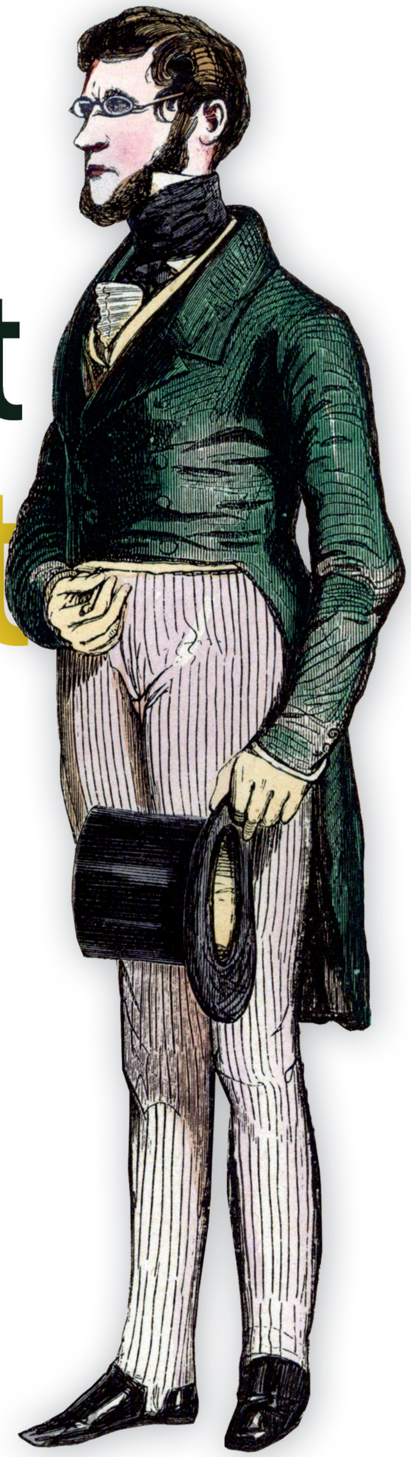
Angular Development with TypeScript

SECOND EDITION

Yakov Fain
Anton Moiseev

SAMPLE CHAPTER

MANNING



Angular Development with TypeScript
by Yakov Fain
Anton Moiseev

Chapter 5

brief contents

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 `Product-Component` 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 `create-Shipment()` 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.

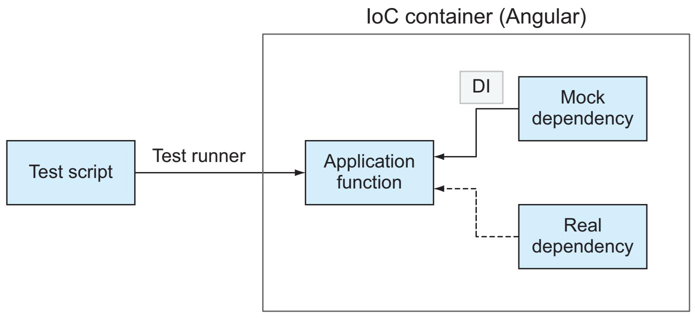


Figure 5.1 DI in testing

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)

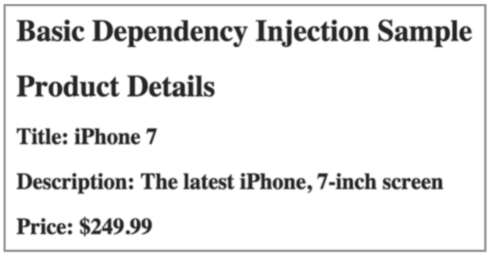
How can you decide which of these properties to use in your code? In the next section, you'll become familiar with the `useClass` property. Section 5.6 illustrates `useFactory` and `useValue`.

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.


Basic Dependency Injection Sample

Product Details

Title: iPhone 7

Description: The latest iPhone, 7-inch screen

Price: $249.99

Figure 5.2 A sample DI application

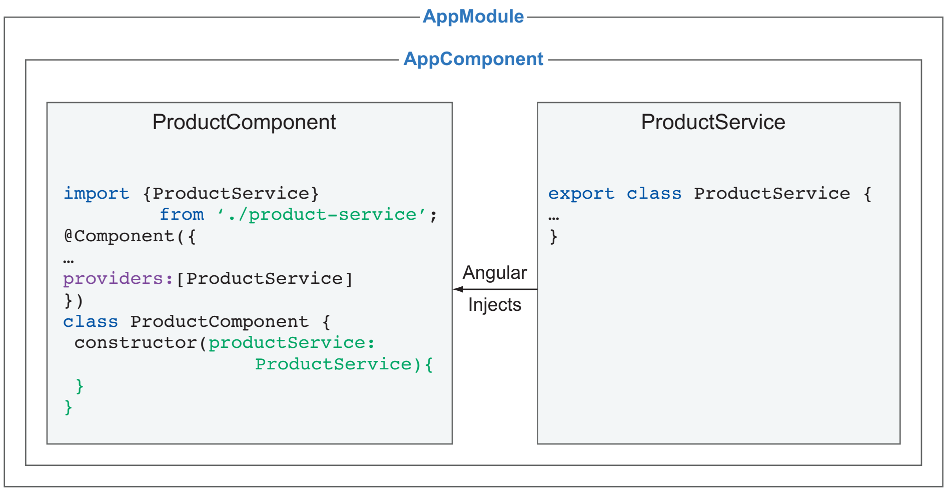
`ProductComponent` can request the injection of the `ProductService` object by declaring the constructor argument with a type:

```
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.



```
import {ProductService}
        from './product-service';
@Component({
…
providers:[ProductService]
})
class ProductComponent {
 constructor(productService:
                ProductService){
 }
}
```

```
export class ProductService {
…
}
```

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`.

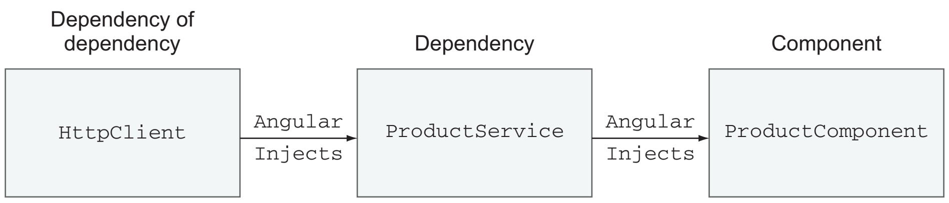


Figure 5.4 A dependency can have its own dependency.

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) {          ◁ Injecting HttpClient
       let products = http.get<string>('products.json')     ◁ Using HTTP GET
                            .subscribe(...);   ◁ 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';   ◁ Imports HttpClientModule in the root module
 ...
@NgModule({
  imports: [
    BrowserModule,
    HttpClientModule      ◁ Adds HttpClientModule to the imports section
   ],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
```

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 {
...
}
```

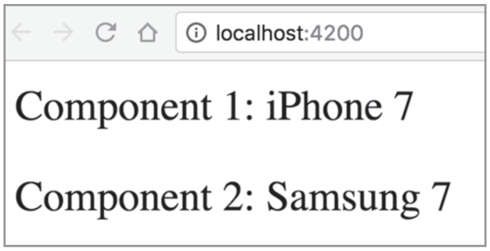
Now that you've seen how to inject an object into a component, let's look at what it takes to replace one implementation of a service with another, using Angular DI.

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`.



Component 1: iPhone 7

Component 2: Samsung 7

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

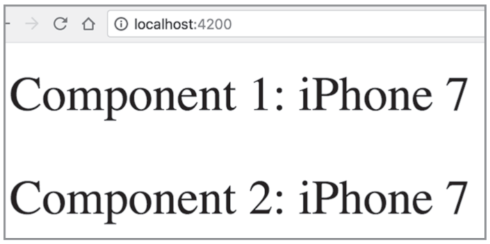
TIP Listing 5.8 has two lines marked as bad design. Read the sidebar "Program to abstractions" for explanations.

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.



Component 1: iPhone 7

Component 2: iPhone 7

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{          <-- Declares an abstract class
   abstract getProduct(): Product;             <-- Declares an abstract method
 }

export class MockProductService extends ProductService{    <-- Creates the first concrete
   getProduct(): Product {                                     implementation of the
    return new Product('Samsung 7');                           abstract class
  }
}

export class RealProductService extends ProductService{    <-- Creates the second
   getProduct(): Product {                                     concrete implementation
    return new Product('iPhone 7');                            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 value from the environment file to the IS_PROD_ENVIRONMENT token

Maps the productServiceFactory factory function to the ProductService 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.

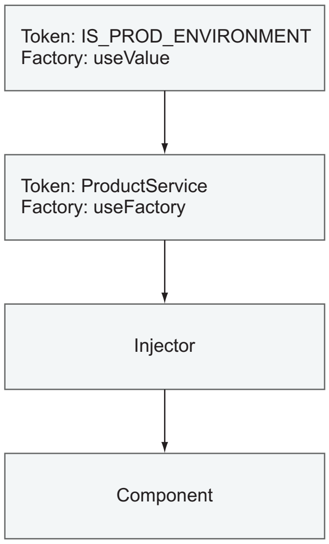


Figure 5.7 Injecting dependencies with dependencies

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

Injects http://myQAserver.com into the BACKEND_URL token

Declares a provider for injecting the value into the 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() {                    <- This service has one method.
     return "I'm the Luxury service from lazy module";
  }
}
```

The `LuxuryModule` declares the provider for this service, as shown in the following listing.

Listing 5.14 luxury.module.ts

```
@NgModule({
    ...
    declarations: [LuxuryComponent],    <- This module has one component.
     providers: [LuxuryService]         <- Declares a provider for LuxuryService
 })

export class LuxuryModule {}            <- 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) {}      ◁ Injects the LuxuryService

  ngOnInit() {
    this.luxuryItem = this.luxuryService.getLuxuryItem();    ◁ 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'} ]    ◁ Specifies the module in quotes for lazy loading
       )
  ],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

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([                    <- Adds a route for a feature module
              {path: 'shipping', component: ShippingComponent}
     ])],
  declarations: [ShippingComponent],
  providers: [ShippingService]
})
export class ShippingModule { }
```

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.

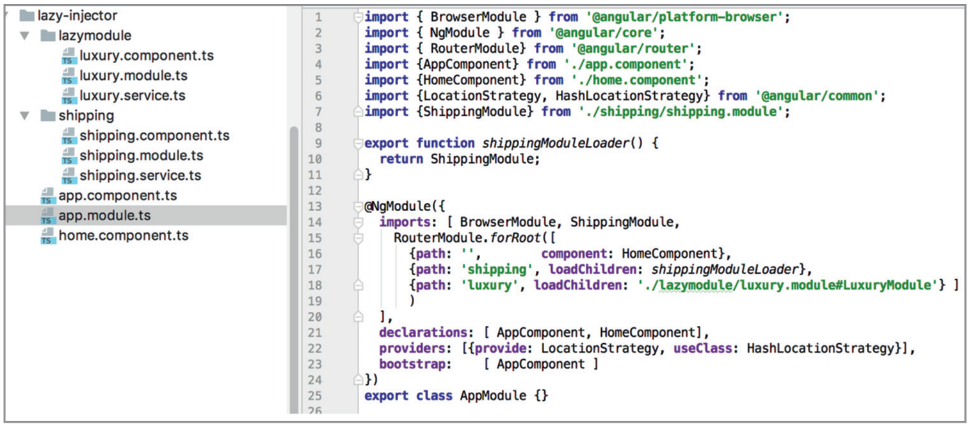
`ShippingComponent` gets `ShippingService` injected and will invoke its `getShippingItem()` method that returns a hardcoded text, "I'm the shipping service from the shipping module."

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) {}     <- Injects ShippingService

  ngOnInit() {
        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`.



```
lazy-injector
  lazymodule
    luxury.component.ts
    luxury.module.ts
    luxury.service.ts
  shipping
    shipping.component.ts
    shipping.module.ts
    shipping.service.ts
  app.component.ts
  app.module.ts
  home.component.ts
```

```
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3  import { RouterModule} from '@angular/router';
4  import {AppComponent} from './app.component';
5  import {HomeComponent} from './home.component';
6  import {LocationStrategy, HashLocationStrategy} from '@angular/common';
7  import {ShippingModule} from './shipping/shipping.module';
8
9  export function shippingModuleLoader() {
10   return ShippingModule;
11 }
12
13 @NgModule({
14   imports: [ BrowserModule, ShippingModule,
15     RouterModule.forRoot([
16       {path: '',        component: HomeComponent},
17       {path: 'shipping', loadChildren: shippingModuleLoader},
18       {path: 'luxury', loadChildren: './lazymodule/luxury.module#LuxuryModule'} ]
19       )
20   ],
21   declarations: [ AppComponent, HomeComponent],
22   providers: [{provide: LocationStrategy, useClass: HashLocationStrategy}],
23   bootstrap:    [ AppComponent ]
24 })
25 export class AppModule {}
26
```

Figure 5.8 An app module that uses feature modules

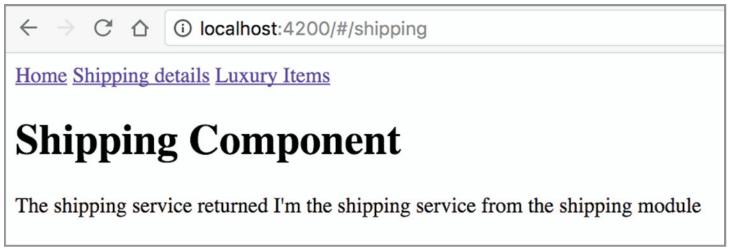
NOTE By the time you read this, the function in lines 9–11 may not be needed, and line 17 for eager loading the `ShippingModule` could look like this: `{path: 'shipping', loadChildren: () => ShippingModule}`. But at the time of writing, using a function in line 17 results in errors during the AOT compilation.

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.


localhost:4200/#/shipping

Home Shipping details Luxury Items

Shipping Component

The shipping service returned I'm the shipping service from the shipping module

Figure 5.9 Navigating to the shipping module

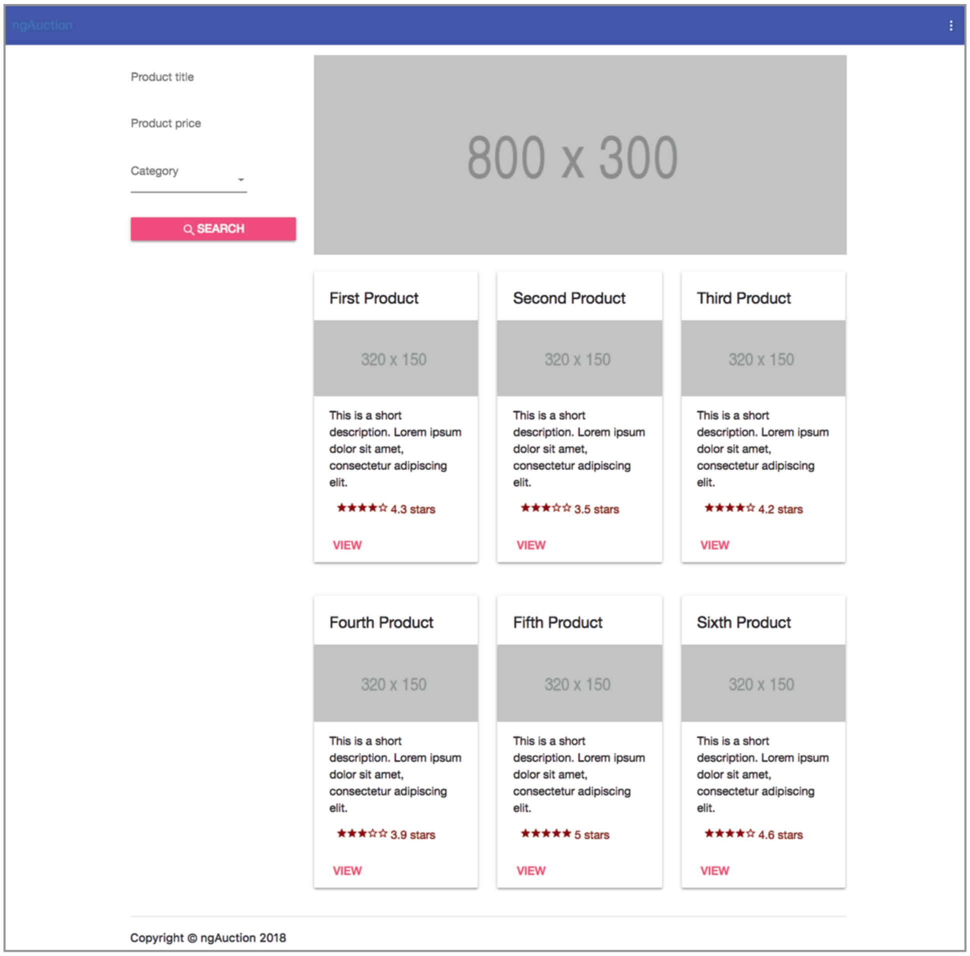
5.9 Hands-on: Using Angular Material components in ngAuction

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.



This is a short description. Lorem ipsum dolor sit amet, consectetur adipiscing elit.

4.3 stars · 3.5 stars · 4.2 stars · 3.9 stars · 5 stars · 4.6 stars

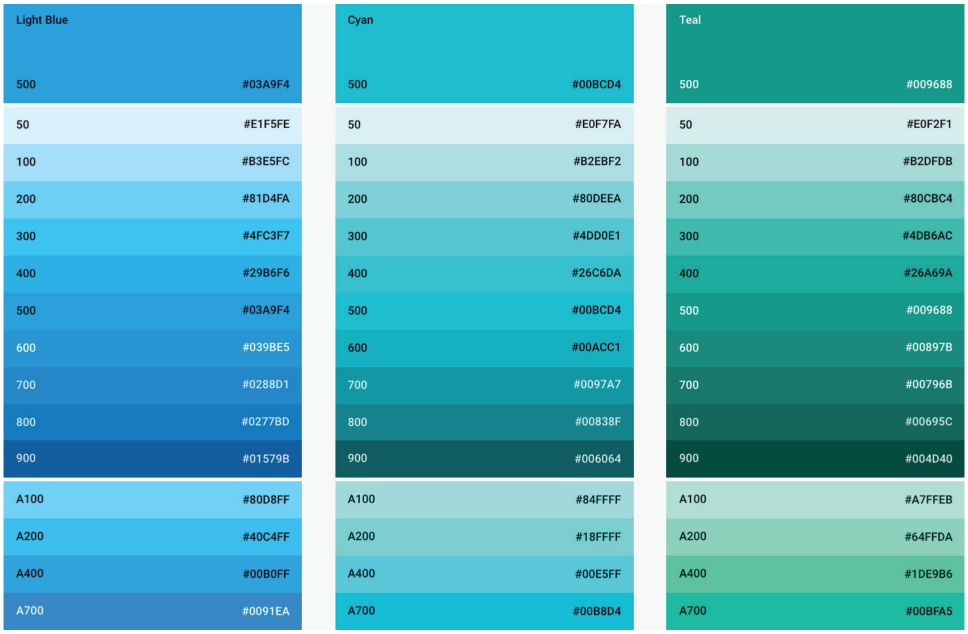
Copyright © ngAuction 2018

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.



Light Blue		Cyan		Teal	
500	#03A9F4	500	#00BCD4	500	#009688
50	#E1F5FE	50	#E0F7FA	50	#E0F2F1
100	#B3E5FC	100	#B2EBF2	100	#B2DFDB
200	#81D4FA	200	#80DEEA	200	#80CBC4
300	#4FC3F7	300	#4DD0E1	300	#4DB6AC
400	#29B6F6	400	#26C6DA	400	#26A69A
500	#03A9F4	500	#00BCD4	500	#009688
600	#039BE5	600	#00ACC1	600	#00897B
700	#0288D1	700	#0097A7	700	#00796B
800	#0277BD	800	#00838F	800	#00695C
900	#01579B	900	#006064	900	#004D40
A100	#80D8FF	A100	#84FFFF	A100	#A7FFEB
A200	#40C4FF	A200	#18FFFF	A200	#64FFDA
A400	#00B0FF	A400	#00E5FF	A400	#1DE9B6
A700	#0091EA	A700	#00B8D4	A700	#00BFA5

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">                  ◁ AM toolbar of primary theme color
   <a [routerLink]="['/']">Home</a>

  <button mat-icon-button>                     ◁ A button with an icon
     <mat-icon>more_vert</mat-icon>            ◁ Places the Google Material icon more_vert on the button
    </button>
</mat-toolbar>
```

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>      ◁ Adds a link to navigate to a default route

<button mat-icon-button >      ◁ 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.

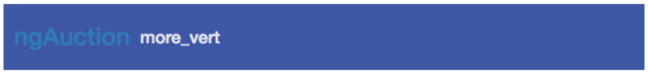


Figure 5.12 A toolbar with a broken icon

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.

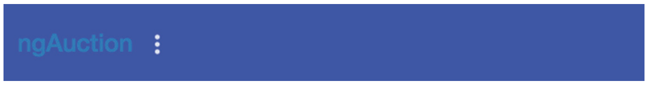


Figure 5.13 A toolbar with a fixed icon

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.

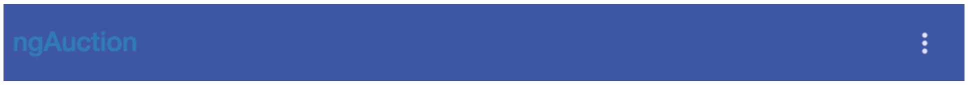


Figure 5.14 Pushing the button to the right

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.

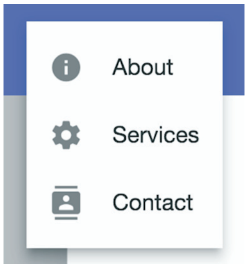


Figure 5.15 The toolbar menu

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>
                                                      Third form field
  <mat-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
     <mat-icon>search</mat-icon>SEARCH                       Submit button
  </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.

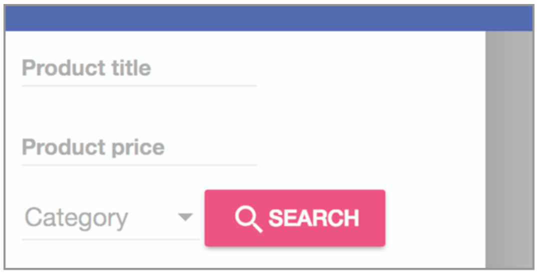


Figure 5.16 The search form with a misaligned button

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.

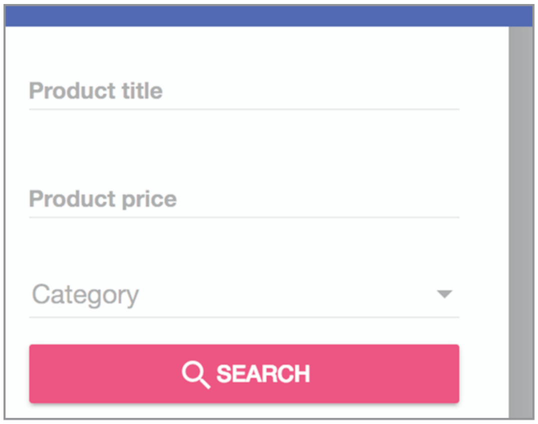


Figure 5.17 The search form

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

```
<mat-card>                                          ◁ Defines the content of the AM <mat-card> component
   <mat-card-title>{{ product.title }}</mat-card-title>   ◁ The product title goes on top.
   <img mat-card-image src="http://placehold.it/320x150">   ◁ Product image
   <mat-card-content>
    {{product.description}}   ◁ Product description
   </mat-card-content>
  <mat-card-actions>
    <a mat-button color="accent"
```

```
        [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.

JAVASCRIPT

Angular Development with TypeScript Second Edition

Fain • Moiseev

Free eBook
See first page

Whether you're building lightweight web clients or full-featured SPAs, Angular is a clear choice. The Angular framework is fast, efficient, and widely adopted. Add the benefits of developing in the statically typed, fully integrated TypeScript language, and you get a programming experience other JavaScript frameworks just can't match.

Angular Development with TypeScript, Second Edition teaches you how to build web applications with Angular and TypeScript. Written in an accessible, lively style, this illuminating guide covers core concerns like state management, data, forms, and server communication as you build a full-featured online auction app. You'll get the skills you need to write type-aware classes, interfaces, and generics with TypeScript, and discover time-saving best practices to use in your own work.

What's Inside

- Code samples for Angular 5, 6, and 7
- Dependency injection
- Reactive programming
- The Angular Forms API

Written for intermediate web developers familiar with HTML, CSS, and JavaScript.

Yakov Fain and **Anton Moiseev** are experienced trainers and web application developers. They have coauthored several books on software development.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/angular-development-with-typescript-second-edition

"Informative, accurate, and insightful."
—Kunal Jaggi, General Motors

"This is the book that you should read, no matter where you are along your Angular learning journey."
—Rahul Rai, Telstra

"A brilliant revisit of a masterwork—not only about Angular but also about superb use of TypeScript. Inspiring and insightful."
—Alain Couniot, STIB-MIVB

"A rock-solid guide to the most important web framework today—authoritative, accessible, and complete."
—Dennis Sellinger, Géotech

ISBN-13: 978-1-61729-534-8
ISBN-10: 1-61729-534-5
54999
9 781617 295348

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]