 6 version of the code samples, the option `--app` is not required: `ng serve oneway`.

If you also want Angular CLI to open the browser to http://localhost:4200, add the `-o` to the preceding command:

```
ng serve --app oneway -o
```

Open the bindings project in your IDE and run the `npm i` command in its Terminal window. After dependencies are installed, run the preceding command to see the one-way sample app in action. It'll render the page as shown in figure 2.3.

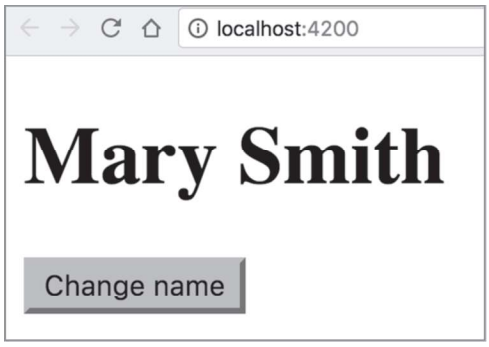


Figure 2.3 Running a one-way binding app

In the Angular 6 version of the code samples, the apps `oneway` and `twoway` are configured in the angular.json file in the `projects` section. The following command will run the `oneway` app, rendering the UI shown in figure 2.3.

```
ng serve oneway
```

The following listing shows the code of the `AppComponent` that rendered this page.

Listing 2.13 The `AppComponent` for the one-way binding sample

```
@Component({
  selector: 'app-root',
  template:`
  <h1>{{name}}</h1>
```

Initially uses a one-way property binding to render the value of the class variable name

```
    <button (click)="changeName()">
    Change name
  </button>
  `
})
export class AppComponent {
  name: string = "Mary Smith";

  changeName() {
  this.name = "Bill Smart";
  }
}
```

A button click updates the value of the variable name to Bill Smart using a one-way event binding to the method changeName().

As soon as the user clicks the button, the `changeName()` method modifies the value of `name`, one-way property binding kicks in, and the new value of the `name` variable will be shown on the page.

Now stop the dev server (Ctrl-C), and run the app configured under the name `twoway`:

```
ng serve --app twoway -o
```

The template of this page has the following HTML tags: `<input>`, `<button>`, and `<p>`. Enter `26 Broadway` in the input field, and you'll see a page like the one shown in figure 2.4.

Figure 2.4 Running a two-way binding sample app

The value of the text inside the `<p>` tag changes as soon as the value in the input field changes. If you click the button, the value of the input field and the paragraph will change to `"The shipping address is 123 Main Street"`. In this app, you use two-way binding. The code for the app component is shown in the following listing.

Listing 2.14 A two-way binding sample

```
@Component({
  selector: 'app-root',
  template: `
  <input type='text'
   placeholder="Enter shipping address"
   [(ngModel)]="shippingAddress">

  <button (click)="shippingAddress='123 Main Street'">
  Set Default Address
```

Using ngModel to denote the two-way binding

```
  </button>                                          <-- Updating the value of shippingAddress on click
   <p>The shipping address is {{shippingAddress}}</p>
  `
})
export class AppComponent {
  shippingAddress: string;
}
```

You bind the value of the input field to the `shippingAddress` variable by using the Angular `ngModel` directive:

```
[(ngModel)]="shippingAddress"
```

Remember, square brackets represent property binding, and parentheses represent event binding. To denote two-way binding, surround a template element's `ngModel` with both square brackets and parentheses. In the preceding code, you instruct Angular to update the `shippingAddress` variable as soon as the value in the input field changes, and update the value of the input field as soon when the value of `shippingAddress` changes. This is what the two-way binding means.

When you were typing `26 Broadway`, the value of the `shippingAddress` variable was changing as well. The click on the button would programmatically change the address to 123 Main Street, and this value is propagated back to the input field.

While reviewing the code of this app located in the two-way directory, note that the app module imports `FormsModule`, required because you use the `ngModel` directive, which is part of the Forms API, covered in chapter 7.

2.7 Hands-on: Getting started with ngAuction

From here on, most chapters will end with a hands-on section containing code review or instructions for developing a certain aspect of the ngAuction app, where people can see a list of featured products, view details for a specific product, perform a product search, and monitor bidding by other users. We'll gradually add code to this application so you can see how different Angular parts fit together. The source code that comes with this book includes the completed version of such hands-on sections in the ngAuction folders of the respective chapters, but we encourage you to try these exercises on your own (source code can be found at https://github.com/Farata/angulartypescript and www.manning.com/books/angular-development-with-typescript-second-edition).

This hands-on exercise contains instructions for developing an initial version of the sample auction introduced in chapter 1. You'll start by generating a new Angular CLI project, and then you'll create the home page, split it into Angular components, and create a service to fetch products.

> **NOTE** The completed version of this exercise is located in the directory chapter2/ngAuction. To run this version of ngAuction, switch to the ngAuction directory, run `npm install`, and start the application by running the `ng serve` command. We assume you have Angular CLI and the npm package manager installed on your computer.

2.7.1 The initial project setup for ngAuction

Let's start the development of ngAuction from scratch. Each section in the hands-on exercise will contain a set of instructions for you to follow so you can develop ngAuction on your own.

Generate a new directory and, using Angular CLI, generate a new project—ngAuction—by running the following command:

```
ng new ngAuction --prefix nga --routing
```

- `ng new`—Generates a new project.
- `--prefix nga`—The generated .angular-cli.json file will have the `prefix` property value `nga` (for ngAuction). The generated app component will have the `nga-root` selector, and all other components that we'll be generating for ngAuction will also have selectors with the prefix `nga-`.
- `--routing`—You'll add navigation to ngAuction in chapter 3. The `--routing` option will generate a boilerplate module for routing support.

Open the newly generated ngAuction directory in your IDE, go to the integrated Terminal view, and run the following command there:

```
ng serve -o
```

This command will build the bundles, start the dev server, and open the browser, which will render the page as in the Hello CLI app shown in chapter 1 in figure 1.5.

In chapters 2, 3, and 5 you'll use the Bootstrap 4 framework (see http://getbootstrap.com) for styling and implementing the responsive web design in ngAuction. The term *responsive web design* (RWD) means that the view layouts can adapt to the screen size of the user's device. Starting in chapter 7, you'll redesign the UI of ngAuction using Angular Material components (see https://material.angular.io) and remove the Bootstrap framework from this project.

Because the Bootstrap library has jQuery and Popper.js as peer dependencies, you need to run the command in the following listing to install them in the ngAuction project.

Listing 2.15 Installing Bootstrap, jQuery, and Popper.js

```
npm i bootstrap jquery popper.js --save-prod
```

TIP If you use npm older than 5.0, use the `--save` option instead of `--save-prod`. In npm 5, there are shortcuts: `-P` for `--save-prod` (default) and `-D` for `--save-dev`.

When you have to use global styles or scripts from an external JavaScript library, you can add them to the .angular-cli.json config file or, starting in Angular 6, to angular.json. In your case, the Bootstrap getting-started guide (see https://getbootstrap.com/docs/4.1/getting-started) instructs you to add bootstrap.min.css to the index .html

of the app. But because you're use Angular CLI, you'll add it to the `styles` section in .angular-cli.json, so it looks like this:

```
"styles": [
  "styles.css",
  "../node_modules/bootstrap/dist/css/bootstrap.min.css"
]
```

The Bootstrap documentation also instructs you to add the jQuery js/bootstrap.min.js file, and you'll add it to the `scripts` section in .angular-cli.json as follows:

```
"scripts": [
  "..node_modules/jquery/dist/jquery.min.js",
  "..node_modules/bootstrap/dist/js/bootstrap.min.js"
]
```

TIP When you're running the `ng serve` or `ng build` commands, the preceding scripts will be placed in the scripts.bundle.js file.

2.7.2 Generating components for ngAuction

Your ngAuction app will consist of several Angular components. In the last section, you generated the project with the root app component. Now, you'll generate more components using the command `ng generate component` (or `ng g c`). Run the commands in the following listing in your Terminal window to generate the components shown in figure 1.10 in chapter 1.

Listing 2.16 Generating components for ngAuction

```
ng g c home
ng g c carousel
ng g c footer
ng g c navbar
ng g c product-item
ng g c product-detail
ng g c search
ng g c stars
```

Each of the components in listing 2.12 will be generated in a separate folder. Open app.module.ts, and you'll see that Angular CLI has also added the import statements and declared all of the these components there.

Now you'll generate the product service that will provide the data for ngAuction in the next chapter. Run the following command to generate the product service:

```
ng g s shared/product
```

Add `ProductService` to the `providers` property of `@NgModule()`:

```
@NgModule({
  ...
  providers: [ProductService],
```

```
  ...
})
export class AppModule { }
```

> **TIP** While writing or generating code, you may see that some of your code fragments are marked with red squiggly lines. Hover over these lines to get more information. It may be not a TypeScript compiler error but a TSLint complaint about your coding style. Run the command `ng lint --fix`, and these styling errors could be automatically fixed.

2.7.3 The application component

The application component is the root component of ngAuction and will host all other components. Your app.component.html will include the following elements: navbar at the top, search on the left, the router outlet on the right, and the footer at the bottom. In chapter 3, you'll use the tag `<router-outlet>` to render either the `HomeComponent` or `ProductDetailComponent`, but in the initial version of ngAuction, you'll be rendering only `HomeComponent` there. Replace the content of app.component.html with the following listing.

Listing 2.17 The `AppComponent` template

```
<nga-navbar></nga-navbar>

<div class="container">
  <div class="row">
    <div class="col-md-3">
      <nga-search></nga-search>
    </div>

    <div class="col-md-9">
      <router-outlet></router-outlet>
    </div>
  </div>
</div>

<nga-footer></nga-footer>
```

The navbar component goes at the top.

Three columns of the Bootstrap's flex grid are given to the search component.

Rendering the search component

Nine columns are given to the router outlet area.

Router outlet area

The footer component is rendered at the bottom.

You may see some unfamiliar CSS classes in HTML elements—they all come from the Bootstrap framework. For example, the styles `col-md-3` and `col-md-9` come from the Bootstrap flexible grid layout system, in which the width of the viewport is divided into 12 invisible columns. You can read about the Bootstrap grid system at https://getbootstrap.com/docs/4.0/layout/grid.

Out of the box, the Bootstrap grid supports five tiers for different widths of user devices: `xs`, `sm`, `md`, `lg`, and `xl`. For example, `md` stands for medium devices (992 px (pixels) or more), `lg` is 1200 px or more, and so on. In your app component, you want to allocate three columns in medium or larger viewports to the search component (`<nga-search>`) and nine columns to the `<router-outlet>`.

Since you didn't specify how many columns to allocate to `<nga-search>` if the device is smaller than `md`, the browser will give the entire viewport's width to your search component, and `<router-outlet>` will be rendered under the search. The UI elements of your app component will be laid out differently, depending on the width of the user's screen.

Start the app with `ng serve -o`, and the browser will display a page that looks similar to the page shown in figure 2.5.

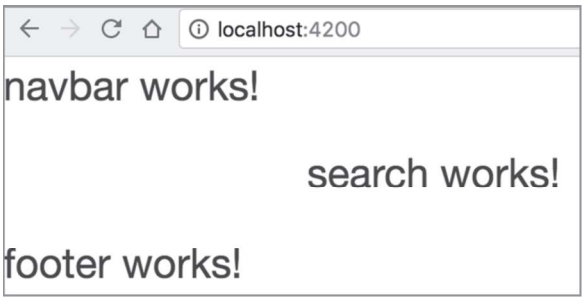


Figure 2.5 Running the first version of ngAuction

This doesn't look like a landing page of an online auction app yet, but at least the app didn't break after you added the Bootstrap framework and generated components and a service. Keep this app running and continue adding code according to the upcoming instructions, and you'll see how it gradually turns into a more usable web page.

2.7.4 The navbar component

A typical navigation bar stays at the top of the page, providing the app menu. The Bootstrap framework offers multiple styles for the navigation bar component described at https://getbootstrap.com/docs/4.0/components/navbar. Replace the contents of navbar.component.html with the following listing.

Listing 2.18 The navbar component template

Bootstrap's nav component and CSS selectors

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
   <a class="navbar-brand" [routerLink]="['/']">ngAuction</a>
   <button class="navbar-toggler" type="button"
    data-toggle="collapse"
    data-target="#navbarSupportedContent"
    aria-controls="navbarSupportedContent"
    aria-expanded="false" aria-label="Toggle navigation>
    <span class="navbar-toggler-icon"></span>
  </button>

  <div class="collapse navbar-collapse"
    id="navbarSupportedContent">
```

Bootstrap's navbar-brand denotes your auction brand and routes to the default page on click.

Bootstrap's collapsible is rendered as three horizontal bars on small screens.

```
     <ul class="navbar-nav mr-auto">
    <li class="nav-item active">
      <a class="nav-link" href="#">Home <span class="sr-
     only">(current)</span></a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="#">About</a>
    </li>
    <li class="nav-item dropdown">
       <a class="nav-link dropdown-toggle" href="#"
         id="navbarDropdown" role="button"
         data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
         Services
       </a>
    <div class="dropdown-menu" aria-labelledby="navbarDropdown">
      <a class="dropdown-item" href="#">Find products</a>
       <a class="dropdown-item" href="#">Place order</a>
    <div class="dropdown-divider"></div>
      <a class="dropdown-item" href="#">Pay</a>
    </div>
    </li>
    </ul>
  </div>
</nav>
```

Services drop-down menu

Drop-down menu items

As soon as you save this file, Angular CLI will automatically rebuild the bundles, and your page will look like figure 2.6.

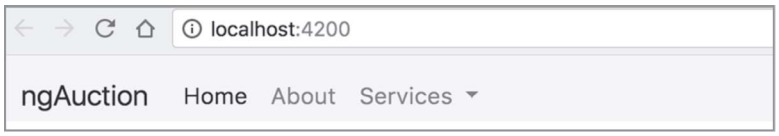


Figure 2.6 Rendering the navbar

What if you don't see the page shown in figure 2.6, and the browser just renders an empty page instead? Any time you see something you don't expect, open Chrome Dev Tools and see if there are any error messages in the Console tab.

Make the width of the browser window small, and the menu will collapse, rendering three horizontal bars on the right, as shown in figure 2.7.

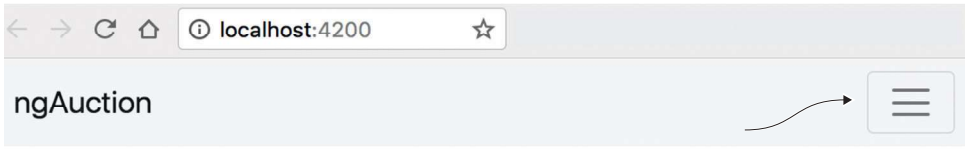


Figure 2.7 Collapsed navbar menu

2.7.5 The search component

Eventually, you'll implement the `SearchComponent` to perform the product search based on the product title, price, or category. But in the initial version of ngAuction, you just want to render the view of the `SearchComponent`. Replace the content of search.component.html to look like the following listing.

Listing 2.19 The search component template

```
<form name="searchForm">
  <div class="form-group">
     <label for="productTitle">Product title:</label>
    <input type="text" id="productTitle"
      placeholder="Title" class="form-control">
   </div>
  <div class="form-group">
    <label for="productPrice">Product price:</label>
    <input id="productPrice"
     name="productPrice" type="number" min="0"
     placeholder="Price" class="form-control">
   </div>
  <div class="form-group">
    <label for="productCategory">Product category:</label>
    <select id="productCategory" class="form-control"></select>
   </div>
  <div class="form-group">
    <button type="submit"
    class="btn btn-primary btn-block">Search</button>
   </div>
</form>
```

All values in class selectors come from the Bootstrap framework.

The rendered app is shown in figure 2.8.

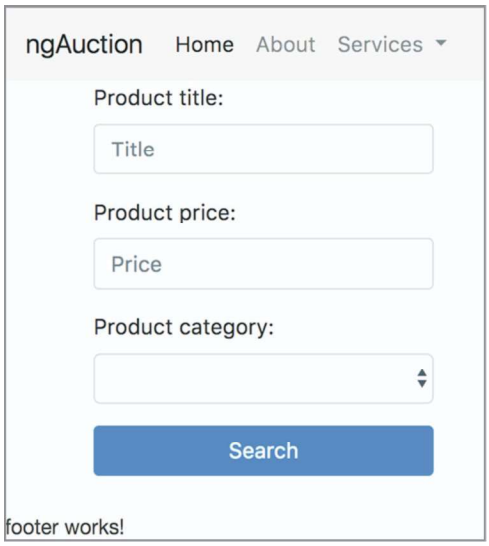


Figure 2.8 Rendering the search component

Now, let's take care of the footer component.

2.7.6 The footer component

Your footer will just display the copyright message. The following listing modifies the content of footer.component.html.

Listing 2.20 The footer component template

```
<div class="container">
  <hr>
  <footer>
   <div class="row">
     <div class="col-lg-12">
       <p>Copyright &copy; ngAuction 2018</p>
    </div>
   </div>
  </footer>
</div>
```

All values in class selectors come from the Bootstrap framework.

Your ngAuction app renders the home page as shown in figure 2.9.

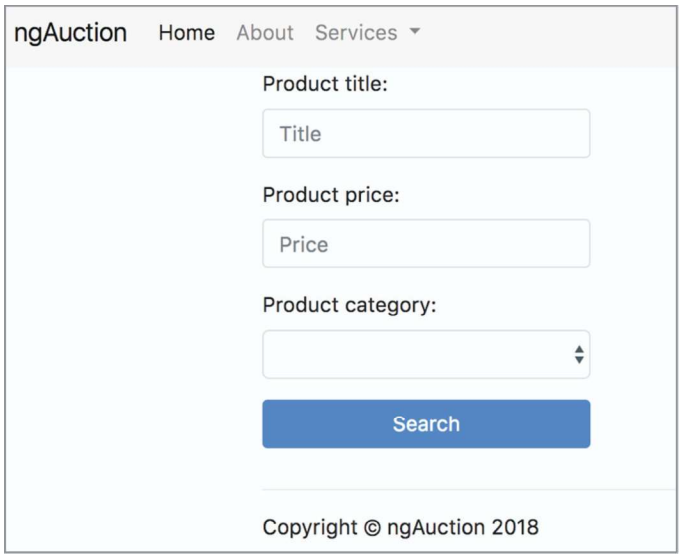


Figure 2.9 Rendering the footer

2.7.7 The carousel component

At the top of the ngAuction landing page, you want to implement a slide show of the featured products. For this, you'll use the carousel component that comes with Bootstrap (see https://getbootstrap.com/docs/4.0/components/carousel). To rotate

slides manually, the carousel component includes previous/next controls (little arrows on the sides) and indicators of the current slide (dashes at the bottom).

For simplicity, you'll use gray rectangles instead of actual images. The handy https://placeholder.com service returns gray rectangles of a specified size, and in the carousel, you'll use three 800 x 300 px gray rectangles.

Modify the code of carousel.component.html to look like the following listing.

Listing 2.21 The carousel component template

```
<div id="myCarousel" class="carousel slide" data-ride="carousel">
  <ol class="carousel-indicators">
    <li data-target="#myCarousel" data-slide-to="0" class="active"></li>
    <li data-target="#myCarousel" data-slide-to="1"></li>
    <li data-target="#myCarousel" data-slide-to="2"></li>
  </ol>
  <div class="carousel-inner">
    <div class="carousel-item active">
      <img class="d-block w-100" src="http://placehold.it/800x300">
    </div>
  <div class="carousel-item">
    <img class="d-block w-100" src="http://placehold.it/800x300">
  </div>
  <div class="carousel-item">
    <img class="d-block w-100" src="http://placehold.it/800x300">
  </div>
  </div>
  <a class="carousel-control-prev" href="#myCarousel"
   role="button" data-slide="prev">
    <span class="carousel-control-prev-icon" aria-hidden="true"></span>
    <span class="sr-only">Previous</span>
  </a>
  <a class="carousel-control-next" href="#myCarousel"
   role="button" data-slide="next">
    <span class="carousel-control-next-icon" aria-hidden="true"></span>
    <span class="sr-only">Next</span>
  </a>
</div>
```

The first slide is an 800 x 300 px gray rectangle.

Second slide

Third slide

Clicking the left arrow renders the previous image.

Clicking the right arrow renders the next image.

Now, you need to add some styles to the carousel. Because it's a custom component, you'll add `display: block` to its CSS file. You also want to add some space at the bottom of the carousel so other components won't overlap. To apply these styles to the component itself and not to its internals, you'll use the pseudo class selector `:host` that represents the carousel, in this case. To ensure that the slide images take the entire width of the `<div>` that hosts the carousel, add the following listing's style to the carousel.component.css file.

Listing 2.22 carousel.component.css

```
:host {
   display: block;
   margin-bottom: 10px;
 }

img {
  width: 100%;
 }
```

Applies styles to the carousel component and not to its internals

Displays the component as a block element that takes the entire width

Images should take the entire width of the carousel.

Adds some space below the carousel

Overriding Bootstrap styles

Most of the Bootstrap framework styles are located in the node_modules/bootstrap/dist/css/bootstrap.css file. If you want to override some of the default styles, see how Bootstrap defines them and decide what you want to change. Then, define the CSS style in your component that matches the selector of the Bootstrap file.

For example, the carousel indicators are rendered as dashes, and the Bootstrap CSS selector `.carousel-indicators li` looks like the following:

```
.carousel-indicators li {
  position: relative;
  -webkit-box-flex: 0;
  -ms-flex: 0 1 auto;
  flex: 0 1 auto;
  width: 30px;
  height: 3px;
  margin-right: 3px;
  margin-left: 3px;
  text-indent: -999px;
  background-color: rgba(255, 255, 255, 0.5);
}
```

If you want to change the indicators from dashes to circles, add the following style to the carousel.component.css:

```
.carousel-indicators li {
  width: 10px;
  height: 10px;
  border-radius: 100%;
}
```

Don't be surprised if, after adding code in the carousel component, the rendering of ngAuction hasn't changed and still looks like figure 2.9. That's because you haven't added the `<nga-carousel>` tag to any parent component yet. You'll add `<nga-carousel>` to the home component, which you'll create next.

2.7.8 The home component

The template of your app component includes the `<router-outlet>` area, which, on the `md`-size viewports, will be located to the right of `<nga-search>`. In chapter 3, you'll modify ngAuction to render either the home or the product-detail component in the `<router-outlet>`, but for now, you'll render the home component there. Your home component will host and render the carousel at the top and several products under it.

Modify the content of the generated home.component.html to look like the following listing.

Listing 2.23 The home component template

```
<div class="row carousel-holder">
  <div class="col-md-12">
  <nga-carousel></nga-carousel>
   </div>
</div>
<div class="row">
  We'll render several ProductItem components here
 </div>
```

The carousel goes at the top of the home component.

In chapter 3, you'll replace this text with product-item components.

The first `<div>` hosts the carousel, and the second one displays the text revealing your plans to render several product items there. Still, the UI of your running ngAuction hasn't changed, and you may have guessed that it's because you didn't include the `<nga-home>` tag in your app component. And you won't be doing that. You'll use Angular Router to render `HomeComponent` inside the `<router-outlet>` area.

Chapters 3 and 4 cover the router in detail—for now, you'll just make a small change in the generated app/app-routing.module.ts file, which includes the line in the following listing for route configuration.

Listing 2.24 Configuring routes

```
const routes: Routes = [];
```

Replace the code in the preceding listing with the code in the following one.

Listing 2.25 Mapping an empty path to the home component

```
const routes: Routes = [
  {
  path: '', component: HomeComponent
  }
];
```

This means that if the path after the base URL is empty (the URL of ngAuction has nothing after the port number), render the `HomeComponent`. You'll also need to add an import statement for `HomeComponent` in the app-routing.module.ts file.

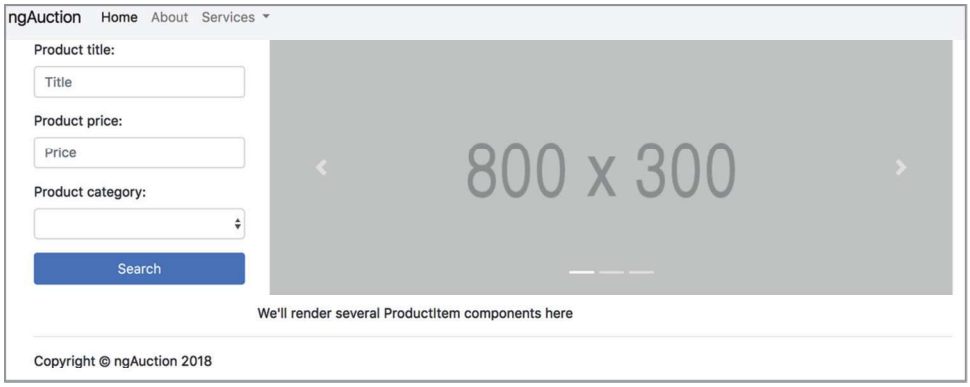


Figure 2.10 Rendering ngAuction with the home component

Now your ngAuction renders as shown in figure 2.10, and the carousel will run a slide show of gray rectangles.

The initial version of ngAuction is ready. You can start reducing the width of the browser window to see how this UI will be rendered on a smaller device. As soon as the window width becomes smaller than 992 pixels (the value of the Bootstrap `md` size), the browser will change the page layout, giving the entire window's width to the search component, and the home component will be rendered under the search. This is responsive web design in action.

Summary

- An Angular application is represented by a hierarchy of components that are packaged into modules.
- Each Angular component contains a template for UI rendering and a class implementing the component's functionality.
- Templates and styles can be either inlined or stored in separate files.
- Angular CLI is a useful tool even after the project has been generated.
- Data binding is a mechanism to keep the UI of the component and the values in the underlying class in sync.
- You can use third-party JavaScript libraries in Angular projects.

3 Router basics

This chapter covers

- Configuring parent and child routes
- Passing data while navigating from one route to another
- Configuring and using child routes

In a single-page application (SPA), the web page won't be reloaded, but its parts may change. You'll want to add navigation to this application so it'll change the content area of the page (known as the *router outlet*) based on the user's actions. The Angular router allows you to configure and implement such navigation without performing a full page reload.

In general, you can think of a *router* as an object responsible for the view state of the application. Every application has one router object, and you need to configure the routes of your app.

In this chapter, we'll discuss the major features of the Angular router, including configuring routes in parent and child components, passing data to routes, and adding router support to the HTML anchor elements.

The ngAuction app now has a home view; you'll add a second view so that if the user clicks the title of a product on the home page, the page's content will change to display the details of the selected product. At any given time, the user will see either the `HomeComponent` or the `ProductDetailComponent` in the `<router-outlet>` area.

3.1 Routing basics

You can think of an SPA as a collection of states, such as home, product detail, and shipping. Each state represents a different view of the same SPA.

Figure 3.1 shows the landing page of the ngAuction app, which has a navigation bar (a component) at the top, a search form (another component) on the left, and a footer (yet another component) at the bottom, and you want these components to remain visible all the time.

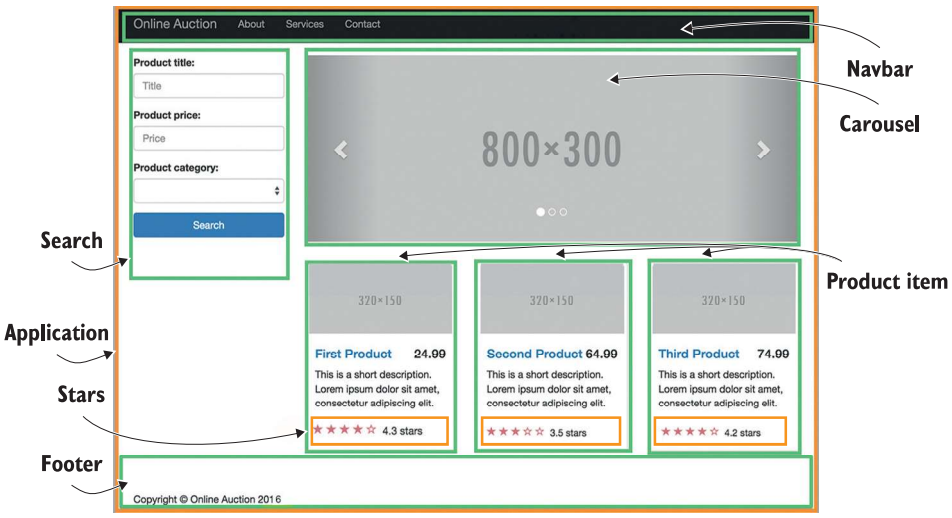


Figure 3.1 Components on the home page of ngAuction

Besides the parts that are always visible, there's a content area (see figure 3.2) that initially will display the `<nga-home>` component and its children but can show other views as well, based on the user's actions. To show other views, you'll need to configure the router so it can display different views in the outlet, replacing one view with another. You'll be assigning a component for each view that you want to display in this area. This content area is represented by the tag `<router-outlet>`.

TIP There can be more than one outlet on the page. We'll cover this in section 4.2 in chapter 4.

The router is responsible for managing client-side navigation, and later in this chapter we provide a high-level overview of the router. In the non-SPA world, site navigation is implemented by a series of requests to a server, which refreshes the entire page by sending the appropriate HTML documents to the browser. With SPAs, the code for rendering components is already on the client (except for the lazy-loading scenarios covered in section 4.3 of chapter 4), and you need to replace one view with another.

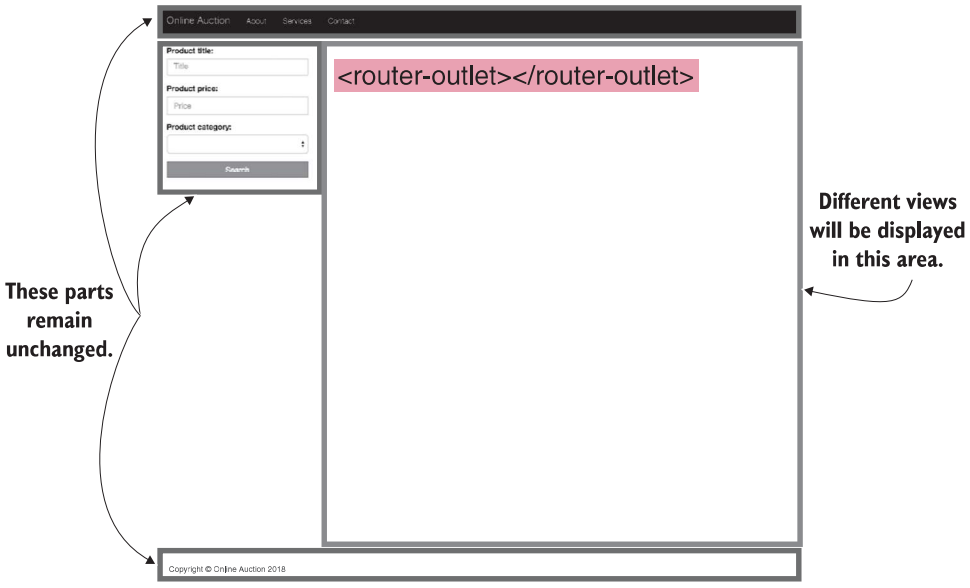


Figure 3.2 Allocating the area for changing views

As the user navigates the application, the app can still make requests to the server to retrieve or send data. Sometimes a view (the combination of the UI code and data) has everything it needs already downloaded to the browser. Other times a view will communicate with the server by issuing AJAX requests or via WebSockets. Each view will have a unique URL shown in the location bar of the browser. We'll discuss that next.

3.2 Location strategies

At any given time, the browser's location bar displays the URL of the current view. A URL can contain different parts, or segments. It starts with a protocol followed by a domain name, and it may include a port number. Parameters that need to be passed to the server may follow a question mark (this is true for HTTP `GET` requests), like this: http://mysite.com:8080/auction?someParam=1234.

In a non-SPA, changing any character in the preceding URL results in a new request to the server. In SPAs, you need the ability to modify the URL without forcing the browser to make a server-side request so the application can locate the proper view on the client. Angular offers two location strategies for implementing client-side navigation:

- `HashLocationStrategy`—A hash sign (#) is added to the URL, and the URL segment after the hash uniquely identifies the view to be used as a web page fragment. This strategy works with all browsers, including the old ones.
- `PathLocationStrategy`—This `History` API–based strategy works only in browsers that support HTML5. This is the default location strategy in Angular.

3.2.1 Hash-based navigation

A sample URL that uses hash-based navigation is shown in figure 3.3. Changing any character to the right of the hash sign doesn't cause a direct server-side request but navigates to the view represented by the path (with or without parameters) after the hash. The hash sign serves as a separator between the base URL and the client-side location of the required content.

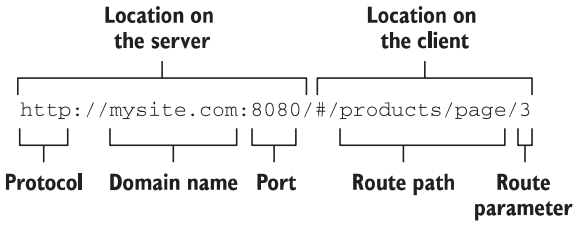


Figure 3.3 Dissecting the URL

Try to navigate an SPA like Gmail and watch the URL. For the Inbox, it looks like this: https://mail.google.com/mail/u/0/#inbox. Now go to the Sent folder, and the hash portion of the URL will change from *inbox* to *sent*. The client-side JavaScript code invokes the necessary functions to display the Sent view. But why does the Gmail app still show you the "Loading . . ." message when you switch to the Sent box? The JavaScript code of the Sent view can still make AJAX requests to the server to get the new data, but it doesn't have to load any additional code, markup, or CSS from the server.

To use hash-based navigation, `@NgModule()` has to include the `providers` value (we discuss providers in the section 5.2 in chapter 5), as shown in the following listing.

Listing 3.1 Using the hash location strategy

```
import {HashLocationStrategy, LocationStrategy} from "@angular/common";
...
@NgModule({
...
  providers:[{provide: LocationStrategy, useClass: HashLocationStrategy}] <-
 })
```

The provider is needed, so Angular injects the service supporting the hash location strategy.

3.2.2 History API-based navigation

The browser's `History` API allows you to move back and forth through the user's navigation history as well as programmatically manipulate the history stack (see "Manipulating the Browser History" in the Mozilla Developer Network, http://mng .bz/i64G). In particular, the `pushState()` method is used to attach a segment to the base URL as the user navigates your SPA.

Consider the following URL: http://mysite.com:8080/products/page/3 (note the absence of the hash sign). The URL segment *products/page/3* can be pushed

(attached) to the base URL programmatically without using the hash tag. If the user navigates from page 3 to 4, the application's code will push the URL segment *products/page/4*, saving the previously visited *products/page/3* in the browser history.

Angular spares you from invoking `pushState()` explicitly—you just need to configure the URL segments and map them to the corresponding components. With the `History` API–based location strategy, you need to tell Angular what to use as a base URL in your application so it can properly append the client-side URL segments. If you want to serve an Angular app on a non-root path, you have to do the following:

- Add the `<base>` tag to the header of index.html, such as `<base href="/mypath">`,or use the `--base-href` option while running `ng build`. Angular CLI–generated projects include `<base href="/">` in index.html.
- Assign a value for the `APP_BASE_HREF` constant in the root module and use it as the `providers` value. The following listing uses `/` as a base URL, but it can be any URL segment that denotes the end of the base URL.

Listing 3.2 Adding support to a `History`-based API

```
import { APP_BASE_HREF } from '@angular/common';
...
@NgModule({
...
  providers:[{provide: APP_BASE_HREF, useValue: '/mypath'}]
 })
class AppModule { }
```

The provider is needed so the router properly resolves URLs.

`APP_BASE_HREF` affects how the router resolves `routerLink` properties and the `router.navigate()` calls within the app, whereas the `<base href=". . .">` tag affects how the browser resolves URLs when loading static resources like `<link>`, `<script>`, and `<img>` tags.

3.3 The building blocks of client-side navigation

Let's get familiar with the main concepts of implementing client-side navigation using the Angular router. Routes are configured using the `RouterModule`. If your application needs routing, make sure your package.json file includes the dependency `@angular/router`. Angular includes many classes supporting navigation—for example, `Router`, `Route`, `Routes`, `ActivatedRoute`, and others. You configure routes in an array of objects of type `Route`, as in the following listing. Each of the elements in this array is an object of type `Route`.

Listing 3.3 A sample routes configuration

```
const routes: Routes = [
  {path: '',        component: HomeComponent},
    {path: 'product', component: ProductDetailComponent}
];
```

Renders the HomeComponent by default

If the URL contains the product fragment, renders ProductDetailComponent

Because route configuration is done on the module level, you need to let the app module know about the routes in the `@NgModule()` decorator. If you declare routes for the root module, use the `forRoot()` method , for example, as shown in the following listing.

Listing 3.4 Letting the root module know about the routes

```
import { BrowserModule } from '@angular/platform-browser';
import { RouterModule } from '@angular/router';
...
@NgModule({
  imports: [BrowserModule,
            RouterModule.forRoot(routes)],
    ...
})
```

Creates a router module and a service for the app root module

If you generated your app using the Angular CLI command `ng new` with the `--routing` option (as you did in the hands-on section in chapter 2), you'll get a separate file, app-routing.module.ts, where you can configure routes, as illustrated in the next listing.

Listing 3.5 A separate module with route support

```
const routes: Routes = [
   { path: '', component: HomeComponent},
  { path: 'product', component: ProductDetailComponent}
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
   exports: [RouterModule]
 })
export class AppRoutingModule {}
```

Configures routes

Creates a router module and a service for the app root module

Makes this module accessible from other modules

If you're configuring routes for a feature module (not for the root one), use the `forChild()` method, which also creates a router module but doesn't create the router service (`forRoot()` should have created the service by now), as you can see in the following listing.

Listing 3.6 Creating a router module for a feature module

```
@NgModule({
  imports: [CommonModule,
            RouterModule.forChild(routes)]
    ...
})
export class MyFeatureModule {}
```

Creates a router module but doesn't create the router service

Let's start with a simple app that illustrates routing. Say you want to create a root component that has two links, Home and Product Details, at the top of the page. The application should render either `HomeComponent` or `ProductDetailComponent`, depending on which link the user clicks. `HomeComponent` will render the text "Home Component,"

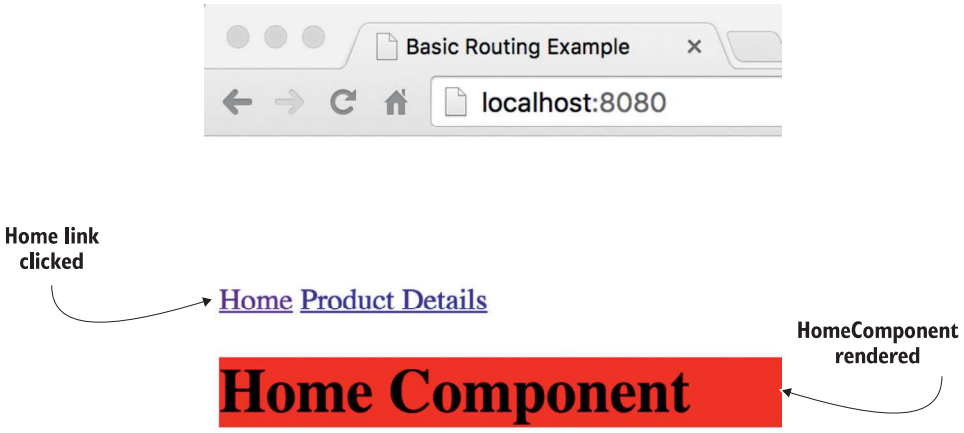


Figure 3.4 Home component rendered with red background

and `ProductDetailComponent` will render "Product Detail Component." Initially the web page should display `HomeComponent`, as shown in figure 3.4.

After the user clicks the Product Details link, the router should display the `ProductDetailComponent`, as shown in figure 3.5.

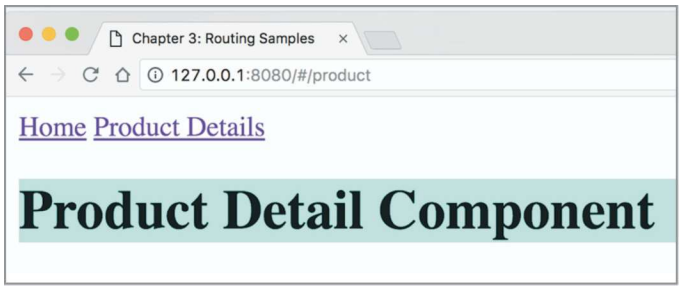


Figure 3.5 Product-detail component rendered with cyan background

You can see how the URLs for these routes look in figures 3.4 and 3.5. The main goal of this basic app is to become familiar with the router, so the components will be very simple, as in the following listing.

Listing 3.7 `HomeComponent`

```
@Component({
    selector: 'home',
    template: '<h1 class="home">Home Component</h1>',
    styles: ['.home {background: red}']})
 export class HomeComponent {}
```

Renders this component with red background

The code of the `ProductDetailComponent` looks similar, as you can see in the following listing, but instead of red it uses a cyan background.

Listing 3.8 `ProductDetailComponent`

```
@Component({
    selector: 'product',
    template: '<h1 class="product">Product Detail Component</h1>',
    styles: ['.product {background: cyan}']})
 export class ProductDetailComponent {}
```

Renders this component with cyan background

The `Routes` type is just a collection of the objects of the type defined in the `Route` interface, as shown in the next listing.

Listing 3.9 Angular's `Route` interface

```
export interface Route {
    path?: string;
    pathMatch?: string;
    matcher?: UrlMatcher;
    component?: Type<any>;
    redirectTo?: string;
    outlet?: string;
    canActivate?: any[];
    canActivateChild?: any[];
    canDeactivate?: any[];
    canLoad?: any[];
    data?: Data;
    resolve?: ResolveData;
    children?: Routes;
    loadChildren?: LoadChildren;
    runGuardsAndResolvers?: RunGuardsAndResolvers;
}
```

You can pass to the `forRoot()` or `forChild()` functions a config object that only has a couple of properties filled in. In the basic app, you use just two properties defined in the `Route` interface: `path` and `component`. We'll do it in a file called app.routing.ts, as in the following listing.

Listing 3.10 app.routing.ts

```
const routes: Routes = [
    {path: '',        component: HomeComponent},
     {path: 'product', component: ProductDetailComponent}
 ];

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

HomeComponent is mapped to a path containing an empty string, which implicitly makes it a default route.

Invokes forRoot() and exports the router configuration so it can be imported by the root module

If the URL has the product segment after the base URL, renders the ProductDetail-Component in the router outlet

The next step is to create a root component that will contain the links for navigating between the home and product-detail views. The following listing shows the root `AppComponent` located in the app.component.ts file.

Listing 3.11 app.component.ts

```
@Component({
    selector: 'app-root',
    template: `
      <a [routerLink]="['/']">Home</a>
       <a [routerLink]="['/product']">Product Details</a>
       <router-outlet></router-outlet>
     `
})
export class AppComponent {}
```

Creates a link that binds routerLink to the empty path

Creates a link that binds routerLink to the path /product

The <router-outlet> specifies the area on the page where the router will render the components (one at a time).

The square brackets around `routerLink` denote property binding, while the brackets on the right represent an array with one element (for example, `['/']`). The second anchor tag has the `routerLink` property bound to the component configured for the /product path. The matched components will be rendered in the area marked with `<router-outlet>`, which in this app is located below the anchor tags.

None of the components are aware of the router configuration, because it's the module's business, as shown in the following listing.

Listing 3.12 app.module.ts

```
...
import {routing} from './app.routing';

@NgModule({
    imports:      [ BrowserModule,
                    routing ],
     declarations: [ AppComponent,
                     HomeComponent,
                     ProductDetailComponent],
    providers:[{provide: LocationStrategy, useClass: HashLocationStrategy}],
     bootstrap:    [ AppComponent ]
})
class AppModule {}
```

Imports the routes configuration

Adds the routes configuration to @NgModule()

Lets the dependency injection mechanism know that you want HashLocationStrategy

The module's `providers` property is an array of registered providers (there's just one in this example) for dependency injection, which is covered in chapter 5. At this point, you just need to know that although the default location strategy is `PathLocationStrategy`, you want Angular to use the `HashLocationStrategy` class for routing (note the hash sign in the URL in figure 3.5).

In the project router-samples that comes with this chapter, we've configured multiple applications in the .angular-cli.json file. The app described in this section has the name *basic*, and you can run it by entering the following command in your Terminal window:

```
ng serve --app basic -o
```

NOTE In Angular 6, the .angular-cli.json file is renamed angular.json. Also, if you decide to run the Angular 6 version of this app (it comes with the book code samples), the `--app` option isn't needed: `ng serve basic -o`.

TIP Don't forget to run `npm install` in the project router-samples.

In the basic routing code sample, we arranged the navigation using `routerLink` in HTML anchor tags. But what if you need to arrange navigation programmatically without asking the user to click a link?

3.4 Navigating to routes with navigate()

Let's modify the basic code sample to navigate by using the `navigate()` method. You'll add a button that will also navigate to the `ProductDetailComponent`, but this time no HTML anchors will be used.

The following listing reuses the router configuration from the previous section but invokes the `navigate()` method on the `Router` instance that will be injected into the `AppComponent` via its constructor.

Listing 3.13 Using `navigate()`

```
@Component({
  selector: 'app',
  template: `
    <a [routerLink]="['/']">Home</a>
    <a [routerLink]="['/product']">Product Details</a>
    <button (click)="navigateToProductDetail()>Product Details
     </button>
    <router-outlet></router-outlet>
  `
})
class AppComponent {

    constructor(private _router: Router){}

    navigateToProductDetail(){
        this._router.navigate(["/product"]);
     }
}
```

Clicking this button invokes the navigateToProductDetail() method.

Angular will inject the instance of Router into the router variable.

Navigates to the configured product route programmatically

In listing 3.13, the user needs to click a button to go to the product route. But the navigation could be implemented without requiring user actions—just invoke the `navigate()` method from your application code when necessary. For example, you can force the app to navigate to the login route if the user isn't logged in.

By default the address bar of the browser changes as the user navigates with the router. If you don't want to show the URL of the current route, use the `skipLocationChange` directive :

```
<a [routerLink]="['/product']" skipLocationChange>Product Details</a>
```

In this case, the URL remains http:// localhost:4200/#/ even when the user navigates to the `product` route. To achieve the same effect with programmatic navigation, use the following syntax:

```
this._router.navigate(["/product"], {skipLocationChange: true});
```

Handling 404 errors

If the user enters a nonexistent URL in your application, the router won't be able to find a matching route and will print an error message on the browser console, leaving the user to wonder why no navigation is happening. Consider creating an application component that will be displayed whenever the application can't find the matching component.

For example, you could create a component named _404Component and configure it with the wildcard path `**`:

```
[
  {path: '',         component: HomeComponent},
  {path: 'product', component: ProductDetailComponent},
  {path: '**', component: _404Component}
])
```

The wildcard route configuration has to be the last element in the array of routes. The router always treats the wildcard route as a match, so any routes listed after the wildcard route won't be considered.

3.5 Passing data to routes

The basic routing application showed how you can display different components in the router outlet area, but you often need to also pass some data to the component. For example, if the app component shows a list of products and you want to navigate to the product-detail route, you need to pass the product ID to the component that represents the destination route. In this case, you need to add a parameter to the `path` property in the route configuration. In the following listing, you change the configuration of the `product` route to indicate that the `ProductDetailComponent` has to be rendered when the URL segment includes the value after `'product'` (the colon denotes the variable part of the path - `:id`).

Listing 3.14 `Routes` configuration

```
const routes: Routes = [
    {path: '',        component: HomeComponent},
    {path: 'product/:id', component: ProductDetailComponent}
 ];
```

If the URL contains the fragment product followed by a value, renders ProductDetailComponent and passes the value to it

Accordingly, your app component needs to include the value of the product ID in the `routerLink`to ensure that the value of the product ID will be passed to the `ProductDetailComponent` if the user chooses to go this route. The new version of the app may look like the following listing.

Listing 3.15 app.component.ts

```
@Component({
    selector: 'app-root',
    template: `
        <a [routerLink]="['/']">Home</a>
        <a [routerLink]="['/product', productId]">Product Detail</a>
         <router-outlet></router-outlet>
    `
})
class AppComponent {
  productId = 1234;
 }
```

If the user clicks this link, navigates to the product route, passing the value of productId

Sets the value of the productId property

The second `routerLink` is bound to the two-element array providing the static part of the path /product and the value `/:id` that represents the product ID. The elements of the array build up the path specified in the routes configuration given to the `RouterModule.forRoot()` method. For the product-detail route, Angular will construct the URL segment /product/1234.

To see this app in action, run the following command in your Terminal window:

```
ng serve --app params -o
```

3.5.1 Extracting parameters from ActivatedRoute

If a parent component can pass a parameter to the route, the component that represents the destination route should be able to receive it. Instruct Angular to inject the instance of `ActivatedRoute` to the constructor of the component that represents the destination route. The instance of the `ActivatedRoute` will include the passed parameters, as well as the route's URL segment and other properties. The new version of the component, which renders product detail and is capable of receiving parameters, will be called `ProductDetailComponent`, which will get an object of type `ActivatedRoute` injected into it, as shown in the following listing.

Listing 3.16 `ProductDetailComponentParam`

```
@Component({
  selector: 'product',
  template: `<h1 class="product">Product Detail for Product: {{productID}}</h1>`,
   styles: ['.product {background: cyan}']
})
export class ProductDetailComponent {
  productID: string;

  constructor(route: ActivatedRoute) {
     this.productID = route.snapshot.paramMap.get('id');
   }
}
```

Displays the received product ID

The ActivatedRoute object is injected into the constructor of this component.

Gets the value of the parameter named id and assigns it to the productID class variable, which is used in the template via binding

Figure 3.6 shows how the product-detail view will be rendered in the browser. Note the URL: the router replaced the product/:id path with /product/1234.

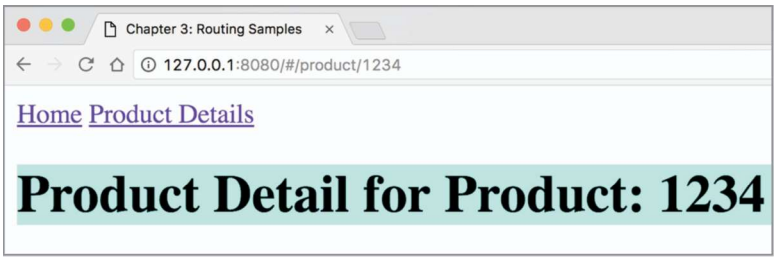


Figure 3.6 The product-detail route received the product ID `1234`.

Passing changing parameters to the route

In this app, you use the property `snapshot` of type `ActivatedRouteSnapshot` to retrieve the parameter's value. The `snapshot` means "one-time deal," and this property is used in scenarios when the parameters passed to the route never change. In your app, it works because the product ID in the parent route never changes and is always `1234`. But if you try to change the URL shown in figure 3.6 manually (for example, make it /product/12345), the `ProductDetailComponent` won't reflect the change in the parameter.

There are scenarios when the parameter value keeps changing after navigating to a route. For example, the `AppComponent` renders a list of products, and the user can select different products. Both `AppComponent` and `ProductDetailComponent` are rendered in the same window. In this case, instead of using the `snapshot` property, you need to subscribe to the `ActivatedRoute.paramMap` property, which will emit a new value each time the user clicks on a different product, for example:

(continued)

```
route.paramMap.subscribe(
  params => this.productID = params.get('id')
);
```

You'll see this example in chapter 6 in section 6.6.

Let's review the steps that Angular performed under the hood to render the main page of the application:

1. Check the content of each `routerLink`.
2. Concatenate the values specified in the array. If an array item is an expression, evaluate this expression (like `productId`). Finally, append the value of `APP_BASE_HREF` in the beginning of the resulting string.
3. The `RouterLink` directive adds the `href` attribute if this directive is attached to an `<a>` element; otherwise, it just listens to the `click` events.

Figure 3.7 shows a snapshot of the home page of the application with the Chrome Developer Tools panel open. Because the `path` property of the configured home route had an empty string, Angular didn't add anything to the base URL of the page. But the anchor under the Product Details link has already been converted into a regular HTML tag. When the user clicks the Product Details link, the router will attach a hash sign and add /product/1234 to the base URL so that the absolute URL of the product-detail view will become http:// localhost:4200/#/product/1234.

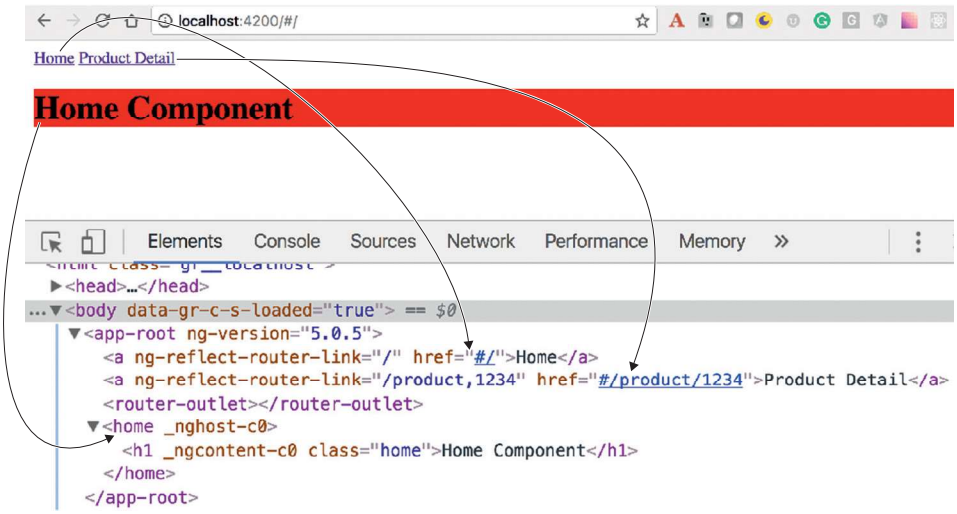


Figure 3.7 Angular-compiled code in the browser

NOTE In this section, you learned how to pass dynamic data to routes—the data that may change during the runtime. Sometimes, you need to pass static data to routes (data that doesn't change). You can pass any arbitrary data to routes by using the `data` property in the routes configuration. You'll see such an example in section 4.3.1 in chapter 4.

Sometimes, you need to pass to a route optional query parameters that are not part of the route configuration. Let's take a look at how to pass query parameters.

3.5.2 Passing query parameters to a route

You can use query parameters (the URL segment after the question mark), as in the following URL: http:// localhost:4200/products?category=sports. Query parameters aren't scoped to a particular route, and if you want to pass them while navigating with the `routerLink`, you can do as follows:

```
<a [routerLink]="['/products']" [queryParams]="{category:'sports'}">
      Sports products </a>
```

Because query parameters aren't scoped to a particular route and can be accessed by any active route, the route configuration doesn't need to include them:

```
{path: 'products', component: ProductDetailComponent}
```

To pass query parameters using programmatic navigation, you need to have access to the `Router` object. The code could look like the following listing.

Listing 3.17 Injecting and accessing the `Router` object

```
constructor (private router: Router) {}  <-- Injects the Router object

showSportingProducts() {
   this.router.navigate(['/products'],          <-- Invokes navigate() on the Router object
                        {queryParams: {category: 'sports'}});
}
```

In this example, you pass an object with one query parameter; but you can specify multiple parameters as well.

To receive query parameters in the destination component, you'll use the `ActivatedRoute` object again.

Listing 3.18 Receiving query parameters

```
@Component({
  selector: 'product',
  template: `<h1 class="product">Showing products in {{productCategory}}</h1>
     `,
  styles: ['.product {background: cyan}']
})
export class ProductDetailComponent {
  productCategory: string;
```

```
  constructor(route: ActivatedRoute) {
     this.productCategory = route.snapshot.queryParamMap.get('category');
   }
}
```

Extracts the query param named category

To see this code sample in action, run the following command:

```
ng serve --app queryparams -o
```

3.6 Child routes

An Angular application is a tree of components that have parent-child relations. A child component can have its own routes, but all routes are configured outside of any component. Imagine that you want to enable `ProductDetailComponent` (the child of the `AppComponent`) to show either the product description or the seller's info. Moreover, there could be more than one seller of the same product, so you'll need to pass the seller ID to show the details of the seller. The following listing configures routes for the child, `ProductDetailComponent`, by using the `children` property of the `Route`.

Listing 3.19 Configuring child routes

```
const routes: Routes = [
  {path: '',            component: HomeComponent},
  {path: 'product/:id', component: ProductDetailComponent,
    children: [
      {path: '',           component: ProductDescriptionComponent},
      {path: 'seller/:id', component: SellerInfoComponent}
    ]}
];
```

This property configures routes for ProductDetailComponent.

ProductDescriptionComponent there by default.

From ProductDetailComponent, the user can navigate to the SellerInfoComponent.

Here, the `children` property is a part of the configuration of the route with the path product/:id. You pass the product ID while navigating to the `product` route, and then, if the user decides to navigate to the `seller`, you pass the seller ID to the `SellerInfoComponent`.

Figure 3.8 shows how the application will look once the user clicks the Product Details link on the root component, which renders `ProductDetailComponent` (the child), showing `ProductDescriptionComponent` by default, because the latter component was configured for the empty `path` property.

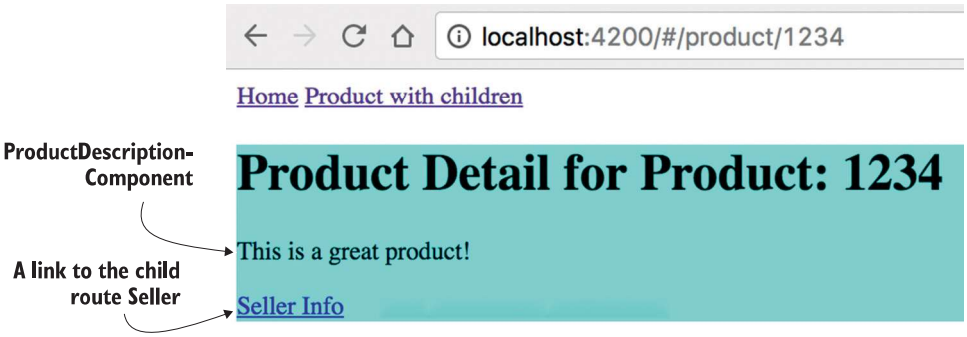


Figure 3.8 The product description route

Figure 3.9 shows the application after the user clicks the Product Details link and then clicks Seller Info.

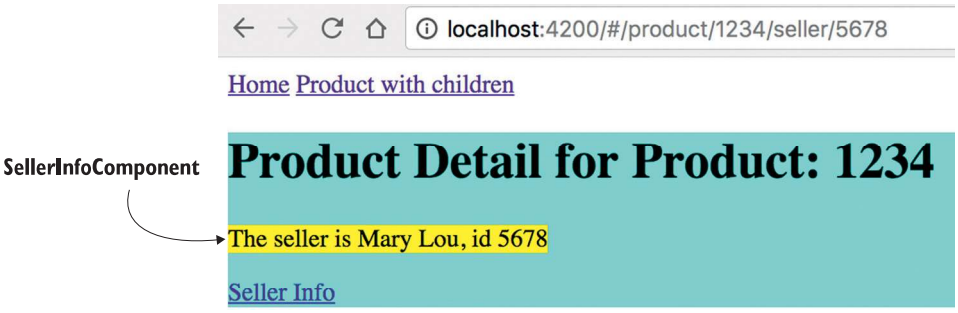


Figure 3.9 The child route renders `SellerInfo`

NOTE If you're reading the electronic version of this book, you'll see that the seller's info is shown on a yellow background. We did this intentionally to discuss the styling of components a bit later in this chapter.

To implement the views shown in figures 3.8 and 3.9, you'll modify `ProductDetailComponent` so it also has two children, `SellerInfoComponent` and `ProductDescriptionComponent`, and its own `<router-outlet>`. Figure 3.10 shows the hierarchy of components that you're going to implement.

The following three listings show the code of the `ProductDetailComponent`, `ProductDescriptionComponent`, and `SellerInfoComponent`. The new version of `ProductDetailComponent` has its own outlet, where it can display either `ProductDescriptionComponent` (the default) or `SellerInfoComponent`.

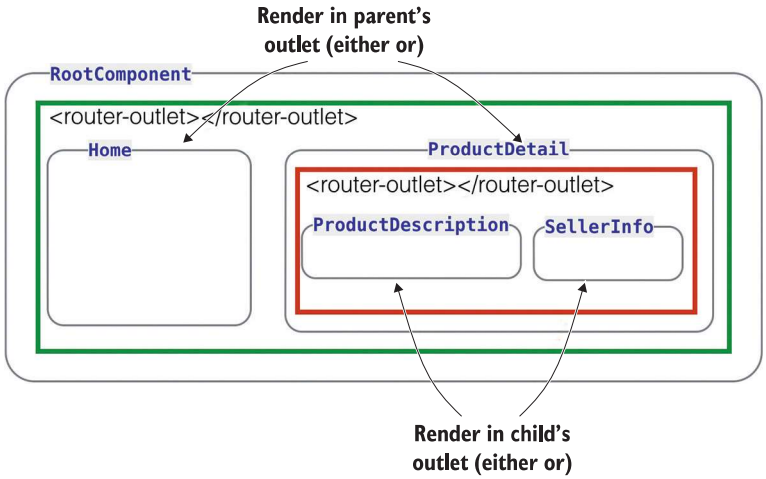


Figure 3.10 The routes hierarchy

Listing 3.20 product.detail.component.ts

```
@Component({
    selector: 'product',
    styles: ['.product {background: cyan}'],
    template: `
      <div class="product">
        <h1>Product Detail for Product: {{productId}}</h1>
        <router-outlet></router-outlet>
         <p><a [routerLink]="['./seller', sellerId]">Seller Info</a></p>
       </div>
    `
})
export class ProductDetailComponent {
  productId: string;
  sellerId = 5678;

  constructor(route: ActivatedRoute) {
    this.productId = route.snapshot.paramMap.get('id');
  }
}
```

ProductDetailComponent allocates its own router-outlet area for rendering its child components one at a time.

When the user clicks this link, Angular adds the /seller/5678 segment to the existing URL and renders SellerInfoComponent.

When the user clicks the Product With Children link, and it has children, the product/1234 segment is added to the URL. The router finds a match to this path in the configuration object and renders the `ProductDetailComponent` in the outlet.

The user navigates to the `ProductDetailComponent`, which by default renders the `ProductDescriptionComponent` as per route configuration. Then, the user clicks the

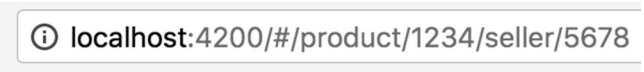


Figure 3.11 The URL of the seller component

Seller Info link, and the URL will include the product/1234/seller/5678 segment after the hash sign, as shown in figure 3.11.

The router will find a match in the configuration object and will render `SellerInfoComponent` in the child's `<router-outlet>`. The code of the `ProductDescriptionComponent` is trivial, as you can see in the following listing.

Listing 3.21 product.description.component.ts

```
@Component({
    selector: 'product-description',
    template: '<p>This is a great product!</p>'
})
export class ProductDescriptionComponent {}
```

Because `SellerInfoComponent` expects to receive the seller ID, its constructor needs an argument of type `ActivatedRoute` to get the seller ID, as the following listing shows, and as you did in `ProductDetailComponent`.

Listing 3.22 seller.info.component.ts

```
@Component({
    selector: 'seller',
    template: 'The seller is Mary Lou, id {{sellerID}} ',
    styles: [':host {background: yellow}']
 })
export class SellerInfoComponent {
   sellerID: string;

   constructor(route: ActivatedRoute){
      this.sellerID = route.snapshot.paramMap.get('id');
    }
}
```

A pseudo class :host is used to display the content of this component on a yellow background.

Injects the ActivatedRoute object

Gets the value of the passed id and assigns it to sellerID for rendering

The `:host` pseudo class selector can be used with elements that are created using Shadow DOM (discussed in section 8.5.1 in chapter 8), which provides better encapsulation for components. Although not all web browsers support Shadow DOM yet, Angular emulates Shadow DOM by default. Here, you use `:host` to apply the yellow background color to `SellerInfoComponent`. In the emulated mode, the `:host` selector is transformed into a randomly generated, attribute-based selector, like this:

```
[ng-host-f23ed] {
  background: yellow;
}
```

The attribute (here, `ng-host-f23ed`) is attached to the element that represents the component. Shadow DOM styles of the components aren't merged with the styles of the global DOM, and the IDs of the components' HTML tags won't overlap with the IDs of the DOM.

To run this code sample, enter the following command in the Terminal window of the router-samples project:

```
ng serve --app child -o
```

Deep linking

Deep linking is the ability to create a link to specific content inside a web page rather than to the entire page. In the basic routing applications, you've seen examples of deep linking:

- The URL http://localhost:4200/#/product/1234 links not just to the product-detail page but to a specific view representing the product with an ID of `1234`.
- The URL http://localhost:4200/#/product/1234/seller/5678 links even deeper. It shows the information about the seller with an ID of `5678` that sells the product whose ID is `1234`.

You can easily see deep linking in action by copying the link http://localhost:4200/#/product/1234/seller/5678 from the application running in Chrome and pasting it into Firefox or Safari. There is a caveat, though. With `PathLocationStrategy`, when you enter the direct URL of the route in the browser's address bar, it still makes a request to the server, which won't find the resource (and rightly so) named as your route. This will cause a 404 error. Configure your web server to do a redirect to index.html of your app in cases when a requested resource isn't found. This will put your Angular app back in control, and the route will be properly resolved. The Angular CLI development server is already configured for redirects.

Router events

As the user navigates your app, Angular dispatches events, such as `NavigationStart`, `NavigationEnd`, and so on. There are about a dozen router events, and you can intercept any of them if need be. In chapter 6, section 6.6, you'll see an example of using router events to decide when to show and hide the progress bar if the navigation is slow. For debugging purposes, you can log router events in the browser's console by using the `enableTracing` option (it works only in the root module):

```
RouterModule.forRoot(
  routes,
  {enableTracing: true}
)
```

Now that you've learned router basic features, let's see how can you apply them in your ngAuction application.

3.7 Hands-on: Adding navigation to the online auction

NOTE Source code for this chapter can be found at https://github.com/Farata/angulartypescript and www.manning.com/books/angular-development-with-typescript-second-edition.

This hands-on exercise starts where we left off in chapter 2. So far, you've partially implemented the landing page of ngAuction; your goal is to render several product items under the carousel component so the landing page looks as shown in figure 3.12.

The data for this view will be provided by `ProductService`. This hands-on exercise contains instructions for injecting the `ProductService` into the `HomeComponent`. You'll also implement the navigation so that when the user clicks the product title, the `Router` will render the `ProductDetail` component in the `<router-outlet>` area.

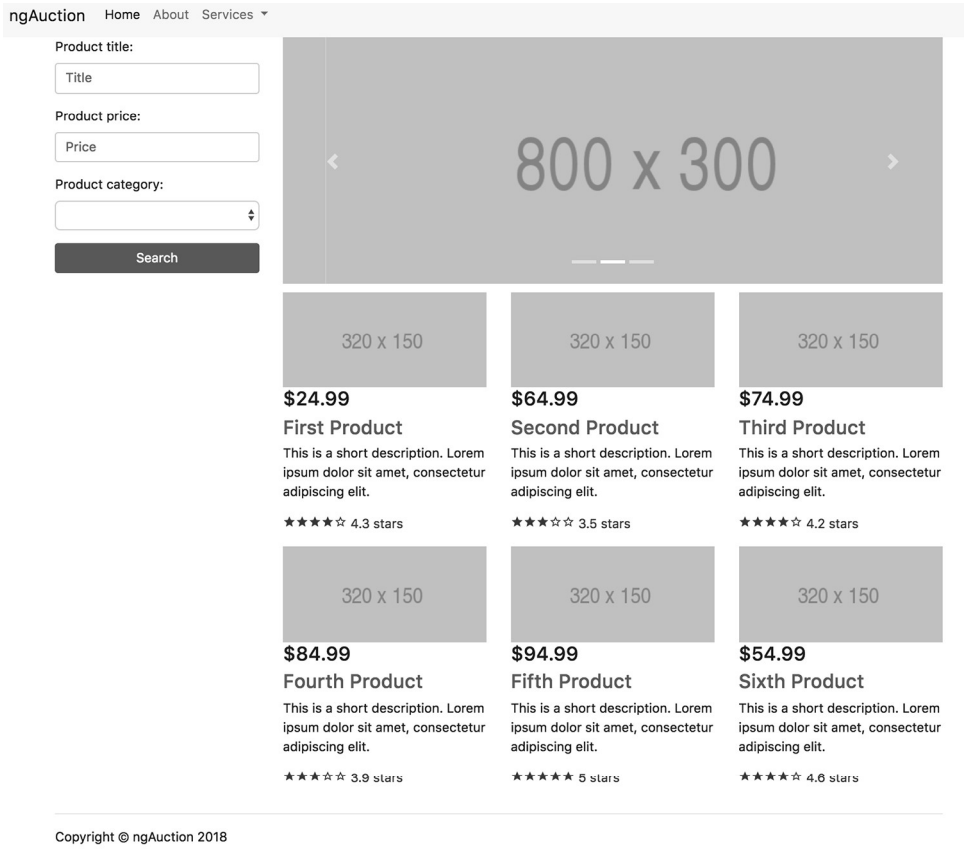


Figure 3.12 The landing page of ngAuction with products

Your `ProductService` will contain hardcoded data about the products. Adding `ProductService` as an argument to the constructor of `HomeComponent` will instruct Angular to instantiate and inject the product object into this component.

As a starting point, you'll use the project in the chapter3/ngAuction folder, which, for the most part, is the same as chapter2/ngAuction with one addition: the shared/product.service.ts file contains the code to provide product data.

To start working on this exercise, open the chapter3/ngAuction folder in your IDE and install the project dependencies by running the `npm install` command.

3.7.1 ProductService

`ProductService` contains hardcoded data about products and the API to retrieve them. Let's review the code in product.service.ts shown in the following listing (we removed the majority of the hardcoded data for brevity).

Listing 3.23 product.service.ts

```
export class Product {
   constructor(
    public id: number,
    public title: string,
    public price: number,
    public rating: number,
    public description: string,
    public categories: string[]) {
  }
}

export class ProductService {

  getProducts(): Product[] {
     return products.map(p => new Product(p.id, p.title,
      p.price, p.rating, p.description, p.categories));
  }

  getProductById(productId: number): Product {
     return products.find(p => p.id === productId);
  }
}

const products = [
   {
    'id': 0,
    'title': 'First Product',
    'price': 24.99,
    'rating': 4.3,
    'description': 'This is a short description.
    ➥Lorem ipsum dolor sit amet, consectetur adipiscing elit.',
    'categories': ['electronics', 'hardware']
  },
  {
    'id': 1,
    'title': 'Second Product',
    'price': 64.99,
```

The Product instances will be returned by the methods of ProductService.

The class ProductService offers an API to get products.

This method returns all hardcoded products.

This method returns one product based on productId.

An array with hardcoded products

```
    'rating': 3.5,
    'description': 'This is a short description.
    ➥Lorem ipsum dolor sit amet, consectetur adipiscing elit.',
    'categories': ['books']
  }];
  }
```

Shortly, you'll be adding the code that will have Angular create an instance of the `ProductService` class and inject it into `ProductItemComponent` and `ProductDetailComponent` so they can invoke the `getProducts()` and `getProductById()` methods on the service.

3.7.2 ProductItemComponent

Figure 3.13 shows six products, each an instance of `ProductItemComponent`.

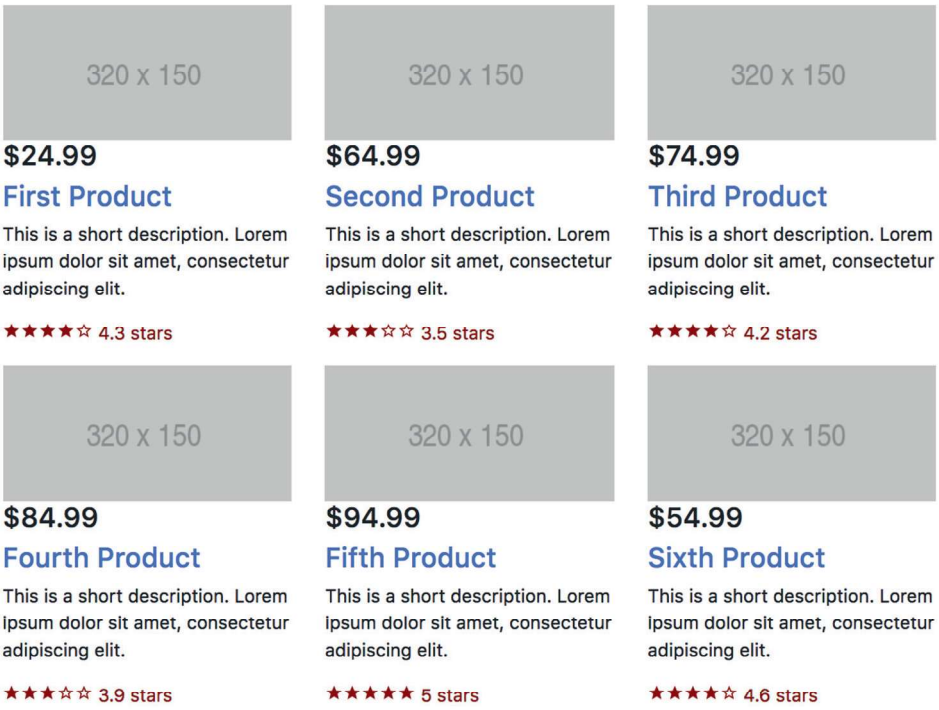


Figure 3.13 Six instances of `ProductItemComponent`

`ProductItemComponent` knows how to render one product based on the product provided by its parent (`HomeComponent`). Modify product-item.component.ts to look like the following listing.

Listing 3.24 product-item.component.ts

```
import {Component, Input} from '@angular/core';
import {Product} from '../shared/product.service';

@Component({
  selector: 'nga-product-item',
  templateUrl: './product-item.component.html',
  styleUrls: ['./product-item.component.css']
})
export class ProductItemComponent {

  @Input() product: Product;
 }
```

This input property will receive the product from the parent component.

The product to render will be given to `ProductItemComponent` via its property `product` decorated with `@Input()`. The `@Input()` properties are described in section 8.2.1. in chapter 8.

Modify product-item.component.html with the content shown in the following listing.

Listing 3.25 product-item.component.html

```
<div class="thumbnail">
  <img src="http://placehold.it/320x150">
  <div class="caption">
    <h4 class="float-right">{{product.price | currency}}</h4>
     <h4><a [routerLink]="['/products', product.id]">{{product.title}}</a>
     </h4>
    <p>{{product.description}}</p>
  </div>
  <!--  <div class="ratings">
     <nga-stars [rating]="product.rating"></nga-stars>
  </div> -->
</div>
```

Applying the currency pipe for formatting

The product title becomes a link to navigate to product details.

The rating component is commented out and will be added later.

Note that you bind the product's `price`, `title`, and `id` properties in the component's template. You also use an Angular built-in pipe, `currency`, for price formatting. For now, you'll keep the `<nga-stars>` component commented out because the code of the `StarsComponent` isn't ready yet. Note that `product.title` is a `routerLink` that will navigate to `ProductDetailComponent` when the user clicks it. The instance of `ProductItemComponent` will be hosted by `HomeComponent`, which you'll update next.

3.7.3 HomeComponent

The home component will

- Use the injected `ProductService` to retrieve all featured products and store them in the `products` array.
- Render the `ProductItemComponent` for each product located in the `products` array.

In section 2.7.8 in chapter 2, you implemented the first version of the `HomeComponent` and added the carousel to its template. Now, you need to modify the constructor to inject the `ProductService` and retrieve the products in the `ngOnInit()` method. Modify the code in home.component.ts to look like the following listing.

Listing 3.26 home.component.ts

```
import {Component, OnInit} from '@angular/core';
import {Product, ProductService} from '../shared/product.service';

@Component({
  selector: 'nga-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {

  products: Product[]=[];
  constructor(private productService: ProductService) { }

  ngOnInit() {
     this.products = this.productService.getProducts();
   }
}
```

Injecting the ProductService

Implementing the lifecycle method ngOnInit() that's invoked after the constructor

Using the ProductService to retrieve products

When Angular instantiates `HomeComponent`, it injects the instance of `ProductService`. Because you used the `private` qualifier, the generated JavaScript will have an instance variable, `productService`.

Angular invokes the component lifecycle method `ngOnInit()` after the constructor, and you invoke the `getProducts()` method there. In section 9.2. in chapter 9, we'll discuss the component lifecycle methods, and you'll see why `ngOnInit()` is the right place for fetching data.

Modify home.component.html to loop through the array `products` with the structural directive `*ngFor` and render each product.

Listing 3.27 home.component.html

```
<div class="row">
  <div class="col-md-12">
    <nga-carousel></nga-carousel>
   </div>
</div>
<div class="row">
  <div *ngFor="let product of products"
            class="col-sm-4 col-lg-4 col-md-4">
    <nga-product-item [product]="product"></nga-product-item>
   </div>
</div>
```

The carousel component goes on top.

Iterating through the array products

Allocates four columns of the Bootstrap flexible grid for each component

Rendering the <nga-product-item> component for each product

Each product is represented by the same HTML fragment on the web page. Because there are multiple products, you need to render the same HTML multiple times. The `NgFor` directive is used inside the component template to loop through the list of items in the data collection, rendering HTML markup for each item. In component templates, `*ngFor` represents the `NgFor` directive.

Because the `*ngFor` directive is located inside a `<div>`, each loop iteration will render a `<div>` with the content of the corresponding `<nga-product-item>` inside. To pass an instance of a product to `ProductItemComponent`, you use the square brackets for property binding:

```
<nga-product-item [product]="product">
```

The `[product]` on the left refers to the property named `product` inside the `<nga-product-item>` component, and `product` on the right is a local template variable declared on the fly in the `*ngFor` directive as `let product`.

The Bootstrap's grid styles `class="col-sm-4 col-lg-4 col-md-4"` instruct the browser to split the width of the `<div>` by evenly allocating 4 columns (out of 12) to each product on small, large, and medium devices, as shown in figure 3.14. Try to remove this class, and see how it affects the UI.

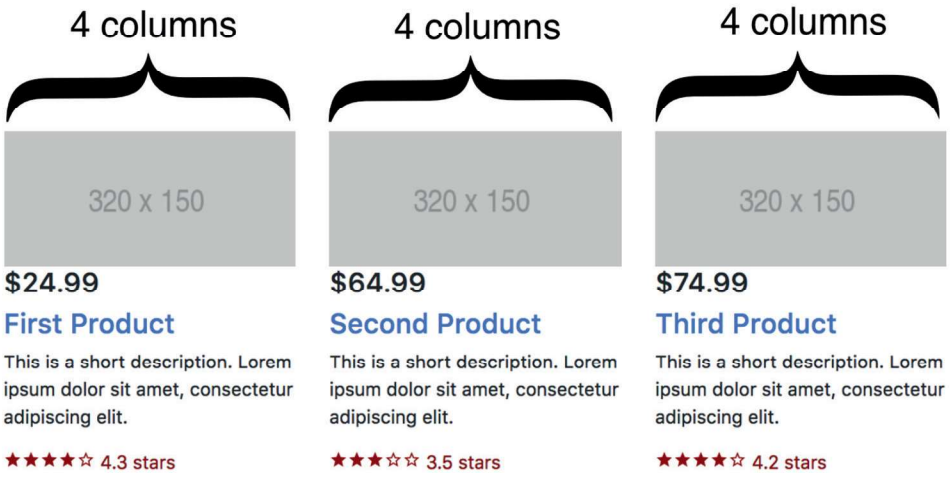


Figure 3.14 Splitting the `<div>` width using Bootstrap grid

Run the `ng serve -o` command , and you'll see six products rendered under the carousel, as shown in figure 3.14, except there won't be any stars with product ratings. We'll take care of the stars next.

3.7.4 StarsComponent

The `StarsComponent` will render stars to display the product rating, as shown in figure 3.15.

Figure 3.15 The `StarsComponent`

On the landing page of ngAuction, the `StarsComponent` will be a child component of `ProductItemComponent`. Eventually, we'll reuse it in the `ProductDetailComponent` as well.

Modify the code of the generated stars.component.ts file to look as follows.

Listing 3.28 stars.component.ts

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

@Component({
  templateUrl: 'stars.component.html',
  styleUrls: ['./stars.component.css'],
  selector: 'nga-stars'
})
export class StarsComponent implements OnInit {
  @Input() count = 5;
   @Input() rating = 0;
  stars: boolean[] = [];

  ngOnInit() {
     for (let i = 1; i <= this.count; i++) {

      this.stars.push(i > this.rating); // push true or false

    }
  }
}
```

Decorates the property count with @Input (covered in section 8.1.2 in chapter 8) so the parent component can assign its value using property binding

Decorates the property rating with @Input for the same reason

Initializes the stars array with Boolean values based on the rating provided by the parent component

Each element of this array corresponds to a single star.

The `count` property specifies the total number of stars to be rendered. If the parent component doesn't provide the value for this input property, five stars will be rendered by default. The `StarsComponent` can render more or fewer stars if need be. The following listing shows how you can render seven stars.

Listing 3.29 Rendering seven stars

```
<nga-stars [rating]="product.rating"
           [count]="7"></nga-stars>
```

Binding 7 to the count input property

Binding the product.rating to the rating input property of the <nga-stars> component

The elements of the `stars` array with the `false` value represent stars without a color, and those with `true` represent stars filled with color. The `rating` property stores the average product rating that determines how many stars should be filled with color and how many should remain empty.

The Bootstrap 4 framework doesn't include images that render stars. There are several popular libraries of icon fonts out there (Material Design Icons, Font Awesome, Octicons, and so on); we'll use Material Design Icons. To keep them local in the project, install these icons as follows:

```
npm i  material-design-icons
```

Then add these fonts to the `styles` section of the .angular-cli.json file so it looks like the following listing.

Listing 3.30 Styles section of .angular-cli.json

```
"styles": [
  "styles.css",
  "../node_modules/bootstrap/dist/css/bootstrap.min.css",
  "../node_modules/material-icons/css/material-design-icons.min.css"
 ]
```

Adding Material Design Icons

Modify the content of stars.component.html to look like the following listing.

Listing 3.31 stars.component.html

```
<p>
  <span *ngFor="let isEmpty of stars"
        class="material-icons">
           {{ isEmpty ? 'star_border' :
                        'star_rate' }}
   </span>
  <span>{{ rating }} stars</span>
</p>
```

Iterates through the Boolean array stars and applies either the filled or empty star

Uses the material-icons style

A Material Design star filled with color is called star_rate.

An empty star style is called star_border.

Note how you bind either one CSS class or another (double curly brackets). To style the star icon, add the following styles to stars.component.css.

Listing 3.32 stars.component.css

```
:host {
  display: block;
}

.material-icons {
   display: inline-block;
  font-size: inherit;
  height: 16px;
  width: 16px;
}
```

Styles the star icon

The `ProductItemComponent` will be the parent of the `StarsComponent`. To make it a child of the `ProductItemComponent`, uncomment the `<div>` in the product-item .component.html file created earlier.

Listing 3.33 Adding a with the <nga-stars> component

```
<div class="ratings">
  <nga-stars [rating]="product.rating"></nga-stars>
</div>
```

Makes the stars dark red (see the CSS in the next listing)

Binding the product.rating to the input property of the <nga-stars> component

The CSS selector `ratings` will be defined in the product-item.component.css file. You'll make the stars dark red and add some padding by using the following listing's style in the product-item.component.css file.

Listing 3.34 product-item.component.css

```
.ratings {
  color: darkred;
}

img {
  width: 100%;
}
```

Setting the color to dark red

Ensures that images won't overlap each other

You add this style to the parent of `StarsComponent` to be able to pick different star colors in the child component if need be. If another component will need to render stars, you can choose another color there. Now your `ProductItemComponent` renders ratings for each child product, as shown in figure 3.14.

It's time to implement navigation with the `Router`.

3.7.5 ProductDetailComponent

In chapter 2, you generated the routing module, but it has only one configured route, which renders `HomeComponent`, as you can see in the following listing.

Listing 3.35 app-routing.module.ts

```
import {NgModule} from '@angular/core';
import {Routes, RouterModule} from '@angular/router';
import {HomeComponent} from "./home/home.component";

const routes: Routes = [
   {
    path: '', component: HomeComponent
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
```

Configuring a default route

Creating a router module and service for the app root module

```
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Reexporting the RouterModule so other modules can access it

You want to add another route so that when the user clicks on the product title in the `ProductItemComponent`, the `Router` replaces `HomeComponent` with `ProductDetailComponent`. During this navigation, you want to pass the ID of the selected product to `ProductDetailComponent`. Modify the routes configuration to look like the following listing, and don't forget to import `ProductDetailComponent`.

Listing 3.36 Configuring a second route

```
const routes: Routes = [
  { path: '', component: HomeComponent},
  { path: 'products/:productId', component: ProductDetailComponent}
];
```

Configuring a route for the URL fragments, like products/123

The `ProductDetailComponent` will receive the selected product ID from the parent via the injected `ActivatedRoute` and then make a request to `ProductService` to retrieve product details. Because `ProductDetailComponent` will reuse the instance of `ProductService` that Angular created for you on app startup, add this service to the constructor's arguments. Modify the code in product-detail.component.ts to look like the following listing.

Listing 3.37 product-detail.component.ts

```
import { Component, OnInit } from '@angular/core';
import {Product, ProductService} from '../shared/product.service';
import {ActivatedRoute} from '@angular/router';

@Component({
  selector: 'auction-product-detail',
  templateUrl: './product-detail.component.html',
  styleUrls: ['./product-detail.component.css']
})
export class ProductDetailComponent implements OnInit {

  product: Product;

  constructor(private route: ActivatedRoute,
              private productService: ProductService) { }

  ngOnInit() {
    const prodId: number = parseInt(
          this.route.snapshot.params['productId']);
    this.product = this.productService.getProductById(prodId);
  }
}
```

Both ActivatedRoute and ProductService are injected into the constructor.

Extracts the parameter productId from the ActivatedRoute

Invokes getProductById() on the service, providing prodId as an argument

The values of the properties of the instance variable `product` will be bound to the component template and rendered by the browser. The template of `ProductDetailComponent` will contain the product image (a gray rectangle) with product price, title, and description.

Modify the content of product-detail.component.html to look like the following listing.

Listing 3.38 product-detail.component.html

```
<div class="thumbnail">
  <img src="http://placehold.it/820x320">
  <div>
    <h4 class="float-right">{{ product.price }}</h4>
    <h4>{{ product.title }}</h4>
    <p>{{ product.description }}</p>
  </div>
  <div class="ratings">
    <p><nga-stars [rating]="product.rating"></nga-stars></p>
  </div>
</div>
```

Renders the product price on the right

Renders the product description

Renders the product title

Renders the StarsComponent with binding on the product rating to the component's property rating

Uses the class ratings to render stars in the dark red color

`ProductDetailComponent` also uses `<nga-stars>`. For a change, let's paint the stars dark green by adding the following listing's style in product-detail.component.css.

Listing 3.39 product-detail.component.css

```
.ratings {
  color: darkgreen;
   padding-left: 10px;
  padding-right: 10px;
}
```

Sets the star color to dark green

Run the app with `ng serve -o`—the navigation to the product-detail view will work. Click the title of the first product, and the `Router` will create an instance of the `ProductDetailComponent`. The browser will show the product details, as shown in figure 3.16.

Run this app with `ng serve -o` to see the landing page of ngAuction. Note that in the product-detail view, the stars are shown in the dark green color, whereas on the landing page, they're dark red.

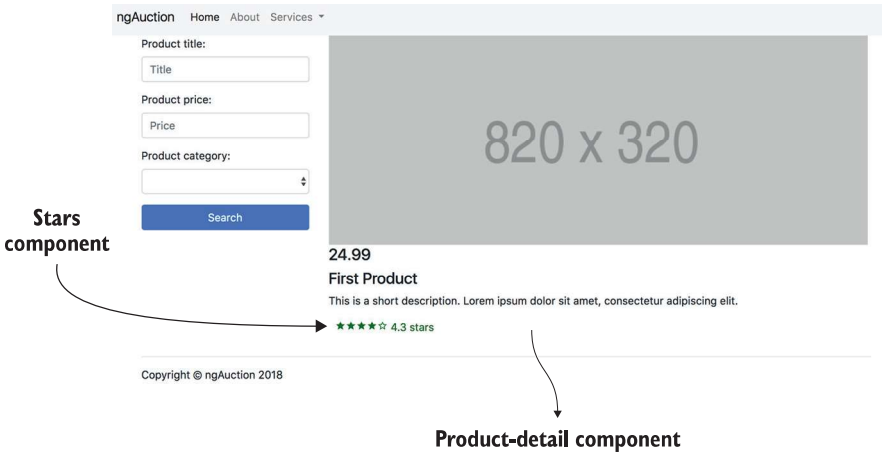


Figure 3.16 Rendering product details

Summary

- You can configure routes in parent and child components.
- You can pass data to routes during navigation.
- During navigation, the router renders components in the area defined by the `<router-outlet>` tag.

4 Router advanced

This chapter covers

- Guarding routes
- Creating components that have multiple router outlets
- Lazy-loading modules

This chapter covers some advanced router features. You'll learn how to use router guards that allow you to restrict access to certain routes, warn the user about unsaved changes, and ensure that important data is retrieved before allowing the user to navigate to a route.

We'll then show you how to create components that have more than one router outlet. Finally, you'll see how to load modules *lazily*—meaning only when the user decides to navigate to certain routes.

This chapter doesn't include the hands-on section for ngAuction. If you're eager to switch from dealing with routing to learning other Angular features, you can skip this chapter and come back to it at a later time.

4.1 Guarding routes

Angular offers several *guard interfaces* that give you a way to mediate navigation to and from a route. Let's say you have a route that only authenticated users can visit.

In other words, you want to guard (protect) the route. Figure 4.1 shows a workflow illustrating how a login guard can protect a route that can be visited only by authenticated users. If the user isn't logged in, the app will render a login view.

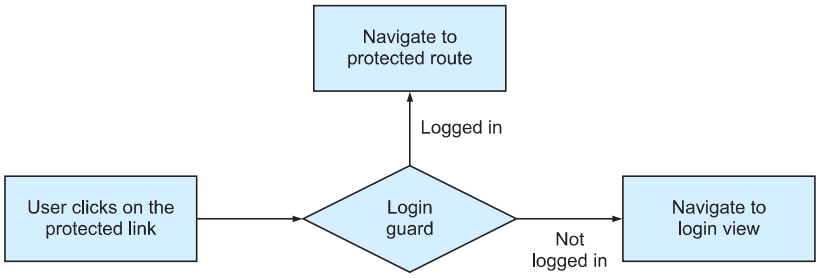


Figure 4.1 A sample login workflow with a guard

Here are some other scenarios where guards can help:

- Open the route only if the user is authenticated and authorized to do so.
- Display a multipart form that consists of several components, and the user is allowed to navigate to the next section of the form only if the data entered in the current section is valid.
- Remind the user about unsaved changes if they try to navigate from the route.
- Allow navigation to the route only after a certain data structure is populated.

These are the guard interfaces:

- `CanActivate` allows or disallows navigation to a route.
- `CanActivateChild` mediates navigation to a child route.
- `CanDeactivate` allows or disallows navigating away from the current route.
- `Resolve` ensures that the required data is retrieved before navigating to a route.
- `CanLoad` allows or disallows lazy-loading modules.

Section 3.3 of chapter 3 mentions that the `Routes` type is an array of items that conforms to the `Route` interface. So far, you've used such properties as `path` and `component` in configuring routes. Now, you'll see how to mediate navigation to or from a route and ensure that certain data is retrieved before navigating to the route. Let's start with adding a guard that'll work when the user wants to navigate to a route.

4.1.1 Implementing the CanActivate guard

Imagine a component with a link that only logged-in users can navigate to. To guard this route, you need to create a new class (for example, `LoginGuard`) that implements the `CanActivate` interface, which declares one method, `canActivate()`. In this method, you implement the validating logic that will return either `true`or `false`. If

`canActivate()` of the guard returns `true`, the user can navigate to the route. You need to assign this guard to the property `canActivate`, as in the following listing.

Listing 4.1 Configuring a route with a `canActivate` guard

```
const routes: Routes = [
    ...
    {path: 'product',
     component: ProductDetailComponent,
     canActivate: [LoginGuard]}
 ];
```

The LoginGuard will mediate navigation to the ProductDetailComponent.

Because `canActivate` properties of `Route` accept an array as a value, you can assign multiple guards if you need to check more than one condition to allow or forbid the navigation.

Let's create a simple app to illustrate how you can protect the `product` route from users who aren't logged in. To keep the example simple, you won't use an authentication service but will generate the login status randomly. The following class implements the `CanActivate` interface. The `canActivate()` function will contain code that returns `true` or `false`. If the function returns `false` (the user isn't logged in), the application won't navigate to the route, will show a warning, and will navigate the user to the login view.

Listing 4.2 login.guard.ts

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

    constructor(private router: Router) {}

    canActivate() {
        // A call to the actual login service would go here
        // For now we'll just randomly return true or false

        let loggedIn = Math.random() < 0.5;

        if (!loggedIn) {
            alert("You're not logged in and will be redirected to Login page");
            this.router.navigate(["/login"]);
        }

        return loggedIn;
    }
}
```

Injects the Router object

Randomly generates the login status

Conditionally displays a "not logged in" message

Redirects to the login page

This implementation of the `canActivate()` function will randomly return `true` or `false`, emulating the user's logged-in status.

The next step is to use this guard in the router configuration. The following listing shows how the routes could be configured for an app that has home and product-detail routes. The latter is protected by `LoginGuard`.

Listing 4.3 Configure one of the routes with a guard

```
export const routes: Routes = [
    {path: '',        component: HomeComponent},
    {path: 'login',   component: LoginComponent},
    {path: 'product', component: ProductDetailComponent,
                      canActivate: [LoginGuard]}
];
```

Adding a guard to the product route

Your `LoginComponent` will be pretty simple—it will show the text "Please login here," as shown in the following listing.

Listing 4.4 login.component.ts

```
@Component({
  selector: 'home',
  template: '<h1 class="home">Please login here</h1>',
  styles: ['.home {background: greenyellow}']
})
export class LoginComponent {}
```

Angular will instantiate the `LoginGuard` class using its DI mechanism, but you have to mention this class in the list of providers that are needed for injection to work. Add the name `LoginGuard` to the list of providers in `@NgModule()`.

Listing 4.5 Adding the guard to the module's providers

```
@NgModule({
    imports:      [BrowserModule, RouterModule.forRoot(routes)],
    declarations: [AppComponent, HomeComponent,
                   ProductDetailComponent, LoginComponent],
    providers:    [LoginGuard]
     bootstrap:    [AppComponent]
})
```

Adds the guard class to the provider's list so Angular can instantiate and inject it

The template of your root component will look like the following listing.

Listing 4.6 The `AppComponent`'s template

```
template: `
        <a [routerLink]="['/']">Home</a>
        <a [routerLink]="['/product']">Product Detail</a>
        <a [routerLink]="['/login']">Login</a>
         <router-outlet></router-outlet>
    `
```

The login page

To see this app in action, run the following command:

```
ng serve --app guards -o
```

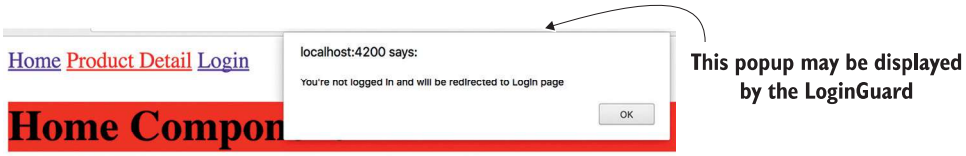


Figure 4.2 Clicking the Product Detail link is guarded

Figure 4.2 shows what happens after the user clicks the Product Detail link, but the `LoginGuard` decides the user isn't logged in.

Clicking OK closes the pop-up window with the warning and navigates to the /login route. In figure 4.2, you implement the `canActivate()` method without providing any arguments to it. But this method can be used with optional parameters:

```
canActivate(destination: ActivatedRouteSnapshot, state: RouterStateSnapshot)
```

The values of `ActivatedRouteSnapshot` and `RouterStateSnapshot` will be injected by Angular automatically, and this may be quite handy if you want to analyze the current state of the router. For example, if you'd like to know the name of the route the user tried to navigate to, this is how you can do it:

```
canActivate(destination: ActivatedRouteSnapshot, state: RouterStateSnapshot)
     {
      console.log(destination.component.name);
   ...
}
```

The `CanActivate` guard controls who gets in, but how you can control whether a user should be allowed to navigate *from* the route? Why do you even need this?

4.1.2 Implementing the CanDeactivate guard

The `CanDeactivate` interface mediates the process of navigating from a route. This guard is quite handy in cases when you want to warn the user that there are some unsaved changes in the view. To illustrate this, you'll update the app from the previous section and add an input field to the `ProductDetailComponent`. If the user enters something in this field and then tries to navigate from this route, your `CanDeactivate` guard will show the "Do you want to save changes" warning, as shown in the following listing.

Listing 4.7 The `ProductDetailComponent` with an input field

```
@Component({
  selector: 'product',
  template: `<h1 class="product">Product Detail Component</h1>
             <input placeholder="Enter your name" type="text"
             [formControl]="name">`,
   styles: ['.product {background: cyan}']
})
```

Binding the variable name to a directive from Forms API

```
export class ProductDetailComponent {
   name: FormControl = new FormControl();
 }
```

Creating an instance of FormControl from Forms API

Listing 4.7 uses the Forms API, which is covered in chapters 10 and 11. At this point, it suffices to know that you create an instance of the `FormControl` class and bind it to the `<input>` element. In your guard, you'll use the `FormControl.dirty` property to know if the user entered anything in the input field. The following listing creates a `UnsavedChangesGuard` class that implements the `CanDeactivate` interface.

Listing 4.8 `UnsavedChangesGuard` implements `CanDeactivate`

```
@Injectable()
export class UnsavedChangesGuard
                  implements CanDeactivate<ProductDetailComponent> {

  canDeactivate(component: ProductDetailComponent) {

    if (component.name.dirty) {
       return window.confirm("You have unsaved changes. Still want to leave?"
     );
    } else {
      return true;
    }
  }
}
```

Implementing the CanDeactivate guard for the ProductDetailComponent

Checking whether the content of the input control has been changed

Implementing canDeactivate() required by the CanDeactivate guard

The `CanDeactivate` interface uses a parameterized type, which you specified using TypeScript generics syntax: `<ProductDetailComponent>`. The method `canDeactivate()` can be used with several arguments (see https://angular.io/api/router/CanDeactivate), but you'll just use one: the component to guard.

If the user entered any value in the input field—`if (component.name.dirty)`—you'll show a pop-up window with a warning. You need to make a couple more additions to the app from the previous section. First, add the `CanDeactivate` guard to the routes configuration.

Listing 4.9 Adding `CanDeactivate` and `CanDeactivate` guards to a route

```
const routes: Routes = [
    {path: '',        component: HomeComponent},
    {path: 'login',   component: LoginComponent},
    {path: 'product', component: ProductDetailComponent,
        canActivate: [LoginGuard],
         canDeactivate: [UnsavedChangesGuard]}
 ];
```

Adding the LoginGuard to the product route

Adding the UnsavedChangesGuard to the product route

The next listing includes the new guard in the providers list in the module.

Listing 4.10 Specifying providers for guards

```
@NgModule({
    ...
    providers: [LoginGuard,
                UnsavedChangesGuard]
})
```

Adding the LoginGuard provider

Adding the UnsavedChangesGuard provider

Run this app (`ng serve --app guards -o`), visit the /product route, and enter something in the input field. Then, try to click another link in the app or the browser's back button. You'll see the message shown in figure 4.3.

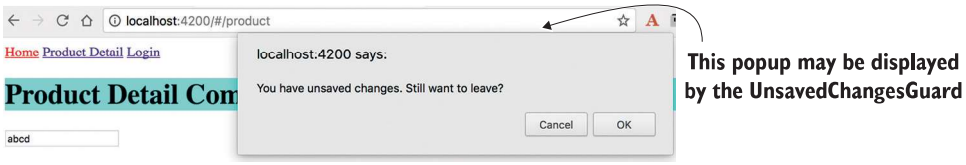


Figure 4.3 Unsaved changes guard in action

Now you know how to control the navigation to and from a route. The next thing is to ensure that the user doesn't navigate to a route too soon, when the data required by the route isn't ready yet.

4.1.3 Implementing the Resolve guard

Let's say you navigate to a product-detail component that makes an HTTP request to retrieve data. The connection is slow, and it takes two seconds to retrieve the data. This means that the user will look at the empty component for two seconds, and then the data will be displayed. That's not a good user experience. What if the server request returns an error? The user will look at the empty component to see the error message after that. That's why it may be a good idea to not even render the component until the required data arrives.

If you want to make sure that by the time the user navigates to a route some data structures are populated, create a `Resolve` guard that allows getting the data *before* the route is activated. A *resolver* is a class that implements the `Resolve` interface. The code in its `resolve()` method loads the required data, and only after the data arrives does the router navigate to the route.

Let's review an app that will have two links: Home and Data. When the user clicks the Data link, it has to render the `DataComponent`, which requires a large chunk of data to be loaded before the user sees this view. To preload the data (a 48 MB JSON file), you'll create a `DataResolver` class that implements the `Resolve` interface. The routes are configured in the following listing.

Listing 4.11 Routes with the resolver

```
const routes: Routes = [
  {path: '',         component: HomeComponent},
  {path: 'mydata', component: DataComponent,
    resolve:{
       loadedJsonData: DataResolver
     }
  }
];
```

Configures the resolve guard for the mydata route

Specifies the class that will preload the data

Note that the `HomeComponent` has no guards. You've configured the `DataResolver` only for the route that renders `DataComponent`. Angular will invoke its `resolve()` method every time the user navigates to the `mydata` route. Because you named the property of the resolve object `loadedJsonData`, you'll be able to access preloaded data in the `DataComponent` using the `ActivatedRoute` object, as follows:

```
activatedRoute.snapshot.data['loadedJsonData'];
```

The code of your resolver is shown next. In this code, you use some of the syntax elements that haven't been covered yet, such as `@Injectable()` (explained in chapter 5), `HttpClient` (chapter 12), and `Observable` (appendix D and chapter 6), but we still want to review this code sample because it's about the router.

Listing 4.12 data.resolver.ts

```
@Injectable()
 export class DataResolver implements Resolve<string[]>{

    constructor ( private httpClient: HttpClient){}

    resolve(): Observable<string[]>{

      return this.httpClient
                 .get<string[]>("./assets/48MB_DATA.json");
     }
}
```

Marks this service as injectable

Injects the HttpClient service to read the data

Implements the resolve() method

Reads the data from the file

Your resolver class is an injectable service that implements the `Resolve` interface, which requires implementing a single `resolve()` method that can return an `Observable`, a `Promise`, or any arbitrary object.

TIP Because a resolver is a service, you need to declare its *provider* (covered in section 5.2 of chapter 5) in the `@NgModule()` decorator.

Here, you use the `HttpClient` service to read the file that contains an array of 360,000 records of random data. The `HttpClient.get()` method returns an `Observable`, and so does your `resolve()` method. Angular generates the code for resolvers that auto-subscribes to the observable and stores the emitted data in the `ActivatedRoute` object.

In the constructor of the `DataComponent`, you extract the data loaded by the resolver and store it in the variable. In this case, you don't display or process the data, because your goal is to show that the resolver loads the data before `DataComponent` is rendered. Figure 4.4 shows the debugger at the breakpoint in the constructor. Note that the data was loaded and is available in the constructor of the `DataComponent`. The UI will be rendered after the code in your constructor completes.

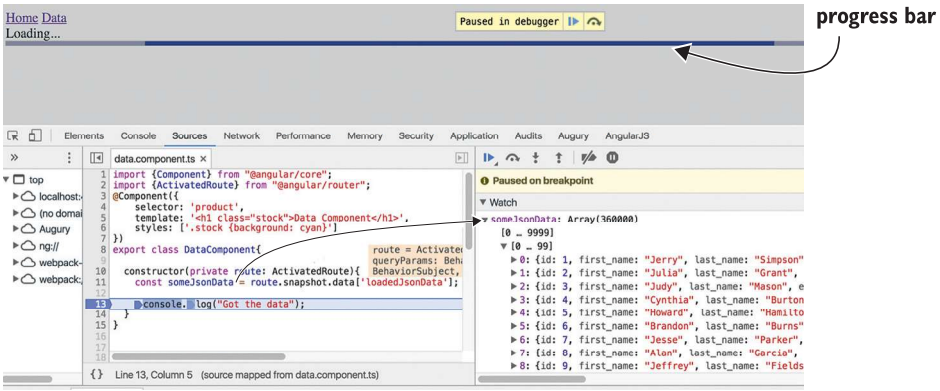


Figure 4.4 The data is loaded.

The source code of this app is located in the directory resolver, and you can see it in action by running `ng serve --app resolver -o`.

Every time you navigate to the `mydata` route, the file will be reloaded and the user will see a progress bar (`mat-progress-bar`) from the Angular Material library of UI components. You'll be introduced to this library in section 5.6 of chapter 5.

The progress bar is used in the template of the `AppComponent`, but how does `AppComponent` know when to start showing the progress bar and when to remove it from the UI? The router triggers events during navigation, such as `NavigationStart`, `NavigationEnd`, and some others. Your `AppComponent` subscribes to these events, and when `NavigationStart` is triggered, the progress bar is displayed, and on `NavigationEnd` it's removed, as shown in the following listing.

Listing 4.13 app.component.ts

```
@Component({
  selector: 'app-root',
  template: `
    <a [routerLink]="['/']">Home</a>
    <a [routerLink]="['mydata']">Data</a>
    <router-outlet></router-outlet>
    <div *ngIf="isNavigating">
       Loading...
      <mat-progress-bar mode="indeterminate"></mat-progress-bar>
    </div>
```

Conditionally shows/hides the progress bar based on the isNavigating flag

```
    `
})
export class AppComponent {

  isNavigating = false;

  constructor (private router: Router){
     this.router.events.subscribe(
       (event) => {
        if (event instanceof NavigationStart){
          this.isNavigating=true;
         }

        if (event instanceof NavigationEnd) {
          this.isNavigating=false;
         }
       }
     );
  }
}
```

Initially the flag isNavigating is false.

Injects the Router object

Subscribes to Router events

Sets the flag to true if NavigationStart is triggered

Sets the flag to false if NavigationEnd is triggered

TIP To avoid reading such a large file over and over again, you can cache the data in memory after the first read. If you're interested in seeing how to do this, review the code of another version of the resolver located in the data.resolver2.ts file. That resolver uses an injectable service from data .service.ts, so on subsequent clicks, instead of the file being read, the data is retrieved from the memory cache. Since the data service is a singleton, it'll survive creations and destructions of the `DataComponent` and cached data remains available.

Reloading the active route

You can reload the route that's already active and rerun its guards and resolvers using the configuration `runGuardsAndResolvers` and `onSameUrlNavigation` options.

Say the user visits the `mydata` route and after some time wants to reload the data *in the same route* by clicking the Data link again. The `routes` configuration in the following listing does this by reapplying the guards and resolvers:

```
const routes: Routes = [
  {path: '',        component: HomeComponent},
  {path: 'mydata', component: DataComponent,
    resolve: {
      mydata: DataResolver
    },
    runGuardsAndResolvers: 'always'
   }
];

export const routing = RouterModule.forRoot(routes,
  {onSameUrlNavigation: "reload"}
   );
```

Runs guards and resolvers always

Reloads the component when the user navigates to the same route

You can read about the other guards in the product documentation at https://angular.io/guide/router#milestone-5-route-guards.

Now, we'll move on to covering another subject: how to create a view that has more than one `<router-outlet>`.

4.2 Developing an SPA with multiple router outlets

The directory ngAuction contains the code of ngAuction that implements the functionality described in chapter 3's hands-on section.

So far, in all routing code samples you've used components that have a single tag, `<router-outlet>`, where Angular renders views based on the configured routes. Now, you'll see how to configure and render views in sibling routes located in the same component. Let's consider a couple of use cases for multi-outlet views:

- Imagine a dashboard-like SPA that has several dedicated areas (outlets), and each area can render more than one component (one at a time). Outlet A can display your stock portfolio, either as a table or as a chart, while outlet B shows either the latest news or an advertisement.
- Say you want to add a chat area to an SPA so the user can communicate with a customer service representative while keeping the current route active as well. You want to add an independent chat route allowing the user to use both routes at the same time and be able to switch from one route to another.

In Angular, you can implement either of those scenarios by having not only a *primary* outlet, but also named *secondary* outlets, which are displayed at the same time as the primary one.

To separate the rendering of components for primary and secondary outlets, you'll need to add yet another `<router-outlet>` tag , but this outlet must have a name. For example, the following code snippet defines primary and chat outlets:

```
<router-outlet></router-outlet>      <-- The primary outlet
 <router-outlet name="chat"></router-outlet>  <-- The secondary (named) outlet
```

Figure 4.5 shows an app with two routes opened at the same time after the user clicks the Home link and then the Open Chat link. The left side shows the rendering of `HomeComponent` in the primary outlet, and the right side shows `ChatComponent` rendered in a named outlet. Clicking the Close Chat link will remove the content of the named outlet (you add an HTML `<input>` field to `HomeComponent` and a `<textarea>` to `ChatComponent` so it's easier to see which component has focus when the user switches between the home and chat routes).

Note the parentheses in the URL of the auxiliary route, http://localhost:4200/#home(aux:chat). Whereas a child route is separated from the parent using the forward slash, an auxiliary route is represented as a URL segment in parentheses. This URL tells you that home and chat are sibling routes.

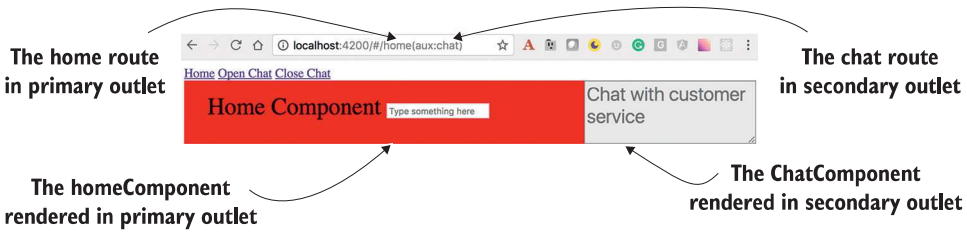


Figure 4.5 Rendering a chat view with a secondary route

The configuration for the chat route specifies the name of the outlet where the `ChatComponent` has to be rendered, shown in the following listing.

Listing 4.14 Configuring routes for two outlets

```
export const routes: Routes = [
  {path: '',  redirectTo: 'home', pathMatch: 'full'},
   {path: 'home', component: HomeComponent},
   {path: 'chat', component: ChatComponent, outlet: "aux"}
 ];
```

Redirects an empty path to the home route

If the URL includes home, renders the HomeComponent in the primary outlet

If the URL includes chat, renders the ChatComponent in the outlet named aux

In this configuration, we wanted to introduce you to the `redirectTo` property. The `HomeComponent` will be rendered in two cases: either by default at the base URL, or if the URL has only the /home segment, as in http:// localhost:4200/home. The `pathMatch: 'full'` means that the client's portion URL must be exactly /, so if you entered the URL http:// localhost:4200/product/home, it wouldn't redirect to `home`.

The template of the app component can look like the following listing.

Listing 4.15 A template of a component that has two outlets

```
template: `
    <a [routerLink]="['']">Home</a>
     <a [routerLink]="['', {outlets: { aux: 'chat'}}]">Open Chat</a>
     <a [routerLink]="[{outlets: { aux: null }}]">Close Chat</a>
     <br/>
    <router-outlet></router-outlet>
     <router-outlet name="aux"></router-outlet>
`
```

A link to navigate to a default route in the primary outlet

A link to navigate to the chat route in the outlet named aux

A link to remove the outlet named aux from the UI

This area is allocated for the secondary outlet named aux.

This area is allocated for the primary outlet.

Note that you have two outlets here: one primary (unnamed) and one secondary (named). When the user clicks the Open Chat link, you instruct Angular to render the component configured for `chat` in the outlet named `aux`. To close a secondary outlet, assign `null` instead of a route name.

If you want to navigate to (or close) the named outlets programmatically, use the `Router.navigate()` method :

```
navigate([{outlets: {aux: 'chat'}}]);
```

To see this app with two router outlets in action, run the following command in the router-samples project:

```
ng serve --app outlets -o
```

There's one more problem the router can help you with. To make the app more responsive, you want to minimize the amount of code that the browser loads to display the landing page of your app. Do you really need to load all the code for each route on application startup?

4.2.1 Lazy-loading modules

Some time ago, one of your authors was working on a website for a European car manufacturer. There was a menu item called "European Delivery" for US citizens, who could fly to the car factory in Europe, pick up their new car there, and spend two weeks driving their own car and enjoying everything that Europe has to offer. After that, the car would be shipped to the United States. Such a trip would cost several thousand dollars, and as you can imagine, not many website visitors would be interested in exploring this option. Then why include the code supporting the menu European Delivery into the landing page of this site, increasing the initial page size?

A better solution would be to create a separate European Delivery module that would be downloaded only if the user clicked the menu item, right? In general, the landing page of a web app should include only the minimal core functionality that must be present when a user visits the site.

Any mid-size or large app should be split into several modules, where each module implements certain functionality (billing, shipping, and so on) and is *lazy-loaded* on demand. In chapter 2, section 2.5.1, you saw an app split into two modules, but both modules were loaded on application startup. In this section, we'll show you how a module can be lazy loaded.

Let's create an app with three links: Home, Product Details, and Luxury Items. Imagine that luxury items have to be processed differently than regular products, and you want to separate this functionality into a feature module called `LuxuryModule`, which will have one component named `LuxuryComponent`. Most users of the app have modest incomes and will rarely click the Luxury Items link, so there's no reason to load the code of the luxury module on application startup. You'll load it *lazily*—only if the user clicks the Luxury Items link. This way of doing things is especially important for mobile apps when they're used in a poor connection area—the code of the root

module has to contain only the core functionality. The code of `LuxuryModule` is shown in the following listing.

Listing 4.16 luxury.module.ts

```
@NgModule({
    imports: [CommonModule,
         RouterModule.forChild([
         {path: '', component: LuxuryComponent}
     ])],
    declarations: [LuxuryComponent]
})

export class LuxuryModule {}
```

Imports CommonModule as required for feature modules

Configures the default route for this feature module using the forChild() method

By default, renders its only component, LuxuryComponent

In the next listing, the code of `LuxuryComponent` just displays the text "Luxury Component" on a yellow (suggesting gold) background.

Listing 4.17 luxury.component.ts

```
@Component({
    selector: 'luxury',
    template: `<h1 class="gold">Luxury Component</h1>`,
     styles: ['.gold {background: yellow}']
 })
export class LuxuryComponent {}
```

Applying the CSS selector gold

Declaring the CSS selector gold

The code of the root module is shown in the following listing.

Listing 4.18 app.module.ts

```
@NgModule({
  imports: [BrowserModule,
    RouterModule.forRoot([
       {path: '',        component: HomeComponent},
      {path: 'product', component: ProductDetailComponent},
      {path: 'luxury', loadChildren: './luxury.module#LuxuryModule'}
     ])
  ],
  declarations: [AppComponent, HomeComponent, ProductDetailComponent],
  providers:[{provide: LocationStrategy, useClass: HashLocationStrategy}],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Configures routes for the root module

Instead of the property component, uses the loadChildren component for lazy loading

Note that the `imports` section only includes `BrowserModule` and `RouterModule`. The feature module `LuxuryModule` isn't listed here. Also, the root module doesn't mention `LuxuryComponent` in its `declarations` section, because this component isn't a part of the root module. When the router parses the routes configuration from both root and feature modules, it'll properly map the `luxury` path to the `LuxuryComponent` that's declared in the `LuxuryModule`.

Instead of mapping the `path` to a component, you use the `loadChildren` property,providing the path and the name of the module to be loaded. Note that the value of `loadChildren` isn't a typed module name, but a string. The root module doesn't know about the `LuxuryModule` type; but when the user clicks the Luxury Items link, the loader module will parse this string and load `LuxuryModule` from the luxury .module.ts file shown earlier.

To ensure that the code supporting `LuxuryModule` isn't loaded on app startup, Angular CLI places its code in a separate bundle. In your project router-samples, this app is configured under the name `lazy` in .angular-cli.json. You can build the bundles by running the following command:

```
ng serve --app lazy -o
```

The Terminal window will print the information about the bundles, as shown in the following listing.

Listing 4.19 The bundles built by ng serve

```
chunk {inline} inline.bundle.js (inline)
chunk {luxury.module} luxury.module.chunk.js () <-- A separate bundle was built for a lazy-loaded bundle.
 chunk {main} main.bundle.js (main) 33.3 kB
chunk {polyfills} polyfills.bundle.js (polyfills)
chunk {styles} styles.bundle.js (styles)
chunk {vendor} vendor.bundle.js (vendor)
```

The second line shows that your luxury module was placed in a separate bundle named luxury.module.chunk.js.

> **NOTE** When you run `ng build --prod`, the names of the bundles for lazy modules are numbers, not names. In the code sample, the default name of the bundle for the luxury module would be zero followed by a generated hash code, something like 0.0797fe80dbf6edcb363f.chunk.js. If your app had two lazy modules, they would be placed in the bundles with the names starting with 0 and 1, respectively.

If you open the browser at localhost:4200 and check the network tab in dev tools, you won't see this module there. See figure 4.6.

Click the Luxury Items link, and you'll see that the browser made an additional request and downloaded the code of the `LuxuryModule`, as seen at the bottom of figure 4.7. The luxury module has been loaded, and the `LuxuryComponent` has been rendered in the router outlet.

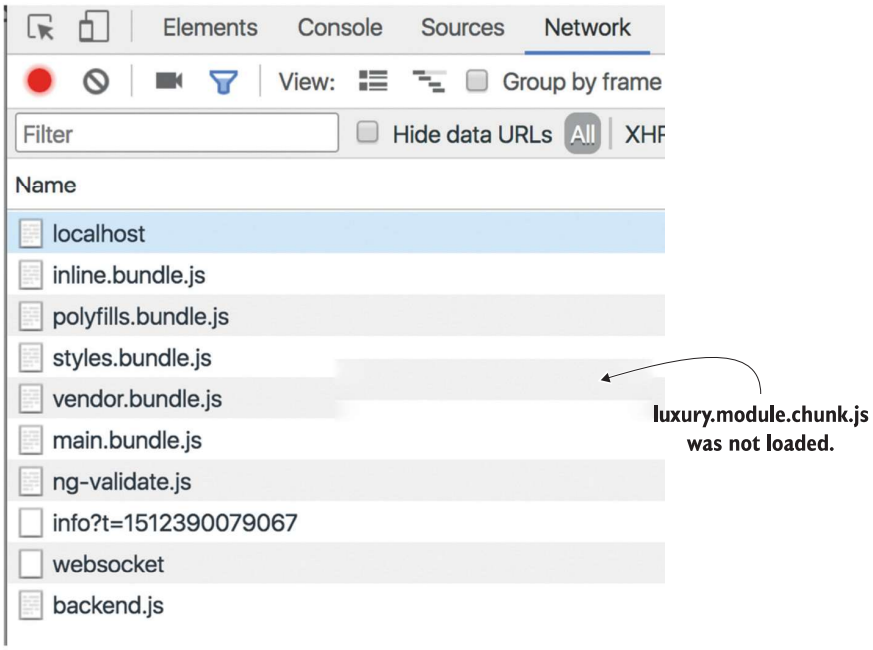


Figure 4.6 Luxury module is not loaded

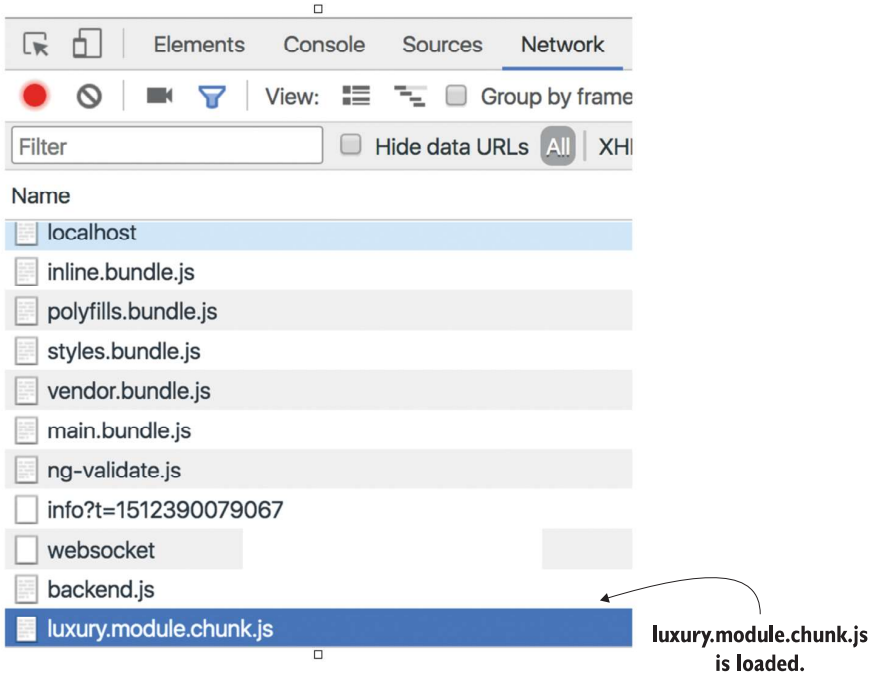


Figure 4.7 Luxury module is loaded after the button click

This simple example didn't substantially reduce the size of the initial download. But architecting large applications using lazy-loading techniques can lower the initial size of the downloadable code by hundreds of kilobytes or more, improving the perceived performance of your application. *Perceived performance* is what the user *thinks* of the performance of your application, and improving it is important, especially when the app is being loaded from a mobile device on a slow network.

On one of our past projects, the manager stated that the landing page of the newly developed web app had to load blazingly fast. We asked, "How fast?" He sent us a link to some app: "As fast as this one." We followed the link and found a nicely styled web page with a menu presented as four large squares. This page did load blazingly fast. After clicking any of the squares, it took more than 10 seconds for the selected module to be lazy loaded. This is perceived performance in action.

TIP Make the root module of your app as small as possible. Split the rest of your app into lazy-loaded modules, and users will praise the performance of your app.

4.2.2 Preloaders

Let's say that after implementing lazy loading, you saved one second on the initial app startup. But when the user clicks the Luxury Items link, they still need to wait this second for the browser to load your luxury module. It would be nice if the user didn't need to wait for that second. With Angular *preloaders*, you can kill two birds with one stone: reduce the initial download time *and* get immediate response while working with lazy-loaded routes.

With Angular preloaders, you can do the following:

- Preload all lazy modules in the background while the user is interacting with your app
- Specify the preloading strategy in the routes configuration
- Implement a custom preloading strategy by creating a class that implements the `PreloadingStrategy` interface

Angular offers a preloading strategy called `PreloadAllModules`, which means that right after your app is loaded, Angular loads all bundles with lazy modules in the background. This doesn't block the application, and the user can continue working with the app without any delays. Adding this preloading strategy as a second argument of `forRoot()` is all it takes, as shown in the following listing.

Listing 4.20 Adding a preloading strategy

```
RouterModule.forRoot([
   {path: '',        component: HomeComponent},
   {path: 'product', component: ProductDetailComponent},
   {path: 'luxury', loadChildren: './luxury.module#LuxuryModule' }
 ],
   {
```

```
    preloadingStrategy: PreloadAllModules
  })
```

Adding the PreloadAllModules preloading strategy

After this code change, the network tab will show that luxury.module.chunk.js was also loaded. Large apps may consist of dozens of lazy modules, and you may want to come up with some custom strategy defining which lazy modules should be preloaded and which shouldn't.

Say you have two lazy modules, `LuxuryModule` and `SuperLuxuryModule`, and you want to preload only the first. You can add some Boolean variable (for example, `preloadme: true`) to the configuration of the `luxury` path:

```
{path: 'luxury', loadChildren: './luxury.module#LuxuryModule', data:
➥{preloadme: true} }
{path: 'luxury', loadChildren: './superluxury.module#SuperLuxuryModule' }
```

Your custom preloader may look like the following listing.

Listing 4.21 A sample custom preloader class

Creates a class implementing the PreloadingStrategy interface

Passes to the preload() method a callback function that returns an Observable

```
@Injectable()
export class CustomPreloadingStrategy implements PreloadingStrategy {

  preload(route: Route,
          load: () => Observable<any>): Observable<any> {
    return (route.data && route.data['preloadme']) ?
        load(): empty();
  }
}
```

No need to preload—returns an empty Observable

Checks the value of the preloadme property on the data object for each route configuration. If it exists and its value is preloadme: true, then invokes the load() callback.

Because `CustomPreloadingStrategy` is an injectable service, you need to add it to the `providers` property of the root module in the `@NgModule` decorator. Don't forget to specify the name of your custom preloader as an argument in the `forRoot()` method.

Summary

- Mediate client-side navigation using guards.
- Create more than one `<router-outlet>` tag in the same component if need be.
- Minimize the initial size of your app by implementing lazy-loading techniques.

5 Dependency injection in Angular

This chapter covers

- Introducing dependency injection as a design pattern
- Understanding how Angular implements DI
- Registering object providers and using injectors
- Adding Angular Material UI components to ngAuction

Chapter 4 discussed the router, and now the ngAuction app knows how to navigate from the home view to the product-detail view. In this chapter, we'll concentrate on how Angular automates the process of creating objects and assembling the application from its building blocks.

An Angular application is a collection of components, directives, and services that may depend on each other. Although each component can explicitly instantiate its dependencies, Angular can do this job using its dependency injection (DI) mechanism.

We'll start this chapter by identifying the problem that DI solves and reviewing the benefits of DI as a software engineering design pattern. Then we'll go over the

specifics of how Angular implements the DI pattern using an example `ProductComponent` that depends on a `ProductService`. You'll see how to write an injectable service and how to inject it into another component.

After that, you'll see a sample application that demonstrates how Angular DI allows you to easily replace one component dependency with another by changing just one line of code. At the end of the chapter, we'll go through a hands-on exercise to build the next version of ngAuction, which uses Angular Material UI components.

Design patterns are recommendations for solving certain common tasks. A given design pattern can be implemented differently depending on the software you use. In the first section, we'll briefly introduce two design patterns: dependency injection and inversion of control (IoC).

5.1 The dependency injection pattern

If you've ever written a function that takes an object as an argument, you already wrote a program that instantiates this object and *injects* it into the function. Imagine a fulfillment center that ships products. An application that keeps track of shipped products can create a `Product` object and invoke a function that creates and saves a shipment record:

```
var product = new Product();
createShipment(product);
```

The `createShipment()` function depends on the existence of an instance of the `Product` object, meaning the `createShipment()` function has a dependency: `Product`. But the function itself doesn't know how to create `Product`. The calling script should somehow create and give (think *inject*) this object as an argument to the function.

Technically, you're decoupling the creation of the `Product` object from its use—but both of the preceding lines of code are located in the same script, so it's not real decoupling. If you need to replace `Product` with `MockProduct`, it's a small code change in this simple example.

What if the `createShipment()` function had three dependencies (such as product, shipping company, and fulfillment center), and each of those dependencies had its own dependencies? In that case, creating a different set of objects for the `createShipment()` function would require many more manual code changes. Would it be possible to ask someone to create instances of dependencies (with their dependencies) for you?

This is what the dependency injection pattern is about: if object A depends on an object identified by a token (a unique ID) B, object A won't explicitly use the `new` operator to instantiate the object that B points at. Rather, it will have B *injected* from the operational environment.

Object A just needs to declare, "I need an object known as B; could someone please give it to me?" Object A doesn't request a specific object type (for example, `Product`) but rather delegates the responsibility of what to inject to token B. It seems that object A doesn't want to be in control of creating instances and is ready to let the framework control this process, doesn't it?

The inversion of control pattern

Inversion of control is a more general pattern than DI. Rather than making your application use some API from a framework (or a software container), the framework creates and supplies the objects that the application needs. The IoC pattern can be implemented in different ways, and DI is one of the ways of providing the required objects. Angular plays the role of the IoC container and can provide the required objects according to your component declarations.

5.2 Benefits of DI in Angular apps

Before we explore the syntax of Angular DI implementation, let's look at the benefits of having objects injected versus instantiating them with a `new` operator. Angular offers a mechanism that helps with registering and instantiating component dependencies. In short, DI helps you write code in a loosely coupled way and makes your code more testable and reusable.

What is injected in Angular

In Angular, you inject services or constants. The *services* are instances of TypeScript classes that don't have a UI and just implement the business logic of your app. *Constants* can be any value. Typically, you'll be injecting either Angular services (such as `Router` or `ActivatedRoute`) or your own classes that communicate with servers. You'll see an example of injecting a constant in section 5.6. A service can be injected either in a component or in another service.

5.2.1 Loose coupling and reusability

Say you have a `ProductComponent` that gets product details using the `ProductService` class. Without DI, your `ProductComponent` needs to know how to instantiate the `ProductService` class. This can be done multiple ways such as by using `new`, calling `getInstance()` on a singleton object, or invoking some factory function `createProductService()`. In any case, `ProductComponent` becomes *tightly coupled* with `ProductService`, because replacing `ProductService` with another implementation of this service requires code changes in `ProductComponent`.

If you need to reuse `ProductComponent` in another application that uses a different service to get product details, you must modify the code, as in `productService = new AnotherProductService()`. DI allows you to decouple application components and services by sparing them from knowing how to create their dependencies.

Angular documentation uses the concept of a *token*, which is an arbitrary key representing an object to be injected. You map tokens to values for DI by specifying providers. A *provider* is an instruction to Angular about *how* to create an instance of an object

for future injection into a target component, service, or directive. Consider the following listing, a `ProductComponent` example that gets the `ProductService` injected.

Listing 5.1 `ProductService` injected into `ProductComponent`

```
@Component({
  providers: [ProductService]
 })
class ProductComponent {
  product: Product;

  constructor(productService: ProductService) {

    this.product = productService.getProduct();
   }
}
```

Specifies the ProductService token as a provider for injection

Injects the object represented by the ProductService token

Uses the API of the injected object

Often the token name matches the type of the object to be injected, so listing 5.1 is a shorthand for instructing Angular to provide a `ProductService` token using the class of the same name. The long version would look like this: `providers:[{provide: ProductService, useClass: ProductService}]`. You say to Angular, "If you see a class with a constructor that uses the `ProductService` token, inject the instance of the `ProductService` class."

Using the `provide`property of `@Component()` or `@NgModule`, you can map the same token to different values or objects (such as to emulate the functionality of `ProductService` while someone else is developing a real service class).

> **NOTE** You already used the `providers` property in chapter 3, section 3.1.2, but it was defined, not on the component, but on the module level in `@NgModule()`.

Now that you've added the `providers` property to the `@Component()` decorator of `ProductComponent`, Angular's DI module will know that it has to instantiate an object of type `ProductService`.

The next question is, when is the instance of the service created? That depends on the decorator in which you specified the provider for this service. In listing 5.1, you specify the provider inside the `@Component()` decorator . This tells Angular to create an instance of `ProductService` when `ProductComponent` is created. If you specify `ProductService` in the `providers` property inside the `@NgModule()` decorator , then the service instance would be created on the app level as a singleton so all components could reuse it.

`ProductComponent` doesn't need to know which concrete implementation of the `ProductService` type to use—it'll use whatever object is specified as a provider. The reference to the `ProductService` object will be injected via the constructor argument, and there's no need to explicitly instantiate `ProductService` in `ProductComponent`.

Just use it as in listing 5.1, which calls the service method `getProduct()` on the `ProductService` instance magically created by Angular.

If you need to reuse the same `ProductComponent` with a different implementation of the `ProductService` type, change the providers line, as in `providers: [{provide: ProductService, useClass: AnotherProductService}]`. You'll see an example of changing an injectable service in section 5.5. Now Angular will instantiate `AnotherProductService`, but the code of `ProductComponent` that uses `ProductService` doesn't require modification. In this example, using DI increases the reusability of `ProductComponent` and eliminates its tight coupling with `ProductService`.

5.2.2 Testability

DI increases the testability of your components in isolation. You can easily inject mock objects if you want to unit test your code. Say you need to add a login feature to your application. You can create a `LoginComponent` (to render ID and password fields) that uses a `LoginService`, which should connect to a certain authorization server and check the user's privileges. While unit testing your `LoginComponent`, you don't want your tests to fail because the authorization server is down.

In unit testing, we often use mock objects that mimic the behavior of real objects. With a DI framework, you can create a mock object, `MockLoginService`, that doesn't connect to an authorization server but rather has hardcoded access privileges assigned to users with certain ID/password combinations. Using DI, you can write a single line that injects `MockLoginService` into your application's login view without needing to wait until the authorization server is ready. Your tests will get an instance of `MockLoginService` injected into your application's login view (as seen in figure 5.1), and your tests won't fail because of issues that you can't control.

NOTE In the hands-on section of chapter 14, you'll see how to unit test injectable services.

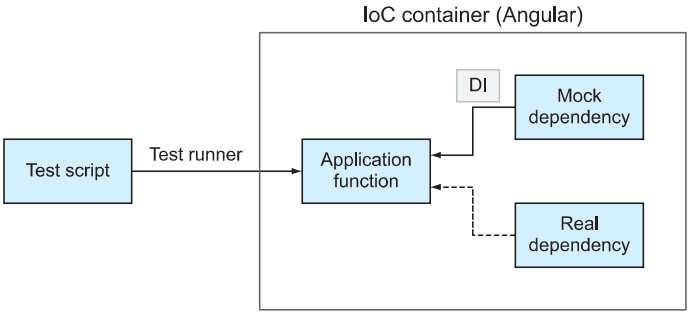


Figure 5.1 DI in testing

5.3 Injectors and providers

Now that you've had a brief introduction to dependency injection as a general, software-engineering design pattern, let's go over the specifics of implementing DI in Angular. In particular, we'll go over such concepts as injectors and providers.

Each component can have an `Injector` instance capable of injecting objects or primitive values into a component. Any Angular application has a root injector available to all of its modules. To let the injector know what to inject, you specify the provider. An injector will inject the object or value specified in the provider into the constructor of a component. Providers allow you to map a custom type (or a token) to a concrete implementation of this type (or value).

> **NOTE** Although eagerly loaded modules don't have their own injectors, a lazy-loaded module has its own subroot injector that's a direct child of the parent's module injector. You'll see an example of injection in a lazy-loaded module in section 5.7.

> **TIP** In Angular, you can inject a service into a class only via its constructor's arguments. If you see a class with a no-argument constructor, it's a guarantee that nothing is injected into this class.

We'll be using `ProductComponent` and `ProductService` in several code samples in this chapter. If your application has a class implementing a particular type (such as `ProductService`), you can specify a provider object for this class on the application level in the `@NgModule()` decorator, like this:

```
@NgModule({
  ...
  providers: [{provide: ProductService, useClass: ProductService}]
})
```

When the token name is the same as the class name, you can use the shorter notation to specify the provider in the module:

```
@NgModule({
  ...
  providers: [ProductService]
})
```

The `providers` line instructs the injector as follows: "When you need to construct an object that has an argument of type `ProductService`, create an instance of the registered class for injection into this object." When Angular instantiates a component that has the `ProductService` token as an argument of the component's constructor, it'll either instantiate and inject `ProductService` or just reuse the existing instance and inject it. In this scenario, we'll have a singleton instance of the service for the entire application.

If you need to inject a different implementation for a particular token, use the longer notation:

```
@NgModule({
  ...
  providers: [{provide: ProductService, useClass: MockProductService}]
})
```

The `providers` property can be specified in the `@Component()` annotation. The short notation of the `ProductService` provider in `@Component()` looks like this:

```
@Component({
  ...
 providers: [ProductService]
})
export class ProductComponent{

    constructor(productService: ProductService) {}
    ...
}
```

You can use the long notation for providers the same way as with modules. If a provider was specified at the component level, Angular will create and inject an instance of `ProductService` during component instantiation.

Thanks to the provider, the injector knows what to inject; now you need to specify *where* to inject the service. With classes, it comes down to declaring a constructor argument specifying the token as its type. The preceding code snippet shows how to inject an object represented by the `ProductService` token. The constructor will remain the same regardless of which concrete implementation of `ProductService` is specified as a provider.

The `providers` property is an array. You can specify multiple providers for different services if need be. Here's an example of a single-element array that specifies the provider object for the `ProductService` token:

```
[{provide: ProductService, useClass: MockProductService}]
```

The `provide` property maps the token to the method of instantiating the injectable object. This example instructs Angular to create an instance of the `MockProductService` class wherever the `ProductService` token is used as a constructor's argument. Angular's injector can use a class or a factory function for instantiation and injection. You can declare a provider using the following properties:

- `useClass`—To map a token to a class, as shown in the preceding example
- `useFactory`—To map a token to a factory function that instantiates objects based on certain criteria
- `useValue`—To map a `string` or a special `InjectionToken` to an arbitrary value (non-class-based injection)

How can you decide which of these properties to use in your code? In the next section, you'll become familiar with the `useClass` property. Section 5.6 illustrates `useFactory` and `useValue`.

5.4 A simple app with Angular DI

Now that you've seen a number of code snippets related to Angular DI, let's build a small application that will bring all the pieces together. This will prepare you to use DI in the ngAuction application.

5.4.1 Injecting a product service

You'll create a simple application that uses `ProductComponent` to render product details and `ProductService` to supply data about the product. If you use the downloadable code that comes with the book, this app is located in the directory di-samples/basic. In this section, you'll build an application that produces the page shown in figure 5.2.

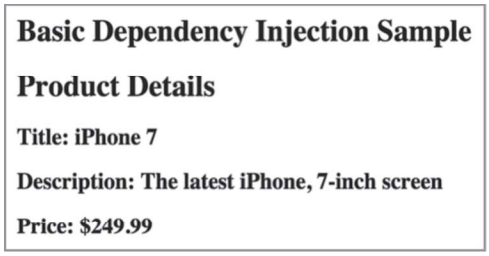


Figure 5.2 A sample DI application

`ProductComponent` can request the injection of the `ProductService` object by declaring the constructor argument with a type:

```
constructor(productService: ProductService)
```

Figure 5.3 shows a sample application that uses these components.

The `AppModule` has a root, `AppComponent`, that includes `ProductComponent`, which is dependent on `ProductService`. Note the `import` and `export` statements. The class definition of `ProductService` starts with the `export` statement, to enable other components to access its content.

The `providers` attribute defined on the component level (refer to figure 5.3) instructs Angular to provide an instance of the `ProductService` class when `ProductComponent` is created. `ProductService` may communicate with some server, requesting details for the product selected on the web page, but we'll skip that part for now and concentrate on how this service can be injected into `ProductComponent`. The following listing implements the components from figure 5.3, starting from the root component.

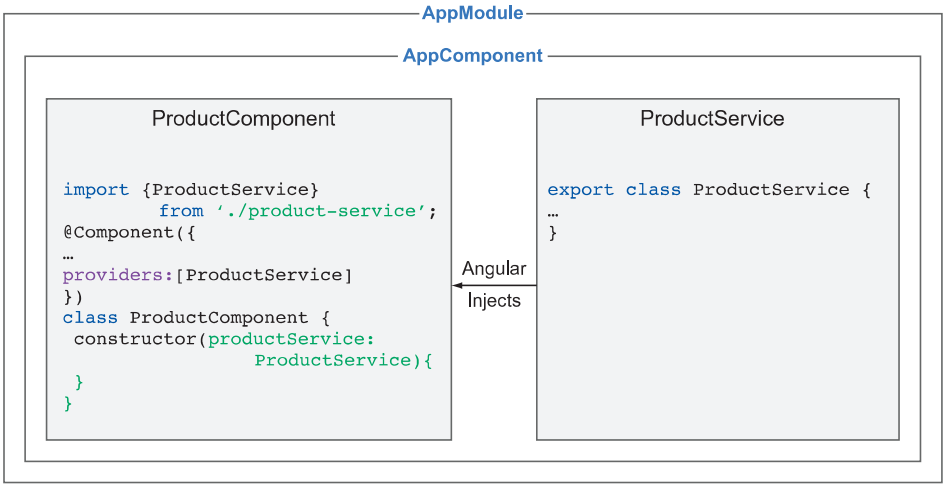


Figure 5.3 Injecting `ProductService` into `ProductComponent`

Listing 5.2 app.component.ts

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

@Component({
  selector: 'app-root',
  template: `<h1>Basic Dependency Injection Sample</h1>
             <di-product-page></di-product-page>`
 })
export class AppComponent {}
```

Including the <di-product-page> component into the template

Based on the `<di-product-page>` tag, you can guess that there's a component with the selector having this value. This selector is declared in `ProductComponent`, whose dependency, `ProductService`, is injected via the constructor, as shown in the next listing.

Listing 5.3 product.component.ts

```
import {Component} from '@angular/core';
import {ProductService, Product} from "./product.service";

@Component({
  selector: 'di-product-page',
   template: `<div>
  <h1>Product Details</h1>
  <h2>Title: {{product.title}}</h2>
  <h2>Description: {{product.description}}</h2>
  <h2>Price: \${{product.price}}</h2>
</div>`,
  providers: [ProductService]
 })
```

Specifying the selector of this component

The short notation of the providers property tells the injector to instantiate the ProductService class.

```
export class ProductComponent {
  product: Product;

  constructor(productService: ProductService) {
    this.product = productService.getProduct();
  }
}
```

Angular instantiates ProductService and injects it here.

In listing 5.3, you use the `ProductService` class as a token for a type with the same name, so you use a short notation without the need to explicitly map the `provide` and `useClass` properties. When specifying providers, you separate the token of the injectable object from its implementation. Although in this case, the name of the token is the same as the name of the type—`ProductService`—the code mapped to this token can be located in a class called `ProductService`, `OtherProductService`, or some other name. Replacing one implementation with another comes down to changing the `providers` line.

The constructor of `ProductComponent` invokes `getProduct()` on the service and places a reference to the returned `Product` object in the `product` class variable, which is used in the HTML template. By using double curly braces, you bind the `title`, `description`, and `price` properties of the `Product` class.

The product-service.ts file includes the declaration of two classes: `Product` and `ProductService`, as you can see in the following listing.

Listing 5.4 product-service.ts

```
export class Product {
   constructor(
    public id: number,
    public title: string,
    public price: number,
    public description: string) {
  }
}

export class ProductService {

  getProduct(): Product {
     return new Product(0, "iPhone 7", 249.99,
           "The latest iPhone, 7-inch screen");
  }
}
```

The Product class represents a product (a value object). It's used outside of this script, so you export it.

For simplicity, the getProduct() method always returns the same product with hardcoded values.

In a real-world application, the `getProduct()` method would have to get the product information from an external data source, such as by making an HTTP request to a remote server.

To run this example, do `npm install` and run the following command:

```
ng serve --app basic -o
```

The browser will open the window, as shown earlier in figure 5.2. The instance of `ProductService` is injected into `ProductComponent`, which renders product details provided by the service.

In the next section, you'll see a `ProductService` decorated with `@Injectable()`, which is required only when the service itself has its own dependencies. It instructs Angular to generate additional metadata for this service. The `@Injectable()` decorator isn't needed in the example because `ProductService` doesn't have any other dependencies injected into it, and Angular doesn't need additional metadata to inject `ProductService` into components.

An alternative DI syntax with @Inject()

In our example, the provider maps a token to a class, and the syntax for injecting is simple: use the constructor argument's type as a token, and Angular will generate the required metadata for the provided type:

```
constructor(productService: ProductService)
```

There's an alternative and more verbose syntax to specify the token using the decorator `@Inject()`:

```
constructor(@Inject(ProductService) productService)
```

In this case, you don't specify the type of the constructor argument, but use the `@Inject()` decorator to instruct Angular to generate the metadata for the `ProductService`. With class-based injection, you don't need to use this verbose syntax, but there are situations where you have to use `@Inject()`, and we'll discuss this in section 5.6.1.

5.4.2 Injecting the HttpClient service

Often, a service will need to make an HTTP request to get necessary data. `ProductComponent` depends on `ProductService`, which is injected using the Angular DI mechanism. If `ProductService` needs to make an HTTP request, it'll have an `HttpClient` object as its own dependency. `ProductService` will need to import the `HttpClient` object for injection; `@NgModule()` must import `HttpClientModule`, which defines `HttpClient` providers. The `ProductService` class should have a constructor for injecting the `HttpClient` object. Figure 5.4 shows `ProductComponent` depending on `ProductService`, which has its own dependency: `HttpClient`.

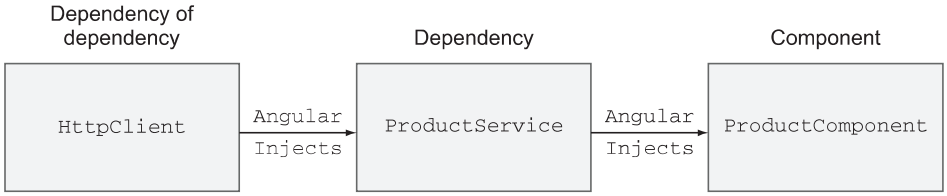


Figure 5.4 A dependency can have its own dependency.

The following listing illustrates the `HttpClient` object's injection into `ProductService` and the retrieval of products from the products.json file.

Listing 5.5 Injecting `HttpClient` into `ProductService`

```
import {HttpClient} from '@angular/common/http';
import {Injectable} from "@angular/core";

@Injectable()
export class ProductService {
    constructor(private http: HttpClient) {
       let products = http.get<string>('products.json')
                      .subscribe(...);
     }
```

Injecting HttpClient

Using HTTP GET

Subscribing to the result of the HTTP request

Because `ProductService` has its own injectable dependency, you need to decorate it with `@Injectable()`. Here, you inject a service into another service. The class constructor is the injection point, but where do you declare the provider for injecting the `HttpClient` type object? All the providers required to inject various flavors of `HttpClient` objects are declared in `HttpClientModule`. You just need to add it to your `AppModule`, as in the following listing.

Listing 5.6 Adding `HttpClientModule`

```
import { HttpClientModule} from '@angular/common/http';
 ...
@NgModule({
  imports: [
    BrowserModule,
    HttpClientModule
   ],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
```

Imports HttpClientModule in the root module

Adds HttpClientModule to the imports section

NOTE Chapter 12 explains how `HttpClient` works.

Starting in Angular 6, the `@Injectable()` decorator allows you to specify the `provideIn` property, which may spare you from explicit declaration of the provider for the service. The following listing shows how you can instruct Angular to automatically create the module-level provider for `ProductService`.

Listing 5.7 Using `provideIn`

```
@Injectable(
  provideIn: 'root'
)
export class ProductService {
...
}
```

Now that you've seen how to inject an object into a component, let's look at what it takes to replace one implementation of a service with another, using Angular DI.

5.5 Switching injectables made easy

Earlier in this chapter, we stated that the DI pattern allows you to decouple components from their dependencies. In the previous section, you decoupled `ProductComponent` from `ProductService`. Now let's simulate another scenario.

Suppose you've started development with a `ProductService` that should get data from a remote server, but the server's feed isn't ready. Rather than modify the code in `ProductService` to introduce hardcoded data for testing, you'll create another class: `MockProductService`.

To illustrate how easy it is to switch from one service to another, you'll create a small application that uses two instances of `ProductComponent`. Initially, the first one will use `MockProductService` and the second, `ProductService`. Then, with a one-line change, you'll make both of them use the same service. Figure 5.5 shows how the app renders two product components that use different implementations of `ProductService`.

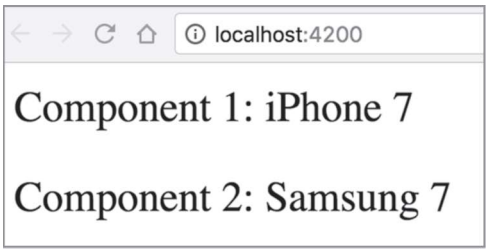


Figure 5.5 Two components and two products

The iPhone 7 product is rendered by `Product1Component`, and the Samsung 7 is rendered by `Product2Component`. This application focuses on switching product services using Angular DI, so we've kept the components and services simple. The app that comes with this chapter has components and services in separate files, but we put all the relevant code in the following listing.

Listing 5.8 Two products and two services

```
// a value object
class Product {
  constructor(public title: string) {}
}

// services
class ProductService {      <-- Bad design
   getProduct(): Product {
    return new Product('iPhone 7');
  }
}

class MockProductService  {      <-- Bad design
```

```
   getProduct(): Product {
    return new Product('Samsung 7');
  }
}

// product components
@Component({
  selector: 'product1',
  template: 'Product 1: {{product.title}}'})
class Product1Component {
  product: Product;

  constructor(private productService: ProductService) {
     this.product = productService.getProduct();
  }
}

@Component({
  selector: 'product2',
  template: 'Product 2: {{product.title}}',
  providers: [{provide: ProductService, useClass: MockProductService}]
 })
class Product2Component {
  product: Product;

  constructor(private productService: ProductService) {
     this.product = productService.getProduct();
  }
}

@Component({
  selector: 'app-root',
  template: `
    <product1></product1>
    <p>
    <product2></product2>
  `
})
class AppComponent {}

@NgModule({
  imports:      [BrowserModule],
  providers:    [ProductService],
   declarations: [AppComponent, Product1Component, Product2Component],
  bootstrap:    [AppComponent]
})
class AppModule { }
```

Since there is no provider declared on this component level, it'll use the app-level provider.

Declares a provider on the component level just for ProductComponent2

ProductComponent2 gets MockProductService because its provider was specified at the component level.

Browser renders two child components of AppComponent

Declares the app-level provider

TIP Listing 5.8 has two lines marked as bad design. Read the sidebar "Program to abstractions" for explanations.

If a component doesn't need a specific `ProductService` implementation, there's no need to explicitly declare a provider for it, as long as a provider was specified at the parent-component level or in `@NgModule()`. In listing 5.8, `Product1Component` doesn't declare its own provider for `ProductService`, and Angular will find one on the application level.

But each component is free to override the `providers` declaration made at the app- or parent-component level, as in `Product2Component`. Each component has its own injector, and during the instantiation of `Product2Component`, this injector will see the component-level provider and will inject `MockProductService`. This injector won't even check whether there's a provider for the same token on the app level.

If you decide that `Product2Component` should get an instance of `ProductService` injected, remove the `providers` line in its `@Component()` decorator.

From now on, wherever the `ProductService` type needs to be injected and no `providers` line is specified on the component level, Angular will instantiate and inject `ProductService`. Running the application after making the preceding change renders the components as shown in figure 5.6.

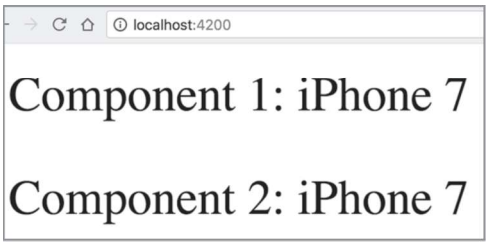


Figure 5.6 Two components and one service

To see this app in action, run the following command:

```
ng serve --app switching -o
```

Imagine that your application had dozens of components using `ProductService`. If each of them instantiated this service with a `new` operator, you'd need to make dozens of code changes. With Angular DI, you're able to switch the service by changing one line in the `providers` declaration.

Program to abstractions

In object-oriented programming, it's recommended to program to interfaces, or *abstractions*. Because the Angular DI module allows you to replace injectable objects, it would be nice if you could declare a `ProductService` interface and specify it as a provider. Then you'd write several concrete classes that implement this interface and switch them in the `providers` declaration as needed.

You can do this in Java, C#, PHP, and other object-oriented languages. The problem is that after transpiling the TypeScript code into JavaScript, the interfaces are removed, because JavaScript doesn't support them. In other words, if `ProductService` were declared as an interface, the following constructor would be wrong, because the JavaScript code wouldn't know anything about `ProductService`:

```
constructor(productService: ProductService)
```

(continued)

But TypeScript supports abstract classes, which can have some of the methods implemented, and some abstract—declared but not implemented. Then, you'd need to implement some concrete classes that extend the abstract ones and implement all abstract methods. For example, you can have the classes shown here:

```
export abstract class ProductService{
   abstract getProduct(): Product;
 }

export class MockProductService extends ProductService{
   getProduct(): Product {
    return new Product('Samsung 7');
  }
}

export class RealProductService extends ProductService{
   getProduct(): Product {
    return new Product('iPhone 7');
  }
}
```

Declares an abstract class

Declares an abstract method

Creates the first concrete implementation of the abstract class

Creates the second concrete implementation of the abstract class

Note that If your abstract class doesn't implement any methods (as in this case), you could use the keyword `implement` instead of `extend`.

The good news is that you can use the name of the abstract class in the constructors, and during the JavaScript code generation, Angular will use a specific concrete class based on the provider declaration. Having the classes `ProductService`, `MockProductService`, and `RealProductService` declared, as in this sidebar, will allow you to write something like this:

```
@NgModule({
  providers: [{provide: ProductService, useClass: RealProductService}],
  ...
})
export class AppModule { }

@Component({...})
export class Product1Component {
  constructor(productService: ProductService) {...};
  }
}
```

Here, you use an abstraction both in the token and in the constructor's argument. This wasn't the case in listing 5.8, where `ProductService` was a concrete implementation of certain functionality. Replacing the providers works the same way as described earlier, if you decide to switch from one concrete implementation of the service to another.

Here, you use an abstraction both in the token and in the constructor's argument. This wasn't the case in listing 5.8, where `ProductService` was a concrete implementation of certain functionality. Replacing the providers works the same way as described earlier, if you decide to switch from one concrete implementation of the service to another.

In listing 5.8, you declared the `ProductService` and `MockProductService` classes as having methods with the same name, `getProducts()`. If you used the abstract-class approach, the TypeScript compiler would give you an error if you'd tried to implement a concrete class but would miss an implementation of one of the abstract methods. That's why two lines in listing 5.8 are flagged as bad design.

What if your component or module can't map a token to a class but needs to apply some business logic to decide which class to instantiate? Furthermore, what if you want to inject just a primitive value and not an object?

5.6 Declaring providers with useFactory and useValue

In general, factory functions are used when you need to apply some application logic prior to instantiating an object. For example, you may need to decide which object to instantiate, or your object may have a constructor with arguments that you need to initialize before creating an instance. Let's modify the app from the previous section to illustrate the use of factory and value providers.

The following listing shows a modified version of `Product2Component`, which you can find in the factory directory of the di-samples app. It shows how you can write a factory function and use it as a provider for injectors. This factory function creates either `ProductService` or `MockProductService`, based on the `boolean` flag `isProd`, indicating whether to run in a production or dev environment, as shown in the following listing.

Listing 5.9 product.factory.ts

```
export function productServiceFactory (isProd: boolean) {
   if (isProd) {
     return new ProductService();
  } else {
    return new MockProductService();
  }
}
```

Injects the value of isProd into the factory function

Instantiates the service based on the value of isProd

You'll use the `useFactory` property to specify the provider for the `ProductService` token. Because this factory requires an argument (a dependency), you need to tell Angular where to get the value for this argument, and you do that using a special property, `deps`, as shown in the following listing.

Listing 5.10 Specifying a factory function as a provider

```
{provide: ProductService,  useFactory: productServiceFactory,
                           deps: ['IS_PROD_ENVIRONMENT']}
```

This function used for instantiating a service

The dependency of this factory function

Here, you instruct Angular to inject a value specified by the `IS_PROD_ENVIRONMENT` token into your factory function. If a factory function has more than one argument, you list the corresponding tokens for them in the `deps` array.

How do you provide a static value for a token represented by a string? You do it by using the `useValue` property. Here's how you can associate the value `true` with the `IS_PROD_ENVIRONMENT` token:

```
{provide: 'IS_PROD_ENVIRONMENT', useValue: true}
```

Note that you map a string token to a hardcoded primitive value, which is not something you'd do in real-world apps. Let's use the environment variables from the environment files generated by Angular CLI in the directory src/environments to find out whether your app runs in dev or production. This directory has two files: environment.prod.ts and environment.ts. Here's the content from environment.prod.ts:

```
export const environment = {
  production: true
};
```

The environment.ts file has similar content but assigns `false` to the `production` environment variable. If you're not using the `--prod` option with `ng serve` or `ng build`, the environment variables defined in environment.ts are available in your app code. When you're building bundles with `--prod`, the variables defined in environment.prod.ts can be used:

```
{provide: 'IS_PROD_ENVIRONMENT', useValue: environment.production}
```

In the environment files you can define as many variables as you need and access them in your application using *dot notation* , as in `environment.myOtherVar`.

The entire code of your app module that uses providers with both `useFactory` and `useValue` is shown in the following listing.

Listing 5.11 Providers with `useFactory` and `useValue`

```
...
import {ProductService} from './product.service';
import {productServiceFactory} from './product.factory';
import {environment} from '../../environments/environment';

@NgModule({
  imports:      [BrowserModule],
  providers: [{provide: ProductService,
```

```
    useFactory: productServiceFactory,
                                              deps: ['IS_PROD_ENVIRONMENT']},
                {provide: 'IS_PROD_ENVIRONMENT',
                useValue: environment.production}],
   declarations: [AppComponent, Product1Component, Product2Component],
   bootstrap:    [AppComponent]
})
export class AppModule {}
```

Maps the productServiceFactory factory function to the ProductService token

Maps the value from the environment file to the IS_PROD_ENVIRONMENT token

Specifies the argument to be injected into the factory function

You can find the complete code of the app that implements `useFactory` and `useValue` as well as the environment variable `production` in the directory called factory of the di-samples project. First, run this app as follows:

```
ng serve --app factory -o
```

In the dev environment, the factory function provides the `MockProductService`, and the browser renders two components showing Samsungs. Now, run the same app in production mode:

```
ng serve --app factory --prod -o
```

This time, the value of `environment.production` is `true`, the factory provides the `ProductService`, and the browser renders two iPhones.

To recap, a provider can map a token to a class, a factory function, or an arbitrary value to let the injector know which objects or values to inject. The class or factory may have its own dependencies, so the providers should specify all of them. Figure 5.7 illustrates the relationships between the providers and the injectors of the sample app.

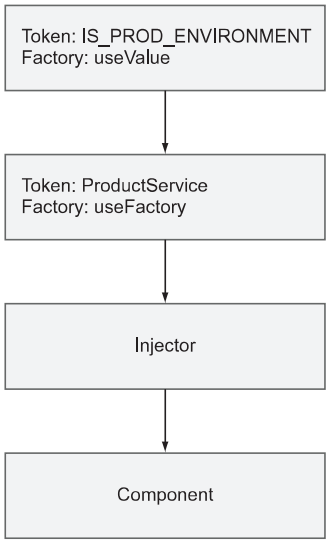


Figure 5.7 Injecting dependencies with dependencies

It's great that you can inject a value into a string token (such as `IS_PROD_ENVIRONMENT`), but this may potentially create a problem. What if your app uses someone else's module that coincidentally also has a token `IS_PROD_ENVIRONMENT` but injects a value with different meaning there? You have a naming conflict here. With JavaScript strings, at any given time there will be only one location in memory allocated for `IS_PROD_ENVIRONMENT`, and you can't be sure what value will be injected into it.

5.6.1 Using InjectionToken

To avoid conflicts caused by using hardcoded strings as tokens, Angular offers an `InjectionToken` class that's preferable to using strings. Imagine that you want to create a component that can get data from different servers (such as dev, production, and QA) and you want to inject the string with the server's URL into a token named `BackendUrl`. Instead of injecting the URL string token, you should create an instance of `InjectionToken`, as shown in the next listing.

Listing 5.12 Using `InjectionToken` instead of a string token

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

export const BACKEND_URL  = new InjectionToken('BackendUrl');

@Component({
  selector: 'app-root',
  template: '<h2>The value of BACKEND_URL is {{url}}</h2>',
  providers: [{provide:BACKEND_URL, useValue: 'http://myQAserver.com'}]
 })
export class AppComponent {
  constructor(@Inject(BACKEND_URL) public url) {}
 }
```

Instantiates InjectionToken

Declares a provider for injecting the value into the token

Injects http://myQAserver.com into the BACKEND_URL token

Here, you wrap the string `BackendUrl` into an instance of `InjectionToken`. Then, in the constructor of this component, instead of injecting a vague string type, you inject a `BACKEND_URL` that points at the concrete instance of `InjectionToken`. Even if the code of another module also has `new InjectionToken('BackendUrl')`, it's going to be a different object.

`BACKEND_URL` isn't a type, so you can't specify your instance of `InjectionToken` as a type of the constructor's argument. You'd get a compilation error:

```
constructor(public url: BACKEND_URL)  // error
```

That's why you didn't specify the argument type of the `AppComponent` constructor but used the `@Inject(BACKEND_URL)` decorator instead to let Angular know which object to inject.

> **TIP** You can't inject TypeScript interfaces, because they have no representation in the transpiled JavaScript code.

You know that providers can be defined on the component and module level, and that module-level providers can be used in the entire app. Things get complicated when your app has more than one module. Will the providers declared in the `@NgModule` of a feature module be available in the root module as well, or will they be hidden inside the feature module?

5.6.2 Dependency injection in a modularized app

Every root app module has its own injector. If you split your app into several eagerly loaded feature modules, they'll reuse the injector from the root module, so if you declare a provider for `ProductService` in the root module, any other module can use it in DI.

What if a provider was declared in a feature module—is it available for the app injector? The answer to this question depends on how you load the feature module.

If a module is loaded eagerly, its providers can be used in the entire app, but each lazy-loaded module has its own injector that doesn't expose providers. Providers declared in the `@NgModule()` decorator of a lazy-loaded module are available within such a module, but not to the entire application. Let's consider two different scenarios: one with a lazy-loaded module and another with an eagerly loaded module.

5.7 Providers in lazy-loaded modules

In this section, you'll experiment with the providers declared inside a lazy-loaded module. You'll start with modifying the app from section 4.3 in chapter 4. This time, you'll add an injectable `LuxuryService` and declare its provider in `LuxuryModule`. The `LuxuryService` will look like the following listing.

Listing 5.13 luxury.service.ts

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

@Injectable()
export class LuxuryService {

  getLuxuryItem() {
     return "I'm the Luxury service from lazy module";
  }
}
```

This service has one method.

The `LuxuryModule` declares the provider for this service, as shown in the following listing.

Listing 5.14 luxury.module.ts

```
@NgModule({
    ...
    declarations: [LuxuryComponent],
     providers: [LuxuryService]
 })

export class LuxuryModule {}
```

This module has one component.

Declares a provider for LuxuryService

Exports the module to make it visible in other modules

The `LuxuryComponent` will use the service, as shown in the following listing.

Listing 5.15 luxury.component.ts

```
@Component({
    selector: 'luxury',
    template: `<h1 class="gold">Luxury Component</h1>
               The luxury service returned {{luxuryItem}} `,
    styles: ['.gold {background: yellow}']
})
export class LuxuryComponent {
  luxuryItem: string

  constructor(private luxuryService: LuxuryService) {}

  ngOnInit() {
    this.luxuryItem = this.luxuryService.getLuxuryItem();
   }
}
```

Injects the LuxuryService

Invokes a method on the LuxuryService

Remember, the `AppModule` lazy loads the `LuxuryModule`, as you can see in the next listing.

Listing 5.16 The root module

```
@NgModule({
  imports: [ BrowserModule,
    RouterModule.forRoot([
        ...
        {path: 'luxury',
           loadChildren: './lazymodule/luxury.module#LuxuryModule'} ]
        )
  ],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Specifies the module in quotes for lazy loading

Running this app will lazy load `LuxuryModule`, and `LuxuryComponent` will get `LuxuryService` injected and will invoke its API.

The following listing tries to inject `LuxuryService` into `HomeComponent` from the root module (both modules belong to the same project).

Listing 5.17 home.component.ts

```
import {Component} from '@angular/core';
import {LuxuryService} from "./lazymodule/luxury.service";

@Component({
  selector: 'home',
  template: '<h1 class="home">Home Component</h1>',
  styles: ['.home {background: red}']
})
export class HomeComponent {

  constructor (luxuryService: LuxuryService) {})
 }
```

Injects LuxuryService into a component of another module

You won't get any compiler errors, but if you run this modified app, you'll get the runtime error "No provider for LuxuryService!" The root module doesn't have access to the providers declared in the lazy-loaded module, which has its own injector.

5.8 Providers in eagerly loaded modules

Let's add a `ShippingModule` to the project described in the previous section, but this one will be loaded eagerly. Similar to `LuxuryModule`, `ShippingModule` will have one component and one injectable service called `ShippingService`. You want to see whether the root module can also use the `ShippingService` whose provider is declared in the eagerly loaded `ShippingModule`, shown in the following listing.

Listing 5.18 shipping.module.ts

```
// imports are omitted for brevity
@NgModule({
  imports: [
    CommonModule,
    RouterModule.forChild([
              {path: 'shipping', component: ShippingComponent}
     ])],
  declarations: [ShippingComponent],
  providers: [ShippingService]
})
export class ShippingModule { }
```

Adds a route for a feature module

TIP In section 2.5.1 in chapter 2, `ShippingModule` also included `exports: [ShippingComponent]` in the `@NgModule()` decorator. You had to export the `ShippingComponent` there because it was used in the `AppComponent` template located in `AppModule`. In this example, you use `ShippingComponent` only inside `ShippingModule`, so no export is needed.

`ShippingComponent` gets `ShippingService` injected and will invoke its `getShippingItem()` method that returns a hardcoded text, "I'm the shipping service from the shipping module."

Listing 5.19 shipping.component.ts

```
import {Component, OnInit} from '@angular/core';
import {ShippingService} from './shipping.service';

@Component({
  selector: 'app-shipping',
  template: `<h1>Shipping Component</h1>
             The shipping service returned {{shippingItem}}`,
  styles: []
})
export class ShippingComponent implements OnInit {

  shippingItem: string;

  constructor(private shippingService: ShippingService) {}

  ngOnInit() {
```

Injects ShippingService

```
        this.shippingItem = this.shippingService.getShippingItem();
    }
}
```

Uses ShippingService

Figure 5.8 shows the structure of the project and the content of the root `AppModule`. In line 17, you eagerly load `ShippingModule`, and in line 18, you lazy load `LuxuryModule`.

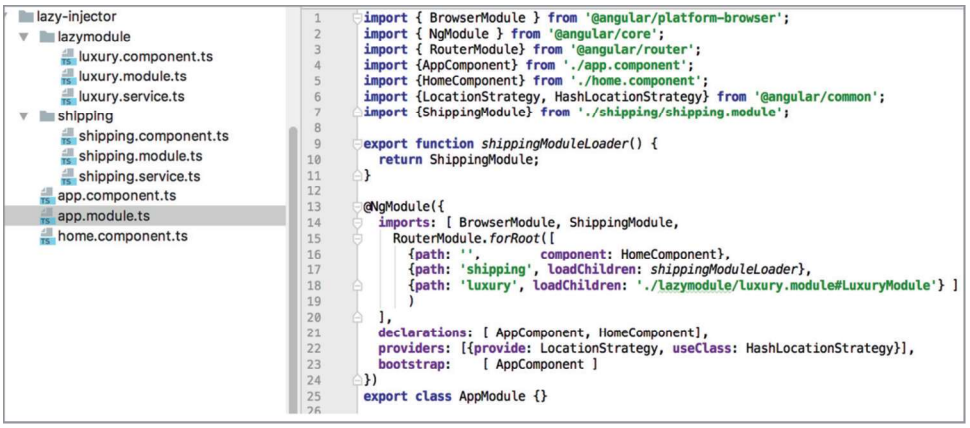


Figure 5.8 An app module that uses feature modules

> **NOTE** By the time you read this, the function in lines 9–11 may not be needed, and line 17 for eager loading the `ShippingModule` could look like this: `{path: 'shipping', loadChildren: () => ShippingModule}`. But at the time of writing, using a function in line 17 results in errors during the AOT compilation.

To see this app in action, run the following command:

```
ng serve --app lazyinjection -o
```

Clicking the Shipping Details link shows the data returned by the `ShippingService`, as shown in figure 5.9. `ShippingService` was injected into `ShippingComponent` even though you didn't declare the provider for `ShippingService` in the root app module.

This proves the fact that providers of eagerly loaded modules are merged with providers of the root module. In other words, Angular has a single injector for all eagerly loaded modules.

Figure 5.9 Navigating to the shipping module

5.9 Hands-on: Using Angular Material components in ngAuction

> **NOTE** Source code for this chapter can be found at https://github.com/Farata/angulartypescript and www.manning.com/books/angular-development-with-typescript-second-edition.

In the hands-on section of chapter 3, you used DI in ngAuction. You added the `ProductService` provider in `@NgModule()`, and this service was injected into `HomeComponent` and `ProductDetailComponent`. In the final version of `ngAuction`, you'll also inject `ProductService` into `SearchComponent`.

In this section, we won't be focusing on DI but rather introducing you to the Angular Material library of modern-looking UI components. The goal is to replace the HTML elements on the landing page of ngAuction with Angular Material (AM) UI components. You'll still keep the Bootstrap library in this version of ngAuction, but starting in chapter 7, you'll do a complete rewrite of ngAuction so it'll use only AM components.

You'll use ngAuction from chapter 3 as a starting point, gradually replacing HTML elements with their AM counterparts, so the landing page will look as shown in figure 5.10.

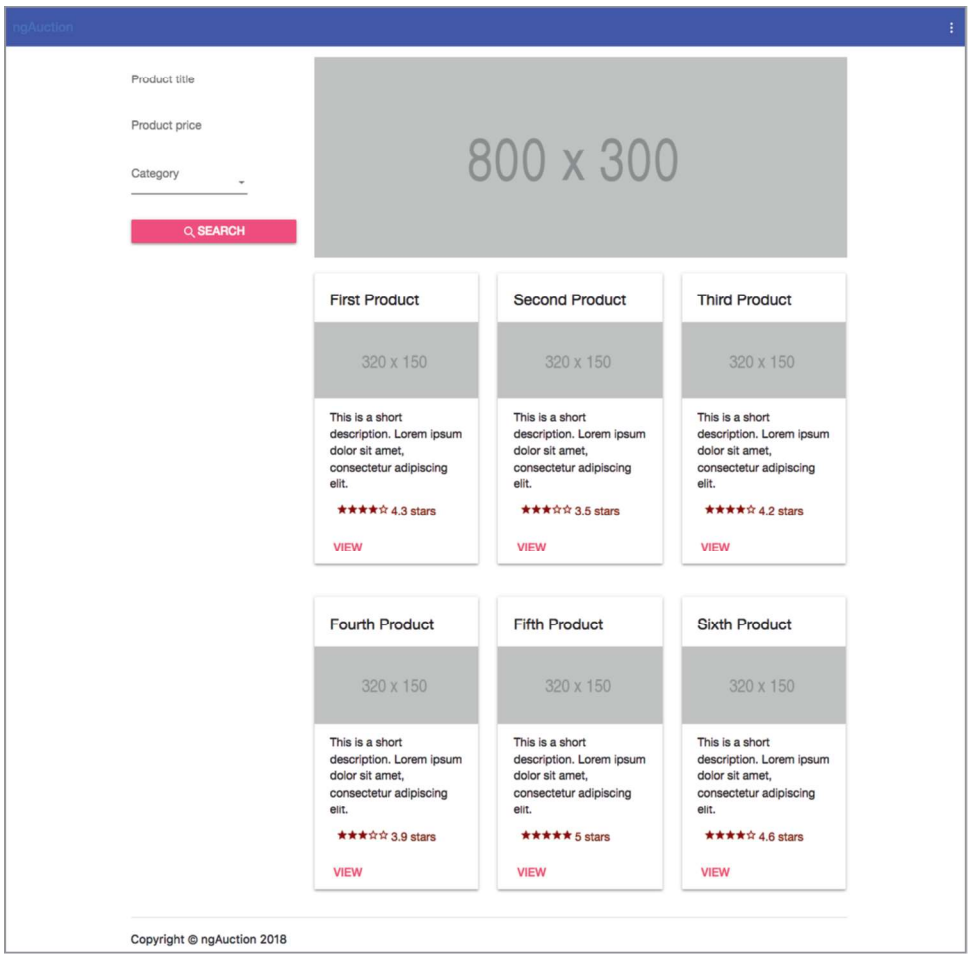


Figure 5.10 ngAuction with Angular Material components

5.9.1 A brief overview of the Angular Material library

Angular Material is a library of UI components developed by Google, based on the Material Design guidelines that define the classic principles of good design and consistent user experience (see https://material.io/guidelines). The guidelines provide suggestions for how the UI for a web or mobile app should be designed.

AM offers more than 30 UI components and four prebuilt themes. A *theme* is a collection of palettes, each of which defines different shades of colors that look good when used together (see https://material.io/guidelines/style/color.html), as seen in figure 5.11.

Figure 5.11 Sample Material Design palettes

The color with the number 500 is a *primary* color for the palette. We'll show you how to customize palettes in the hands-on section of chapter 7. At the time of writing, AM comes with four prebuilt themes: `deeppurple-amber`, `indigo-pink`, `pink-bluegrey`, and `purple-green`. One way to add a theme to your app is by using the `<link>` tag in your index.html:

```
<link href="../node_modules/@angular/material/prebuilt-themes/indigo-
     pink.css" rel="stylesheet">
```

Alternatively, you can add a theme to your global CSS file (styles.css) as follows:

```
@import '~@angular/material/prebuilt-themes/indigo-pink.css';
```

Any app built with AM can specify the following colors for the UI components:

- `primary`—Main colors
- `accent`—Secondary colors
- `warn`—For errors and warnings
- `foreground`—For text and icons
- `background`—For component backgrounds

While styling the UI of your app, for the most part you won't be specifying the color names or codes as it's done in regular CSS. You'll be using one of the preceding keywords.

TIP In the hands-on section of chapter 7, you'll start using the CSS extension SaaS for styling.

The following line shows how to add the AM toolbar component styled with the primary color for whatever theme is specified:

```
<mat-toolbar color="primary"></mat-toolbar>
```

Should you decide to switch to a different theme, there's no need to change the preceding code—the `<mat-toolbar>` will use the primary color of the newly selected theme.

AM components include input fields, radio buttons, checkboxes, buttons, date picker, toolbar, grid list, data table, and more. For the current list of components, refer to product documentation at https://material.angular.io. Some of the components are added to your component templates as tags, and some as directives. In any case, the AM component names begin with the prefix `mat-`.

The following listing shows how create a toolbar that contains a link and a button with an icon.

Listing 5.20 Creating a toolbar with Angular Material

```
<mat-toolbar color="primary">
   <a [routerLink]="['/']">Home</a>

  <button mat-icon-button>
     <mat-icon>more_vert</mat-icon>
    </button>
</mat-toolbar>
```

AM toolbar of primary theme color

A button with an icon

Places the Google Material icon more_vert on the button

Here, you use two AM tags, `<mat-toolbar>` and `<mat-icon>`, and one directive, `mat-icon-button`. Each AM component is packaged in a feature module, and you'll need to import the modules for the required AM components in the `@NgModule()` decorator of your `AppModule`. You'll see how to do this while giving a face lift to your ngAuction.

TIP If you want to build the new version of ngAuction on your computer, copy the directory chapter3/ngAuction into another location, run `npm install` there, and follow the instructions in the next sections.

5.9.2 Adding the AM library to the project

First, you need to install three modules required by the AM library by running the following commands in the project root directory:

```
npm i @angular/material @angular/cdk @angular/animations
```

In this version of ngAuction, you'll use the prebuilt `indigo-pink` theme, so replace the content of the styles.css file with this line:

```
@import '~@angular/material/prebuilt-themes/indigo-pink.css';
```

TIP Starting from Angular CLI 6, you can add the AM library to your project with one command: `ng add @angular/material`. This command will install the required packages and modify the code in several files of your app, so you have less typing to do. We didn't use this command here because we want to keep all AM components used in this app in a separate feature module.

5.9.3 Adding a feature module with AM components

AM components are packaged as feature modules, and you should add only those modules that your app needs rather than adding the entire content of the AM library. You can either add the required modules to the root app module or create a separate module and list all required components there.

In this version of ngAuction, you'll keep the UI components for ngAuction in a separate module. Generate a new `AuctionMaterialModule` by running the following command:

```
ng g m AuctionMaterial
```

This command will generate boilerplate of a feature module in the app/auction-material/auction-material.module.ts file. Modify the code of this file to look like the following listing.

Listing 5.21 A feature module for AM UI components

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import {MatToolbarModule} from '@angular/material/toolbar';
import {MatIconModule} from '@angular/material/icon';
import {MatMenuModule} from '@angular/material/menu';
import {MatButtonModule} from '@angular/material/button';
import {MatInputModule} from '@angular/material/input';
import {MatSelectModule, } from '@angular/material/select';
import {MatCardModule} from '@angular/material/card';
import {MatFormFieldModule} from '@angular/material/form-field';
import {BrowserAnimationsModule}
               from '@angular/platform-browser/animations';

@NgModule({
  imports: [
    CommonModule
   ],
  exports: [
    MatToolbarModule, MatIconModule, MatMenuModule, MatButtonModule,
    MatInputModule, MatSelectModule, MatCardModule,
    MatFormFieldModule, BrowserAnimationsModule
  ]
})
export class AuctionMaterialModule { }
```

Imports only those AM modules that ngAuction needs

This module declares providers for animation services.

Reexports the AM modules so they can be used in other modules of ngAuction

This is a feature module, so import the CommonModule.

Now, open app.module.ts (the root module of ngAuction) and add your `AuctionMaterialModule` feature module to the `imports` property of `@NgModule()`, as you see in the following listing.

Listing 5.22 Adding the AM feature module

```
import {AuctionMaterialModule} from "./auction-material/auction-
     material.module";
...
@NgModule({
  ...
  imports: [
    ...
    AuctionMaterialModule
   ]
  ...
})
```

Adds the AM feature module to the root one

Now's a good time to build and run ngAuction:

```
ng serve -o
```

You won't see any changes in the ngAuction UI just yet, but keep the dev server running so the appearance of the landing page will gradually change as you add more code in the next sections.

5.9.4 Modifying the appearance of NavbarComponent

The navbar component is a black bar with a menu. You'll start by replacing the existing content of navbar.component.html to use `<mat-toolbar>`, which will eventually contain the menu of the auction. Remove the current content of this file and add an empty toolbar there:

```
<mat-toolbar color="primary"></mat-toolbar>
```

While making changes, keep an eye on the UI of your running ngAuction—it has an empty blue toolbar now. You want the toolbar to contain the link to the home page and a pop-up menu that will be activated by a button click. The button should contain an icon with three vertical dots (see figure 5.14), and the directive `mat-icon-button` turns a regular button into a button that can contain `<mat-icon>`. For the image, you'll use `more_vert`, which is the name of one of the Google material icons available for free at https://material.io/icons.

Add the link and the button by modifying the content of navbar.component.html to match the following listing.

Listing 5.23 Adding a link and an icon button to the toolbar

```
<mat-toolbar color="primary">
<a [routerLink]="['/']">ngAuction</a>
<button mat-icon-button >
```

Adds a link to navigate to a default route

Adds a button that looks like three vertical dots

```
    <mat-icon>more_vert</mat-icon>
</button>

</mat-toolbar>
```

Now the toolbar will look like figure 5.12.

Figure 5.12 A toolbar with a broken icon

You specified the name of the icon `more_vert`, but didn't add Google material icons to index.html. Add the following to the `<head>` section of index.html:

```
<link href="https://fonts.googleapis.com/icon?family=Material+Icons"
       rel="stylesheet">
```

Now the `more_vert` icon is properly shown on the button, as shown in figure 5.13.

Figure 5.13 A toolbar with a fixed icon

The next step is to push this button to the right side of the toolbar, regardless of screen width. You'll add a `<div>` between the link and the button to fill the space. Add the following style to navbar.component.css:

```
.fill {
  flex: 1;
}
```

By default, the toolbar has the CSS flexbox layout (see https://css-tricks.com/snippets/css/a-guide-to-flexbox). The style `flex:1` translates to "Give the entire width to the HTML element."

Place the `<div>` between the `<a>` and `<button>` tags in navbar.component.html:

```
<div class="fill"></div>
```

Now the button is pushed all the way to the right, as shown in figure 5.14.

Figure 5.14 Pushing the button to the right

At this point, clicking the button doesn't open a menu for two reasons:

- You haven't created a menu yet.
- You haven't linked the menu to the button.

The ngAuction app from chapter 3 had three links: About, Services, and Contacts. Let's turn them into a pop-up menu. Each menu item will have an icon (`<mat-icon>`) and text. In Angular Material, a menu is represented by `<mat-menu>`, which can contain one or more items, such as `<button mat-menu-item>` components.

Add the code in the following listing right after the `</mat-toolbar>` tag in navbar.component.html.

Listing 5.24 Declaring items for a pop-up menu

```
<mat-menu #menu="matMenu">      <---- Uses the AM menu control
   <button mat-menu-item>       <---- First menu item
     <mat-icon>info</mat-icon>
    <span>About</span>
  </button>
  <button mat-menu-item>        <---- Second menu item
     <mat-icon>settings</mat-icon>
    <span>Services</span>
  </button>
  <button mat-menu-item>        <---- Third menu item
     <mat-icon>contacts</mat-icon>
    <span>Contact</span>
  </button>
</mat-menu>
```

Each `<mat-icon>` uses one of the Google Material icons (`info`, `settings`, and `contacts`). Note that you declare a local template variable, `#menu`, to reference this menu and assigned it to the AM `matMenu` directive. In itself, `<mat-menu>` doesn't render anything until it's attached to a component with the `matMenuTriggerFor` directive. To attach this menu to your toolbar button, bind the `menu` template variable to the `matMenuTriggerFor` directive . Update the button to look as follows:

```
<button mat-icon-button [matMenuTriggerFor]="menu">
  <mat-icon>more_vert</mat-icon>
</button>
```

NOTE You can replace `<button>` tags with `<a [routerLink]>` links.

If you click the toolbar button now, it'll show the menu, as shown in figure 5.15.

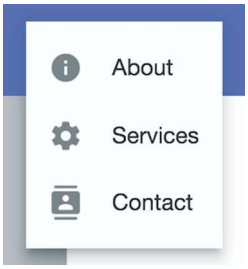


Figure 5.15 The toolbar menu

5.9.5 Modifying the SearchComponent UI

The `SearchComponent` template will contain a form with three controls: a text input, a number input, and a select dropdown, which will be implemented with the `matInput`

directives (they should be placed inside `<mat-form-field>`) and `<mat-select>`. To make the Search button stand out, you'll also add a `mat-raised-button` directive and the search icon to this button.

Modify the code in the search.component.html file to look like the following listing.

Listing 5.25 search.component.html

```
<form #f="ngForm">                                  <-- Uses the template-driven Forms API
  <mat-form-field>                                  <-- First form field
     <input matInput
            type="text"
            placeholder="Product title"
            name="title" ngModel>
  </mat-form-field>

  <mat-form-field>                                  <-- Second form field
      <input matInput
            type="number"
            placeholder="Product price"
            name="price" ngModel>
  </mat-form-field>

  <mat-form-field>                                  <-- Third form field
     <mat-select placeholder="Category" name="category" ngModel>
      <mat-option *ngFor="let c of categories"
                  [value]="c">{{ c }}</mat-option>
    </mat-select>
  </mat-form-field>

  <button mat-raised-button color="accent" type="submit">   <-- The form's Submit button
     <mat-icon>search</mat-icon>SEARCH
  </button>
</form>
```

The `ngForm` and `ngModel` directives are parts of template-driven forms defined in the `FormModule` (described in section 10.2.1 of chapter 10), and you need to add it to the `@NgModule()` decorator in `AppModule`, as shown in the following listing.

Listing 5.26 Adding support for the Forms API

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

@NgModule({
  ...
  imports: [
   ...
    FormsModule                                     <-- Adds support for the template-driven Forms API
   ]
})
```

Let's make sure the UI is properly rendered. For now, on smaller screens, it looks like figure 5.16.

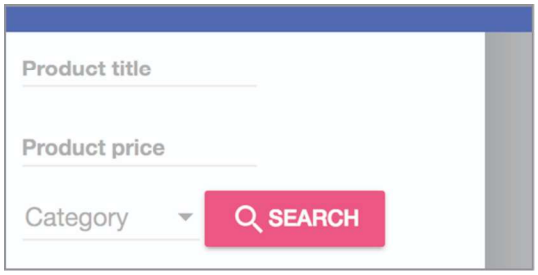


Figure 5.16 The search form with a misaligned button

The `mat-form-field` components and the `mat-select` dropdown should occupy the entire width of the search component. You also want to add more space between the form controls.

Add the styles in the following listing to the search.component.css file.

Listing 5.27 A fragment of search.component.css

```
mat-form-field, mat-select, [mat-raised-button] {
  display: block;
  margin-top: 16px;
  width: 100%;
}
```

`display: block;` tells the browser to render the search component as a standard `<div>`. Now the search form is well aligned, as shown in figure 5.17.

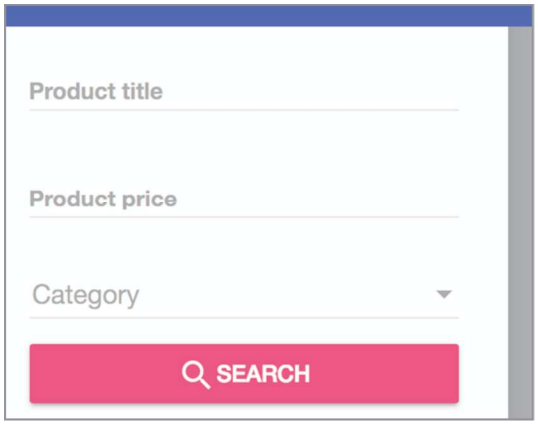


Figure 5.17 The search form

The Search button won't perform search in this version of ngAuction, and the search form won't do input validation either. You'll fix this in section 11.8 in chapter 11 after we discuss the Angular Forms API.

5.9.6 Replacing the carousel with an image

At the time of writing, Angular Material doesn't have a carousel component. In a real-world project, you'd find the carousel component in one of the third-party libraries, such as the PrimeNG library (www.primefaces.org/primeng/#/carousel), but in this version of ngAuction, you'll replace the carousel with a static image.

Replace the content of carousel.component.html with the following code:

```
<img src="http://placehold.it/800x300" alt="Banner">
```

Now the browser shows a gray rectangle in place of the carousel. You could have kept the Bootstrap carousel in place, but it's not worth loading the entire Bootstrap library just for the carousel. The goal is to gradually switch to AM components.

5.9.7 More fixes with spacing

Let's put some space between the toolbar and other components by adding the following style to app.component.css:

```
.container {
  margin-top: 16px;
}
```

Now, add some space between the carousel and product items. Modify the home.component.css file to look like the following code (`display:block` is for rendering this custom component as a `<div>`):

```
:host {
  display: block;
}

auction-carousel {
  margin-bottom: 16px;
}
```

5.9.8 Using mat-card in ProductItemComponent

The next step is to display your products as tiles, and each `<nga-product-item>` will use the `<mat-card>` component. To render each product inside the card, modify the content of the product-item.component.htmlfile to look like the following listing.

Listing 5.28 product-item.component.html

Defines the content of the AM <mat-card> component

The product title goes on top.

```
<mat-card>
   <mat-card-title>{{ product.title }}</mat-card-title>
   <img mat-card-image src="http://placehold.it/320x150">
   <mat-card-content>
    {{product.description}}
   </mat-card-content>
  <mat-card-actions>
    <a mat-button color="accent"
```

Product image

Product description

```
        [routerLink]="['/products', product.id]">VIEW</a>
    </mat-card-actions>
</mat-card>
```

A link to navigate to product details

5.9.9 Adding styles to HomeComponent

Your `HomeComponent` hosts several instances of `ProductItemComponent`. Now let's add more styles to home.component.css so the products are displayed nicely and aligned using the CSS style `flex`. Add the following listing's styles to home.component.css.

Listing 5.29 home.component.css

```
.product-grid {
  display: flex;
   flex-wrap: wrap;
  margin: 0 -8px;
}

nga-product-item {
  margin: 0 8px 16px;
  flex-basis: calc(100% / 3 - 16px);
 }
```

Uses CSS flexbox

Gives one third of the screen width plus a margin to each component

Now the landing page of ngAuction has a more modern look, as shown earlier in figure 5.10. Not only does it look better than the version of ngAuction from chapter 3, but its controls (the search form, the menu) provide fast and animated response to the user's actions. Try to place the focus in one of the search fields, and you'll see how the field prompt moves to the top. The button search also shows a ripple effect.

You didn't change the look of the product-detail page shown in figure 3.16 in chapter 3. See if you can do that on your own. When all standard UI elements are replaced with AM components, you can remove the dependency on the Bootstrap library from both package.json and .angular-cli.json (or from angular.json, if you use Angular 6).

Summary

- Providers register objects for future injection.
- You can declare a provider that uses not only a class, but a function or a primitive value as well.
- Injectors form a hierarchy, and if Angular can't find the provider for the requested type at the component level, it'll try to find it by traversing parent injectors.
- A lazy-loaded module has its own injector, and providers declared inside lazy-loaded modules aren't available in the root module.
- Angular Material offers a set of modern-looking UI components.

6 Reactive programming in Angular

This chapter covers

- Handling events as observables
- Using observables with Angular `Router` and forms
- Using observables in HTTP requests
- Minimizing network load by discarding unwanted HTTP responses

The goal of the first five chapters was to jump-start your application development with Angular. In those chapters, we discussed how to generate a new project from scratch, covering modules, routing, and dependency injection. In this chapter, we'll show you how Angular supports a *reactive* style of programming, in which your app reacts on changes either initiated by the user or by asynchronous events like data arriving from a router, form, or server. You'll learn which Angular APIs support data push and allow you to subscribe to RxJS-based observable data streams.

NOTE If you're not familiar with RxJS library concepts such as observables, observers, operators, and subscriptions, read appendix D before proceeding with this chapter.

Angular offers ready-to-use observables for implementing various scenarios: handling events, subscribing to the route's parameters, checking the status of a form, handling HTTP requests, and more. You'll see some examples of using Angular observables, but each of the following chapters contains reactive code as well.

You may say that any JavaScript app can use event listeners and provide callbacks to handle events, but we'll show you how to treat events as data streams that *push* values to the observers over time. You'll be writing code that *subscribes* to observable event streams and handles them in the observer objects. You'll be able to apply one or more operators to handle an event as it moves to the observer, which isn't possible with the regular event listeners.

> **NOTE** Source code for this chapter can be found at https://github.com/Farata/angulartypescriptand www.manning.com/books/angular-development-with-typescript-second-edition. You can find the code samples used in this section in the directory named observables. Open this directory in your IDE and run `npm install` to install Angular and its dependencies. We provide instructions on how to run code samples when it's required.

Let's start by discussing how to handle events with and without observables.

6.1 Handling events without observables

Each DOM event is represented by the object containing properties describing the event. Angular applications can handle standard DOM events and can emit custom events as well. A handler function for a UI event can be declared with an optional `$event` parameter . With standard DOM events, you can use any functions or properties of the browser's `Event` object (see "Event" in the Mozilla Developer Network documentation, http://mzl.la/1EAG6iw).

In some cases, you won't be interested in reading the event object's properties, such as when the only button on a page is clicked and this is all that matters. In other cases, you may want to know specific information, like what character was entered in the `<input>` field when the `keyup` event was dispatched. The following code listing shows how to handle a DOM `keyup` event and print the value from the input field that emitted this event.

Listing 6.1 Handling the `keyup` event

```
template:`<input id="stock" (keyup)="onKey($event)">`  <-- Binds to the keyup event
 ...

onKey(event:any) {
  console.log("You have entered " + event.target.value);  <-- Event handler method
 }
```

In this code snippet, you care only about one property of the `Event` object: the `target`. By applying *object destructuring* (see section A.9.1 in appendix A), the `onKey()` handler

can get the reference to the `target` property on the fly by using curly braces with the function argument:

```
onKey({target}) {
  console.log("You have entered " + target.value);
}
```

If your code dispatches a custom event, it can carry application-specific data, and the event object can be strongly typed (not be `any` type). You'll see how to specify the type of a custom event in listing 8.4 in chapter 8.

A traditional JavaScript application treats a dispatched event as a one-time deal; for example, one click results in one function invocation. Angular offers another approach where you can consider any event as an observable stream of data happening over time. For example, if the user enters several characters in the `<input>` field, each of the characters can be treated as an emission of the observable stream.

You can subscribe to observable events and specify the code to be invoked when each new value is emitted and, optionally, the code for error processing and stream completion. Often you'll specify a number of chained RxJS operators and then invoke the `subscribe()` method.

Why do we even need to apply RxJS operators to events coming from the UI? Let's consider an example that uses event binding to handle multiple `keyup` events as the user types a stock symbol to get its price:

```
<input type='text' (keyup) = "getStockPrice($event)">
```

Isn't this technique good enough for handling multiple events dispatched as the user types? Imagine that the preceding code is used to get a price quote for AAPL stock. After the user types the first `A`, the `getStockPrice()` function will make a request to the server, which will return the price of `A`, if there is such a stock. Then the user enters the second `A`, which results in another server request for the `AA` price quote. The process repeats for `AAP` and `AAPL`.

This isn't what you want. To defer the invocation of `getStockPrice()`, you can place it inside the `setTimeout()` function with, say, a 500-millisecond delay to give the user enough time to type several letters:

```
const stock = document.getElementById("stock");

  stock.addEventListener("keyup", function(event) {
    setTimeout(function() {
      getStockPrice(event);
    }, 500);
  }, false);
```

Don't forget to call `clearTimeout()` and start another timer should the user continue typing in the input field.

How about composing several functions that should preprocess the event before invoking `getStockPrice()`? There's no elegant solution to this. What if the user types

slowly, and during the 500-millisecond interval manages only to enter AAP? The first request for AAP goes to the server, and 500 milliseconds later the second request for AAPL is sent. A program can't discard the results of the first HTTP request if the client returns a `Promise` object, and may overload the network with unwanted HTTP responses.

Handling events with RxJS offers you a convenient operator named `debounceTime` that makes the observable emit the next value only if a specified time passes (such as 500 milliseconds) and the data producer (the `<input>` field in our case) doesn't produce new values during this interval. There's no need to clear and re-create the timer. Also, the `switchMap` operator allows easy cancellation of the observable waiting for a pending request (for example, `getStockPrice()`) if the observable emits new values (for example, the user keep typing). What can Angular offer to handle events from an input field with subscribers?

6.2 Turning DOM events into observables

In Angular applications, you can get direct access to any DOM element using a special class, `ElementRef`, and we'll use this feature to illustrate how you can subscribe to events of an arbitrary HTML element. You'll create an app that will subscribe to the `<input>` element where the user inputs the stock symbol to get its price, as discussed in the previous section.

To turn a DOM event into an observable stream, you need to do the following:

1. Get a reference to the DOM object.
2. Create an observable using `Observable.fromEvent()`, providing the reference to the DOM object and the event you want to subscribe to.
3. Subscribe to this observable and handle the events.

In a regular JavaScript app, to get a reference to the DOM element, you use a DOM selector API, `document.querySelector ()`. In Angular, you can use the `@ViewChild()` decorator to get a reference to an element from a component template.

To uniquely identify the template elements, you'll use local template variables that start with the hash symbol. The following code snippet uses the local template variable `#stockSymbol` as an ID of the `<input>` element:

```
<input type="text" #stockSymbol placeholder="Enter stock">
```

If you need to get a reference to the preceding element inside the TypeScript class, you can use the `@ViewChild('stockSymbol')` decorator, and the application in the following listing illustrates how to do that. Note that you import only those RxJS members that you actually use.

Listing 6.2 fromevent/app.component.ts

```
import {AfterViewInit, Component, ElementRef, ViewChild} from '@angular/core';
import {Observable} from "rxjs";
import {debounceTime, map} from 'rxjs/operators';
```

```
@Component({
  selector: "app-root",
  template: `
    <h2>Observable events</h2>
    <input type="text" #stockSymbol placeholder="Enter stock" >
  `
})
export class AppComponent implements AfterViewInit {

  @ViewChild('stockSymbol') myInputField: ElementRef;

  ngAfterViewInit() {

    let keyup$: =
      Observable.fromEvent(this.myInputField.nativeElement, 'keyup');

     let keyupValue$ = keyup$
    .pipe(
      debounceTime(500),
       map(event => event['target'].value))
      .subscribe(stock => this.getStockQuoteFromServer(stock));
  }

   getStockQuoteFromServer(stock: string) {

     console.log(`The price of ${stock} is
➥${(100 * Math.random()).toFixed(4)}`);
  }
}
```

Declares the property myInputField that holds a reference to the <input> field

Places the code in the ngAfterViewInit() component lifecycle method

Creates an observable from the keyup event

Invokes the getStockQuoteFromServer() method for each value emitted by the observable

Converts the DOM event into the target.value property, which has the stock symbol entered by the user

Prints the generated random stock price

Waits for a 500 ms pause in the observable's emissions

TIP Starting from Angular 6, instead of `Observable.fromEvent()`, just write `fromEvent()`.

NOTE In listing 6.2, the code to subscribe to events is placed in the `ngAfterViewInit ()` component lifecycle method, which Angular invokes when the component's UI is initialized. You'll learn about component lifecycle methods in section 9.2 in chapter 9.

You can see this code sample in action by running the following command:

```
ng serve --app fromevent -o
```

Open the browser's console and start entering the stock symbol. Depending on the speed of your typing, you'll see one or more messages in the console reporting stock price(s).

It's great that you can turn any DOM event into an observable, but directly accessing the DOM by using `ElementRef` is discouraged, because it may present some security vulnerabilities (see https://angular.io/api/core/ElementRef for details). What's a better way to subscribe to value changes in a DOM object?

6.3 Handling observable events with the Forms API

The Angular Forms API (covered in chapters 10 and 11) offers ready-to-use observables that push notifications about important events that are happening with the entire form or with form control. Here are two examples:

- `valueChanges` —This property is an observable that emits data when the value of the form control is changing.
- `statusChanges` —This property is an observable that emits the validity status of the form control or the entire form. The status changes from valid to invalid or vice versa.

In this section, we'll show you how to use the `valueChanges` property with the HTML `<input>` element.

The `FormControl` class, one of the fundamental blocks of forms processing, represents a form control. By default, whenever the value of the form control changes, the underlying `FormControl` object emits an event through its `valueChanges` property of type `Observable`, and you can subscribe to it.

Let's rewrite the app from the previous section by using the Forms API to subscribe to the `input` event of the `<input>` field and generate stock quotes. The form elements can be bound to component properties via the `formControl` directive, and you'll use it instead of accessing the DOM object directly.

The following listing applies the RxJS `debounceTime`operator prior to invoking `subscribe ()`, instructing the `this.searchInput.valueChanges` observable to emit the data if the user isn't typing anything during 500 milliseconds.

Listing 6.3 formcontrol/app.component.ts

```
import {Component} from '@angular/core';
import {FormControl} from '@angular/forms';
import {debounceTime} from 'rxjs/operators;

@Component({
  selector: 'app-root',
  template: `
       <h2>Observable events from formcontrol</h2>
      <input type="text" placeholder="Enter stock"
             [formControl]="searchInput">
    `
})
export class AppComponent {

  searchInput = new FormControl('');

  constructor() {
    this.searchInput.valueChanges
      pipe(debounceTime(500))
      .subscribe(stock => this.getStockQuoteFromServer(stock));
   }

  getStockQuoteFromServer(stock: string) {
```

Links this <input> element to the component property searchInput

The valueChanges property is an observable.

Waits for 500 ms of quiet time before emitting the content of the <input> element

Subscribes to the observable

```
    console.log(`The price of ${stock} is ${(100 * Math.random()).toFixed(4)}
     `);
  }
}
```

Your `subscribe()` method provides the `Observer` with one method (no error or stream-completion handlers). Each value from the stream generated by the `searchInput` control is given to the `getStockQuoteFromServer()` method. In a real-world scenario, this method would issue a request to the server (and you'll see such an app in section 6.4), but your method just generates and prints a random stock price.

If you didn't use the `debounceTime` operator, `valueChanges` would be emitting values after each character typed by the user. Figure 6.1 shows what happens after you start this application and enter `AAPL` in the input field.

Figure 6.1 Getting the price for AAPL

To see this app in action, run the following command in the Terminal:

```
ng serve --app formcontrol -o
```

> **NOTE** You may argue that you could implement this example in listing 6.3 and figure 6.1 by simply binding to the `change` event, which would be dispatched when the user finished entering the stock symbol and moved the focus out of the input field. This is true, but in many scenarios you'll want an immediate response from the server, such as retrieving and filtering a data collection as the user types.

In listing 6.3, you don't make any network requests to the server for price quotes—you generate random numbers on the user's computer. Even if the user enters a wrong stock symbol, this code sample will invoke `Math.random()`, which has a negligible effect on the application's performance. In a real-world application, the user's typos may generate network requests that introduce delays while returning quotes for mistakenly entered stock symbols. How would you go about discarding the results of unwanted requests?

6.4 Discarding results of unwanted HTTP requests with switchMap

One of the advantages of observables over promises is that observables can be cancelled. In the previous section, we offered one scenario in which a typo might result in a server request that returns unwanted results. Implementing master-detail views is another use case for a request cancellation. Say a user clicks a row in a list of products to see the product details that must be retrieved from the server. Then they change their mind and click another row, which issues another server request; in that case, the results of the pending request should ideally be discarded.

In Angular, HTTP requests return observables. Let's look at how to discard the results of pending HTTP requests by creating an application that issues HTTP requests as the user types in the input field. We'll use two observable streams:

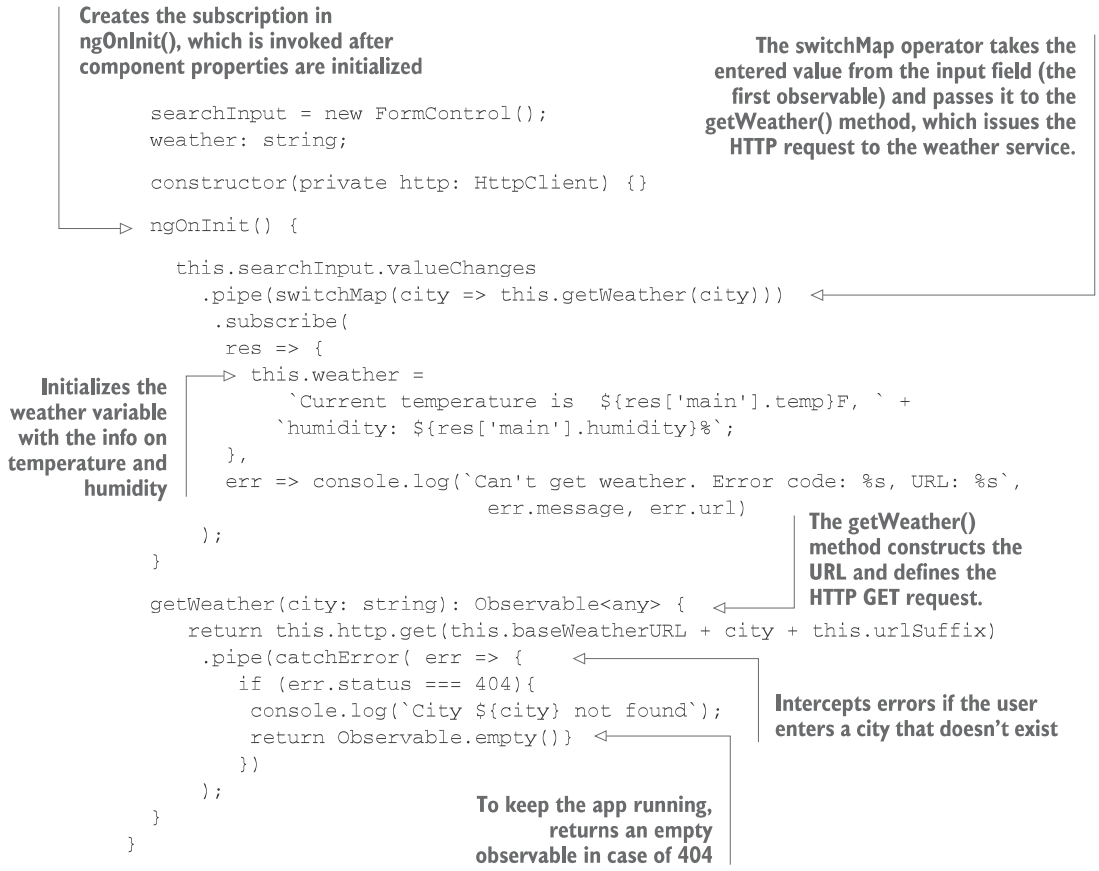
- The observable stream produced by the search `<input>` field
- The observable stream produced by the HTTP requests issued while the user is typing in the search field

For this example, you'll use the free weather service at http://openweathermap.org, which provides an API for making weather requests for cities around the world. To use this service, go to OpenWeatherMap and receive an application ID (`appid`). This service returns the weather information as a JSON-formatted `string`. For example, to get the current temperature in London in Fahrenheit (`units=imperial`), the URL could look like this: http://api.openweathermap.org/data/2.5/find?q=London&units=imperial&appid=12345.

You'll construct the request URL by concatenating the base URL with the entered city name and the application ID. As the user enters the letters of the city name, the code subscribes to the event stream and issues HTTP requests. If a new request is issued before the response from the previous one comes back, the `switchMap` operator (explained in section D.8 in appendix D) cancels and discards the previous inner observable (so the results of the previous HTTP request never reach the browser) and sends the new one to this weather service. This example, shown in the following listing, also uses the `FormControl` directive to generate an observable stream from the input field where the user enters the name of the city.

Listing 6.4 weather/app.component

```
@Component({
  selector: "app-root",
  template: `
    <h2>Observable weather</h2>
    <input type="text" placeholder="Enter city" [formControl]="searchInput">
    <h3>{{weather}}</h3>
  `
})
export class AppComponent implements OnInit{
  private baseWeatherURL = 'http://api.openweathermap.org/data/2.5/weather?q=';
  private urlSuffix: = "&units=imperial&appid=12345";
```



TIP Starting from TypeScript 2.7, you need to initialize the class variables either during declaration or in the constructor: for example, `weather = ''`. If you don't want to do this, set the TypeScript compiler's `strictPropertyInitialization` option to `false`.

Note two observables in listing 6.4:

- The `FormControl` directive creates an observable from the input field events (`this.searchInput.valueChanges`).
- `getWeather()` also returns an observable.

We often use the `switchMap` operator when the data generated by the outer observable (the `FormControl`, in this case) is given to the inner observable (the `getWeather()` function): Observable1 -> `switchMap(function)` -> Observable2 -> `subscribe()`.

If Observable1 pushes the new value, but the inner Observable2 hasn't finished yet, Observable2 gets cancelled. We're switching over from the current inner observable to the new one, and the `switchMap` operator unsubscribes from the pending Observable2 and resubscribes again to handle the new value produced by Observable1.

In listing 6.4, if the observable stream from the UI pushes the next value before the observable returned by `getWeather()` has emitted a value, `switchMap` kills the observable from `getWeather()`, gets the new value for the city from the UI, and invokes `getWeather()` again. Cancelling `getWeather()` results in `HttpClient` discarding the results of the pending HTTP request that was slow and didn't complete in time.

The `subscribe()` method has only a callback for handling data coming from the server, where you extract the temperature and humidity from the returned JSON. If the user makes a request to a nonexistent city, the API offered by this weather service returns 404. You intercept and handle this error in the `catchError` operator. Imagine a slow typer who enters `Lo` while trying to find the weather in *London*. The HTTP request for `Lo` goes out, a 404 is returned, and you create an empty observable so the `subscribe()` method will get an empty result, which is not an error.

To run this app, you need to first obtain your own key (it takes one minute) at http://api.openweathermap.org and replace 12345 in the code in listing 6.4 with your own key. Then you can run this app with the following command:

```
ng serve --app weather -o
```

The browser will open the app at http://localhost:4200, rendering a window with a single input field where you can enter the city name. Figure 6.2 shows the network traffic as you type the word *London* on a computer with a fast 200 Mbps internet connection.

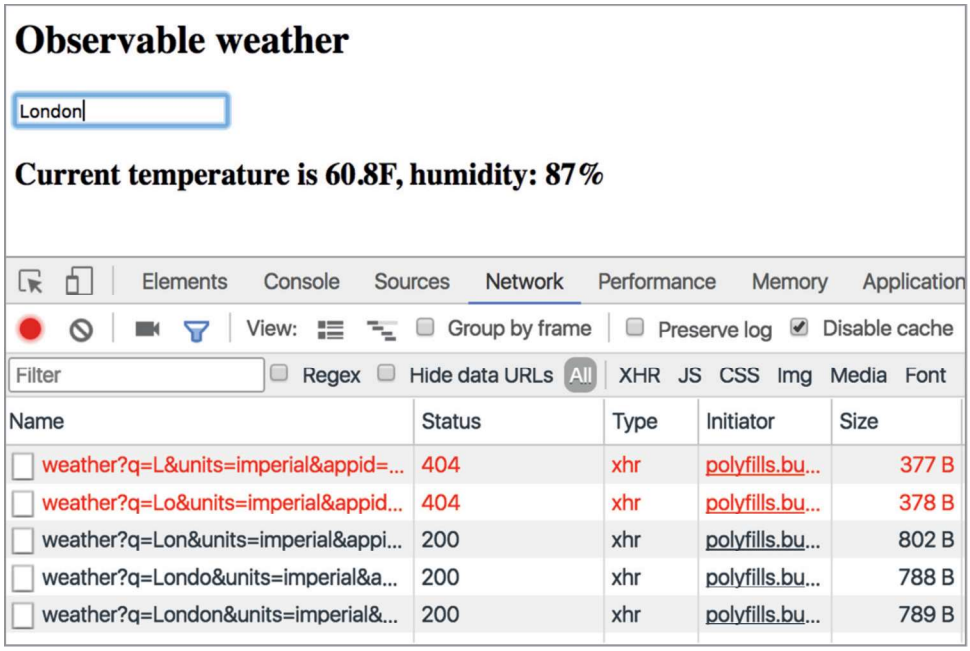


Figure 6.2 Getting weather without throttling

In this case, six HTTP requests were made and returned the HTTP responses. Read the queries in the first two. The requests for the cities `L` and `Lo` came back with 404. But requests for `Lon`, `Lond`, `Londo`, and `London` completed successfully, sending back hundreds of bytes each, unnecessarily congesting the network. Add these bytes up—you'll get 3,134 bytes in total, but users on a fast network wouldn't even notice this.

Now let's emulate a slow network and verify that discarding unwanted results works. On a slow internet connection, each HTTP request takes more than 200 ms to complete, but the user keeps typing, and the responses of the pending HTTP requests should be discarded.

The Network tab of Chrome Developer Tools has a dropdown with selected option Online, which means use the full connection speed. Now, let's emulate a slow connection by selecting the Slow 3G option instead. Retyping the word *London* results in multiple HTTP requests, but the connection is slow now, and the results of pending requests are discarded and never reach the browser, as shown in figure 6.3. Note that this time you get 789 bytes back, which is much better than 3,134.

With very little programming, you save bandwidth by eliminating the need for the browser to handle four HTTP responses for cities you're not interested in and that may not even exist. Just by adding one line with `switchMap`, you accomplish a lot. Indeed, with good frameworks or libraries, you write less code. Angular pipes also allow you to achieve more with less manual coding, and in the next section you'll learn about `AsyncPipe`, which will eliminate the need to make the `subscribe()` call.

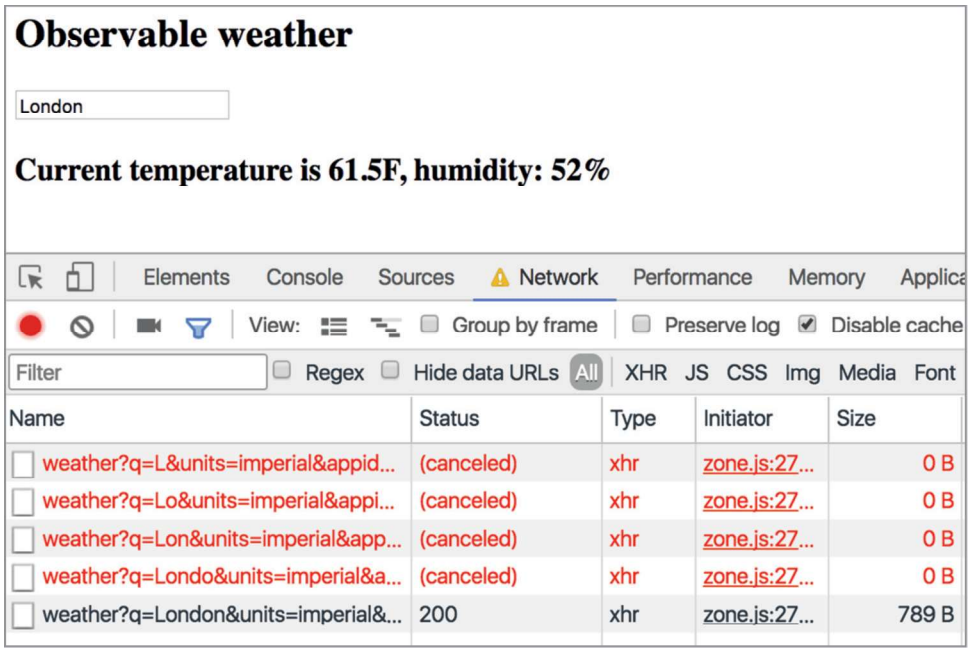


Figure 6.3 Getting weather with throttling

6.5 Using AsyncPipe

Section 2.5 in chapter 2 introduced you to pipes, which are used in a component template and can convert the data right inside the template. For example, a `DatePipe` could convert and display a date in the specified format. A pipe is placed in the template after the vertical bar, for example:

```
<p> Birthday: {{birthday | date: 'medium'}}</p>
```

In this code snippet, `birthday` is a component property of type `Date`. Angular offers an `AsyncPipe` that can take a component property of type `Observable`, autosubscribe to it, and render the result in the template.

The next listing declares a `numbers` variable of type `Observable<number>` and initializes it with an observable that emits a sequential number with an interval of 1 second. The `take(10)` operator will limit the emission to the first 10 numbers.

Listing 6.5 asyncpipe/app.component.ts

```
import {Component} from '@angular/core';
import 'rxjs/add/observable/interval';
import {take} from 'rxjs/operators';
import {Observable} from "rxjs";

@Component({
  selector: "app-root",
  template: `{{numbers$ | async}}`
 })
export class AppComponent {

  numbers$: Observable<number> =
              Observable.interval(1000)
                 .pipe(take(10));
 }
```

Autosubscribes to observable numbers

Emits sequential numbers every second

Takes only 10 numbers from 0 to 9

TIP Starting in Angular 6, instead of `Observable.interval()`, just write `interval()`.

As explained in appendix D, to start getting data from an observable, we need to invoke the `subscribe()` method. In listing 6.5, there's no explicit invocation of `subscribe()`, but note the `async` pipe in the template. The `async` pipe autosubscribes to the `numbers` observable and displays the numbers from 0 to 9 as they're being pushed by the observable. To see this example in action, run the following command:

```
ng serve --app asyncpipe -o
```

This was a pretty simple example that never throws any errors. In real-world applications, things happen, and you should add error handling to the observable with the `catch` operator, as you did in the previous section in the weather example.

Now let's consider one more app that uses the `async` pipe. This time, you'll invoke a function that returns an observable array of products, and you'll use the `async` pipe to render its values. This app will use the `ProductService` injectable, whose

`getProducts()` method returns an observable array of the `Product` objects, as shown in the following listing.

Listing 6.6 asyncpipe-products/product.service.ts

```
import {Injectable} from '@angular/core';
import {Observable} from "rxjs";
import 'rxjs/add/observable/of';

export interface Product {                    <-- Defines the Product type
    id: number;
   title: string;
   price: number
}

@Injectable()
export class ProductService {

  products: Product[] = [                     <-- Populates the products array
     {id: 0, title: "First Product", price: 24.99},
    {id: 1, title:"Second Product", price: 64.99},
    {id: 2, title:"Third Product", price: 74.99}
  ];

  getProducts(): Observable<Product[]> {

    return Observable.of(this.products);      <-- Turns the products array into an observable
   }
}
```

TIP Starting from Angular 6, instead of `Observable.of()` just write `of()`.

The next listing shows an app component that gets the `ProductService` injected and invokes `getProducts()`, which returns an observable. Note that there's no explicit invocation of `subscribe()` there—you use the `async` pipe in the template. In this component, you use Angular's structural directive `*ngFor` to iterate through products and for each product render the `<li>` element with the product title and price, as you can see in the following listing.

Listing 6.7 asyncpipe-products/app.component.ts

```
import {Component} from '@angular/core';
import {Product, ProductService} from "./product.service";
import {Observable} from "rxjs";

@Component({
  selector: "app-root",
  template: `
       <ul>
         <li *ngFor="let product of products$ | async">     <-- Iterates through products and pipes them to async for subscription
            {{product.title}} {{product.price}}            <-- Renders the product title and price
          </li>
       </ul>
    `
})
```

```
export class AppComponent {
  products$: Observable<Product[]>;
  constructor(private productService: ProductService) {}
  ngOnInit() {
    this.products$ = this.productService.getProducts();
   }
}
```

Declares the observable products$ using generics syntax for type checking

Assigns the value to products$

It's important to understand that the `getProducts()` function returns an empty observable that hasn't emitted anything yet, and you assign it to the `products$` variable. No data is pushed to this component until you subscribe to `products$`, and the `async` pipe does it in the template.

To see this application in action, run the following command:

```
ng serve --app asyncpipe-products -o
```

Figure 6.4 shows how the browser will render products.

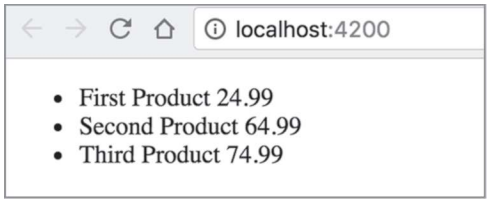


Figure 6.4 Rendering observable products

Since we're talking about pipes, let's apply the Angular built-in `currency` pipe to show the price in US dollars. All it takes is adding the `currency` pipe right after `product.price`:

```
{{product.title}} {{product.price | currency : "USD"}}
```

You can read more about the `currency` pipe and its parameters at https://angular.io/api/common/CurrencyPipe. Figure 6.5 shows how the browser will render products after applying the currency pipe for US dollars.

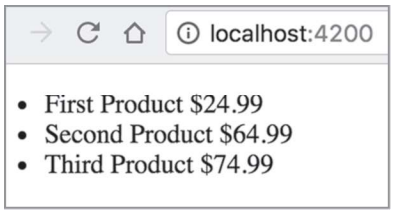


Figure 6.5 Rendering observable products

Using async as

With async pipes , you can use the special syntax `async as` to avoid creating multiple subscriptions in templates. Consider the following code, which creates two subscriptions in a template, assuming that there's an observable `product$`:

```
<div>
  <h4>{{ (product$ | async).price}}</h4>   <-- First subscription
   <p>{{ (product$ | async).description }}</p>  <-- Second subscription
 </div>
```

The following code rewrites the previous by creating a local template variable, `product`, which would store the reference to a single subscription and then reuse it in multiple places in the same template:

```
<div *ngIf="product$ | async as product">      <-- Creates a subscription and stores it in the product variable
   <h4>{{ product.price}}</h4>
   <p>{{ product.description }}</p>
 </div>
```

Uses the subscription called product

Now let's see where observables can be used during navigation with the router.

6.6 Observables and the router

Angular Router offers you observable properties in various classes. Is there an easy way to find them? The fastest way is to open the type definition file (see appendix B) for the class you're interested in. Usually, IDEs offer you an option on the context (right-click) menu to go to the declaration of the selected class. Let's use the `ActivatedRoute` class as an example and take a look at its declaration. It's located in the router_state.d.ts file (we removed some content for brevity), as shown in the following listing.

Listing 6.8 A fragment from `ActivatedRoute`

```
export declare class ActivatedRoute {
    url: Observable<UrlSegment[]>;
    queryParams: Observable<Params>;
    fragment: Observable<string>;
    data: Observable<Data>;
    snapshot: ActivatedRouteSnapshot;
    ...
    readonly paramMap: Observable<ParamMap>;
    readonly queryParamMap: Observable<ParamMap>;
}
```

In section 3.4 in chapter 3, you injected `ActivatedRoute` into the `ProductDetailComponent` so it could receive the route parameters during navigation. Back then, you used the `snapshot` property of `ActivatedRoute` to get the value from the parent route. This technique works fine if you need to get parameters that never change. But

if the parameters in the parent route change over time, you need to subscribe to an observable such as `paramMap`.

Why would the value of the parent's parameter change? Imagine a component that shows a list of products, and when the user selects a product, the app navigates to the route that shows product details. Often, these use cases are called *master-detail communications* .

When the user clicks the product for the first time, the router performs the following steps:

1. Instantiates `ProductDetailComponent`
2. Attaches `ProductDetailComponent` to the DOM object
3. Renders `ProductDetailComponent` in the router outlet
4. Passes the parameter (for example, product ID) to `ProductDetailComponent`

If the user selects another product in the parent's component, the first three steps won't be performed, because `ProductDetailComponent` is already instantiated, attached to the DOM, and rendered by the browser. The router will just pass a newly selected product ID to `ProductDetailComponent`, and that's why subscribing to `paramMap` is the way to go. The following listing implements this scenario, starting from `AppComponent`.

Listing 6.9 master-detail/app.component.ts

```
interface Product {  <-- Defines the Product type
   id: number;
  description: string;
}

@Component({
  selector: 'app-root',
  template: `
        <ul style="width: 100px;">
           <li *ngFor="let product of products"
            [class.selected]="product === selectedProduct"
            (click) = onSelect(product)>          <-- When the user selects an item, invokes the onSelect() handler
              <span>{{product.id}} {{product.description}} </span>
           </li>
       </ul>

        <router-outlet></router-outlet>
    `,
  styles:[`.selected {background-color: cornflowerblue}`]
})
export class AppComponent {

  selectedProduct: Product;

  products: Product[] = [
    {id: 1, description: "iPhone 7"},
    {id: 2, description: "Samsung 7"},
    {id: 3, description: "MS Lumina"}
  ];
```

```
  constructor(private _router: Router) {}

  onSelect(prod: Product): void {
    this.selectedProduct = prod;
    this._router.navigate(["/productDetail", prod.id]);
   }
}
```

Injects the router so you can use its navigate() method

Navigates to the productDetail route

The routes for this app are configured as follows:

```
[
 {path: 'productDetail/:id', component: ProductDetailComponent}
]
```

The `ProductDetailComponent`'s code that subscribes to `paramMap` is shown in the following listing.

Listing 6.10 master-detail/product.detail.component.ts

```
@Component({
  selector: 'product',
  template: `<h3 class="product">Details for product id {{productId}}</h3>`,
   styles: ['.product {background: cyan; width: 200px;} ']
})
export class ProductDetailComponent {

  productId: string;

  constructor(private route: ActivatedRoute) {

    this.route.paramMap
      .subscribe(
        params => this.productId = params.get('id'));
   }
}
```

Embeds the value of productId into the header

Subscribes to the paramMap observable

Extracts the current product id and assigns it to the productId property for displaying in the UI

Now `ProductDetailComponent` will render the text identifying the current product according to user selections. Figure 6.6 shows how the UI looks after the user selects the second product in the list. To see this app in action, run the following command:

```
ng serve --app master-detail -o
```

In chapter 7, you'll rewrite ngAuction, and you'll see how the `ObservableMedia` class from the Flex Layout library will notify you about changes in the screen size (for example, the user reduces the width of the window). This observable is also quite handy in changing the UI layout based on the viewport width of smaller devices like smartphones and tablets.

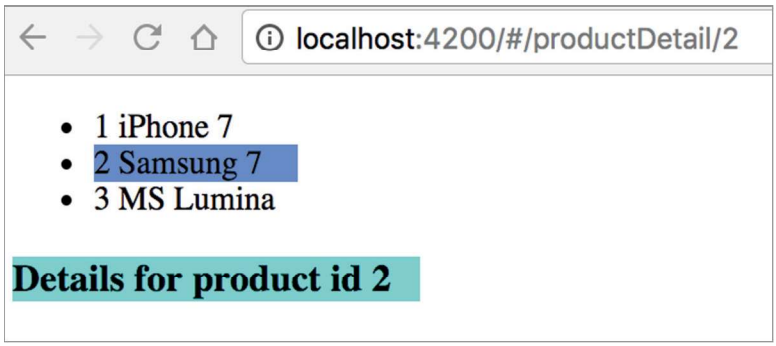


Figure 6.6 Implementing a master-detail scenario

Summary

- Using observable data streams simplifies asynchronous programming. You can subscribe to and unsubscribe from a stream as you need.
- Using the `async` pipe is a preferable way to subscribe to observables.
- The `async` pipe automatically unsubscribes from the observable.
- Using the `switchMap` operator combined with `HttpClient` allows you to easily discard unwanted results of pending HTTP requests.