

CHAPTER 15



Understanding Components

Components are directives that have their own templates, rather than relying on content provided from elsewhere. Components have access to all the directive features described in earlier chapters and still have a host element, can still define input and output properties, and so on. But they also define their own content using templates.

It can be easy to underestimate the importance of the template, but attribute and structural directives have limitations. Directives can do useful and powerful work, but they don't have much insight into the elements they are applied to. Directives are most useful when they are general-purpose tools, such as the `ngModel` directive, which can be applied to any data model property and any form element, without regard to what the data or the element is being used for.

Components, by contrast, are closely tied to the contents of their templates. Components provide the data and logic that will be used by the data bindings that are applied to the HTML elements in the template, which provide the context used to evaluate data binding expressions and act as the glue between the directives and the rest of the application. Components are also a useful tool in allowing large Angular projects to be broken up into manageable chunks.

In this chapter, I explain how components work and explain how to restructure an application by introducing some additional components. Table 15-1 puts components in context.

Table 15-1. Putting Components in Context

Question	Answer
What are they?	Components are directives that define their own HTML content and, optionally, CSS styles.
Why are they useful?	Components make it possible to define self-contained blocks of functionality, which makes projects more manageable and allows for functionality to be more readily reused.
How are they used?	The <code>@Component</code> decorator is applied to a class, which is registered in the application's Angular module.
Are there any pitfalls or limitations?	No. Components provide all the functionality of directives, with the addition of providing their own templates.
Are there any alternatives?	An Angular application must contain at least one component, which is used in the bootstrap process. Aside from this, you don't have to add additional components, although the resulting application becomes unwieldy and difficult to manage.

Table 15-2 summarizes the chapter.

Table 15-2. Chapter Summary

Problem	Solution	Listing
Creating a component	Apply the @Component directive to a class	1-5
Defining the content displayed by a component	Create an inline or external template	6-8
Including data in a template	Use a data binding in the component's template	9
Coordinating between components	Use input or output properties	10-16
Displaying content in an element to which a component has been applied	Project the host element's content	17-21
Styling component content	Create component styles	22-31
Querying the content in the component's template	Use the @ViewChildren decorator	32

Preparing the Example Project

In this chapter, I continue using the example project that I created in Chapter 9 and have been modifying since. No changes are required to prepare for this chapter.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Run the following command in the example folder to start the Angular development tools:

```
ng serve
```

Open a new browser and navigate to <http://localhost:4200> to see the content in Figure 15-1.

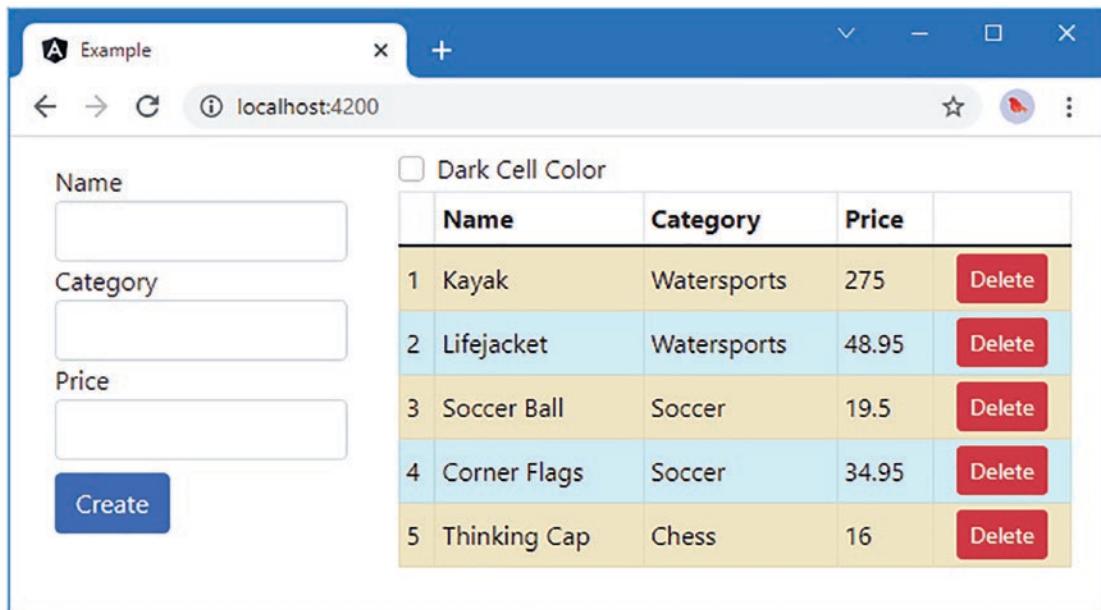


Figure 15-1. Running the example project

Structuring an Application with Components

At the moment, the example project contains only one component and one template. Angular applications require at least one component, known as the *root component*, which is the entry point specified in the Angular module.

The problem with having only one component is that it ends up containing the logic required for all the application's features, with its template containing all the markup required to expose those features to the user. The result is that a single component and its template are responsible for handling a lot of tasks. The component in the example application is responsible for the following:

- Providing Angular with an entry point into the application, as the root component
- Providing access to the application's data model so that it can be used in data bindings
- Defining the HTML form used to create new products
- Defining the HTML table used to display products
- Defining the layout that contains the form and the table
- Maintaining state information used to prevent invalid data from being used to create data
- Maintaining state information about whether the table should be displayed

There is a lot going on for such a simple application, and not all of these tasks are related. This effect tends to creep up gradually as development proceeds, but it means that the application is harder to test because individual features can't be isolated effectively, and it is harder to enhance and maintain because the code and markup become increasingly complex.

Adding components to the application allows features to be separated into building blocks that can be used repeatedly in different parts of the application and tested in isolation. In the sections that follow, I create components that break up the functionality contained in the example application into manageable, reusable, and self-contained units. Along the way, I'll explain the different features that components provide beyond those available to directives. To prepare for these changes, I have simplified the existing component's template, as shown in Listing 15-1.

Listing 15-1. Simplifying the Content of the template.html File in the src/app Folder

```
<div class="container-fluid">
  <div class="row p-2">
    <div class="col-4 p-2 bg-success text-white">
      Form will go here
    </div>
    <div class="col p-2 bg-primary text-white">
      Table will go here
    </div>
  </div>
</div>
```

When you save the changes to the template, you will see the content in Figure 15-2. The placeholders will be replaced with application functionality as I develop the new components and add them to the application.

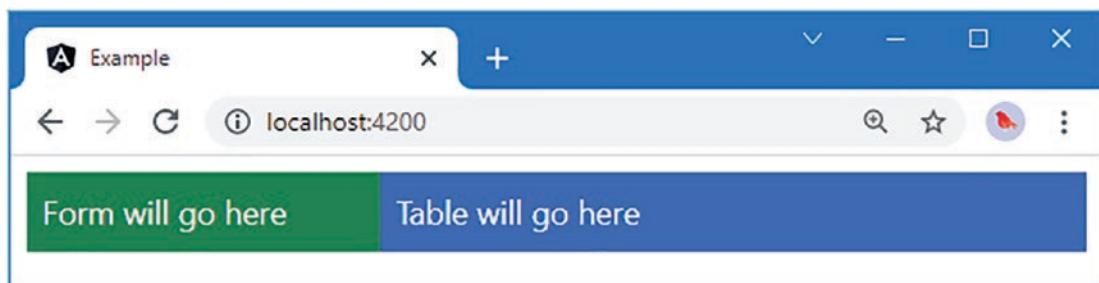


Figure 15-2. Simplifying the existing template

Creating New Components

To create a new component, I added a file called `productTable.component.ts` to the `src/app` folder and used it to define the component shown in Listing 15-2.

Listing 15-2. The Contents of the productTable.component.ts File in the src/app Folder

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

@Component({
  selector: "paProductTable",
  template: "<div>This is the table component</div>"
})
export class ProductTableComponent {

}
```

A component is a class to which the `@Component` decorator has been applied. This is as simple as a component can get, and it provides just enough functionality to count as a component without doing anything useful.

The naming convention for the files that define components is to use a descriptive name that suggests the purpose of the component, followed by a period and then `component.ts`. For this component, which will be used to generate the table of products, the filename is `productTable.component.ts`. The name of the class should be equally descriptive. This component's class is named `ProductTableComponent`.

The `@Component` decorator describes and configures the component. The most useful decorator properties are described in Table 15-3, which also includes details of where they are described (not all of them are covered in this chapter).

Table 15-3. Useful Component Decorator Properties

Name	Description
animations	This property is used to configure animations, as described in Chapter 27.
encapsulation	This property is used to change the view encapsulation settings, which control how component styles are isolated from the rest of the HTML document. See the “Setting View Encapsulation” section for details.
selector	This property is used to specify the CSS selector used to match host elements, as described after the table.
styles	This property is used to define CSS styles that are applied only to the component’s template. The styles are defined inline, as part of the TypeScript file. See the “Using Component Styles” section for details.
styleUrls	This property is used to define CSS styles that are applied only to the component’s template. The styles are defined in separate CSS files. See the “Using Component Styles” section for details.
template	This property is used to specify an inline template, as described in the “Defining Templates” section.
templateUrl	This property is used to specify an external template, as described in the “Defining Templates” section.
providers	This property is used to create local providers for services, as described in Chapter 17.
viewProviders	This property is used to create local providers for services that are available only to view children, as described in Chapter 18.

For the second component, I created a file called `productForm.component.ts` in the `src/app` folder and added the code shown in Listing 15-3.

Listing 15-3. The Contents of the `productForm.component.ts` File in the `src/app` Folder

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

@Component({
  selector: "paProductForm",
  template: "<div>This is the form component</div>"
})
export class ProductFormComponent {

}
```

This component is equally simple and is just a placeholder for the moment. Later in the chapter, I'll add some more useful features. To enable the components, they must be declared in the application's Angular module, as shown in Listing 15-4.

Listing 15-4. Enabling New Components in the `app.module.ts` File in the `src/app` Folder

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoWay.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

The component class is brought into scope using an `import` statement and is added to the `NgModule` decorator's `declarations` array. The final step is to add an HTML element that matches the component's `selector` property, as shown in Listing 15-5, which will provide the component with its host element.

Listing 15-5. Adding a Host Element in the template.html File in the src/app Folder

```
<div class="container-fluid">
  <div class="row p-2">
    <div class="col-4 p-2 bg-success text-white">
      <paProductForm></paProductForm>
    </div>
    <div class="col p-2 bg-primary text-white">
      <paProductTable></paProductTable>
    </div>
  </div>
</div>
```

When all the changes have been saved, the browser will display the content shown in Figure 15-3, which shows that parts of the HTML document are now under the management of the new components.

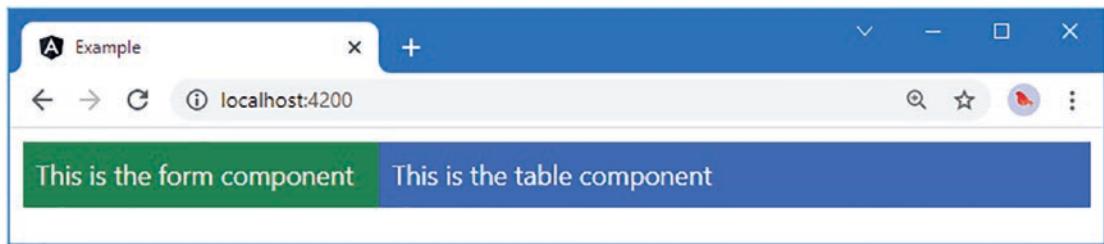


Figure 15-3. Adding new components

Understanding the New Application Structure

The new components have changed the structure of the application. Previously, the root component was responsible for all the HTML content displayed by the application. Now, however, there are three components, and responsibility for some of the HTML content has been delegated to the new additions, as illustrated in Figure 15-4.

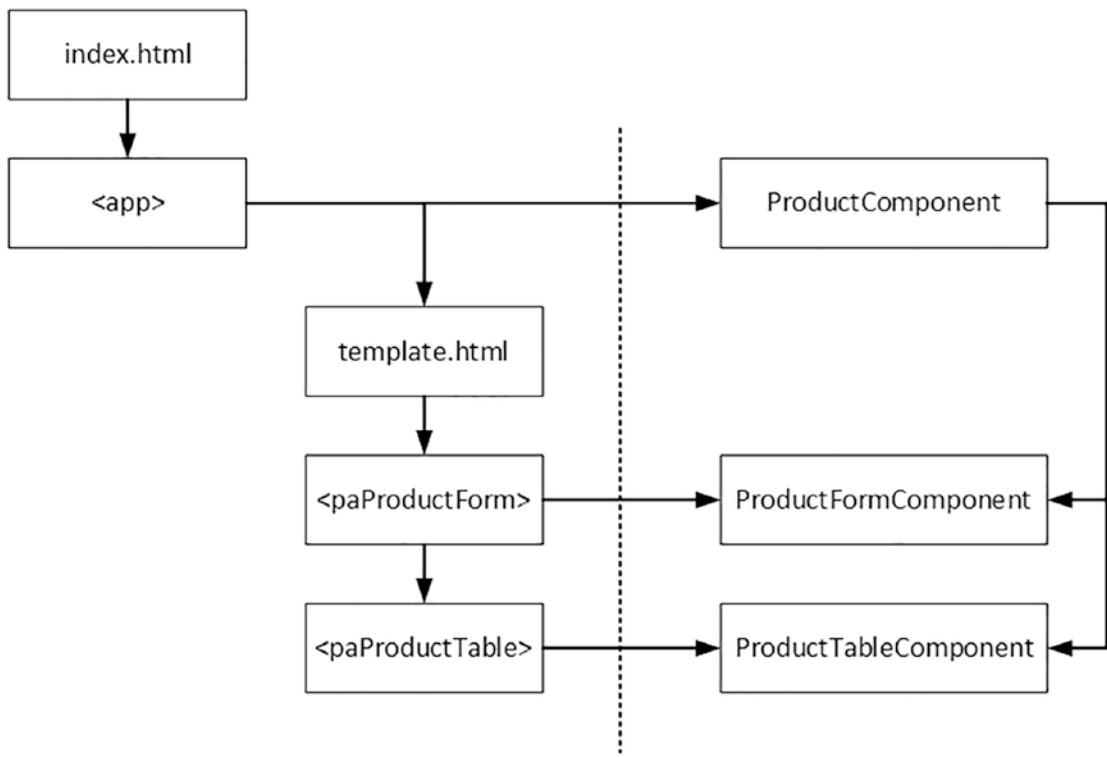


Figure 15-4. The new application structure

When the browser loads the `index.html` file, the Angular bootstrap process starts, and Angular processes the application's module, which provides a list of the components that the application requires. Angular inspects the decorator for each component in its configuration, including the value of the `selector` property, which is used to identify which elements will be hosts.

Angular then begins processing the body of the `index.html` file and finds the `app` element, which is specified by the `selector` property of the `ProductComponent` component. Angular populates the `app` element with the component's template, which is contained in the `template.html` file. Angular inspects the contents of the `template.html` file and finds the `paProductForm` and `paProductTable` elements, which match the `selector` properties of the newly added components. Angular populates these elements with each component's template, producing the placeholder content shown in Figure 15-3.

There are some important new relationships to understand. First, the HTML content that is displayed in the browser window is now composed of several templates, each of which is managed by a component. Second, the `ProductComponent` is now the parent component to the `ProductFormComponent` and `ProductTableComponent` objects, a relationship that is formed by the fact that the host elements for the new components are defined in the `template.html` file, which is the `ProductComponent` template. Equally, the new components are children of the `ProductComponent`. The parent-child relationship is an important one when it comes to Angular components, as you will see as I describe how components work in later sections.

Defining Templates

Although there are new components in the application, they don't have much impact at the moment because they display only placeholder content. Each component has its own template, which defines the content that will be used to replace its host element in the HTML document. There are two different ways to define templates: inline within the `@Component` decorator or externally in an HTML file.

The new components that I added use templates, where a fragment of HTML is assigned to the `template` property of the `@Component` decorator, like this:

```
...
template: "<div>This is the form component</div>"
...
```

The advantage of this approach is simplicity: the component and the template are defined in a single file, and there is no way that the relationship between them can be confused. The drawback of inline templates is that they can get out of control and be hard to read if they contain more than a few HTML elements.

Note Another problem is that editors that highlight syntax errors as you type usually rely on the file extension to figure out what type of checking should be performed and won't realize that the value of the `template` property is HTML and will simply treat it as a string.

If you are using TypeScript, then you can use multiline strings to make inline templates more readable. Multiline strings are denoted with the backtick character (the ``` character, which is also known as the *grave accent*), and they allow strings to spread over multiple lines, as shown in Listing 15-6.

Listing 15-6. Using a Multiline String in the `productTable.component.ts` File in the `src/app` Folder

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

@Component({
  selector: "paProductTable",
  template: `<div class='bg-info p-2'>
    This is a multiline template
  </div>`
})
export class ProductTableComponent { }
```

Multiline strings allow the structure of the HTML elements in a template to be preserved, which makes it easier to read and increase the size of the template that can be practically included inline before it becomes too unwieldy to manage. Figure 15-5 shows the effect of the template in Listing 15-6.

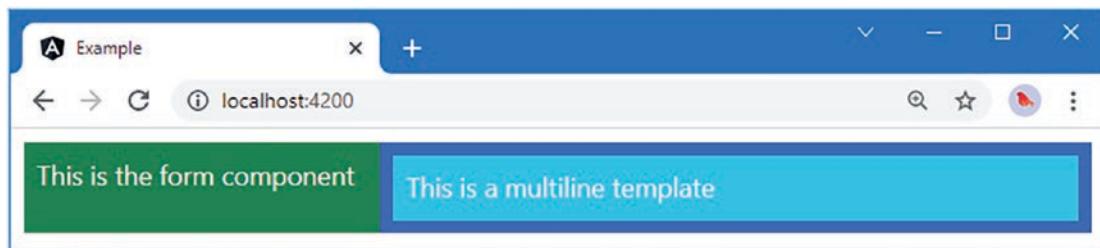


Figure 15-5. Using a multiline inline template

My advice is to use external templates (explained in the next section) for any template that contains more than two or three simple elements, largely to take advantage of the HTML editing and syntax highlighting features provided by modern editors, which can go a long way to reduce the number of errors you discover when running the application.

Defining External Templates

External templates are defined in a different file from the rest of the component. The advantage of this approach is that the code and HTML are not mixed together, which makes both easier to read and unit test, and it also means that code editors will know they are working with HTML content when you are working on a template file, which can help reduce coding-time errors by highlighting errors.

The drawback of external templates is that you have to manage more files in the project and ensure that each component is associated with the correct template file. The best way to do this is to follow a consistent filenames strategy so that it is immediately obvious that a file contains a template for a given component. The convention for Angular is to create pairs of files using the convention <componentname>.component.<type> so that when you see a file called productTable.component.ts, you know it contains a component called Products written in TypeScript, and when you see a file called productTable.component.html, you know that it contains an external template for the Products component.

Tip The syntax and features for both types of template are the same, and the only difference is where the content is stored, either in the same file as the component code or in a separate file.

To define an external template using the naming convention, I created a file called productTable.component.html in the src/app folder and added the markup shown in Listing 15-7.

Listing 15-7. The Contents of the productTable.component.html File in the src/app Folder

```
<div class="bg-info p-2">
  This is an external template
</div>
```

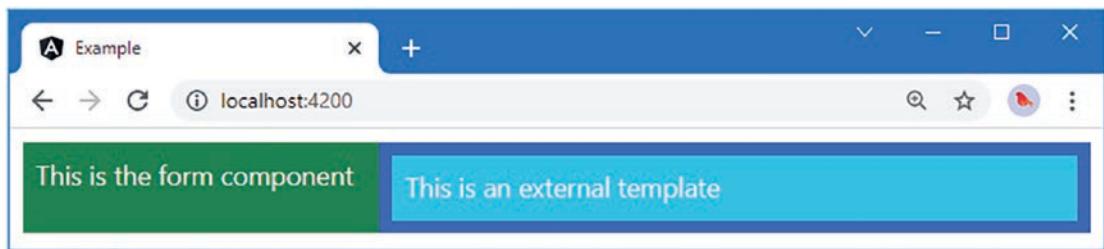
This is the kind of template that I have been using for the root component since Chapter 9. To specify an external template, the templateUrl property is used in the @Component decorator, as shown in Listing 15-8.

Listing 15-8. Using an External Template in the productTable.component.ts File in the src/app Folder

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

@Component({
  selector: "paProductTable",
  // template: `<div class='bg-info p-2'>
  //           This is a multiline template
  //         </div>`
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {
```

Notice that different properties are used: `template` is for inline templates, and `templateUrl` is for external templates. Figure 15-6 shows the effect of using an external template.

**Figure 15-6.** Using an external template

Using Data Bindings in Component Templates

A component's template can contain the full range of data bindings and target any of the built-in directives or custom directives that have been registered in the application's Angular module. Each component class provides the context for evaluating the data binding expressions in its template, and by default, each component is isolated from the others. This means the component doesn't have to worry about using the same property and method names that other components use and can rely on Angular to keep everything separate. As an example, Listing 15-9 shows the addition of a property called `model` to the form child component, which would conflict with the property of the same name in the root component were they not kept separate.

Listing 15-9. Adding a Property in the productForm.component.ts File in the src/app Folder

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

@Component({
  selector: "paProductForm",
  template: "<div>{{model}}</div>"
})
export class ProductFormComponent {

  model: string = "This is the model";
}
```

The component class uses the `model` property to store a message that is displayed in the template using a string interpolation binding. Figure 15-7 shows the result.

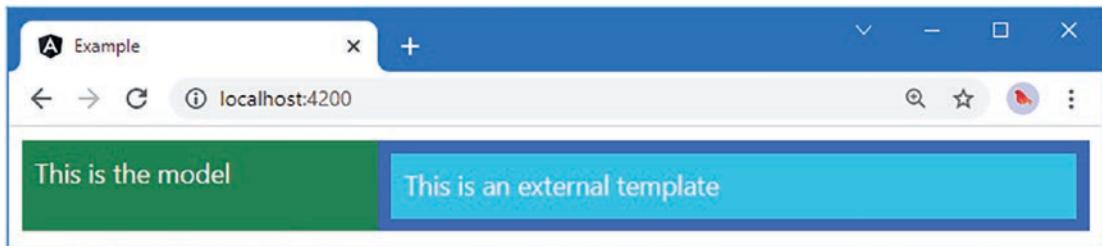


Figure 15-7. Using a data binding in a child component

Using Input Properties to Coordinate Between Components

Few components exist in isolation and need to share data with other parts of the application. Components can define input properties to receive the value of data binding expressions on their host elements. The expression will be evaluated in the context of the parent component, but the result will be passed to the child component's property.

To demonstrate, Listing 15-10 adds an input property to the table component, which it will use to receive the model data that it should display.

Listing 15-10. Defining an Input Property in the `productTable.component.ts` File in the `src/app` Folder

```
import { Component, Input } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {

  @Input("model")
  dataModel: Model | undefined;

  getProduct(key: number): Product | undefined {
    return this.dataModel?.getProduct(key);
  }

  getProducts(): Product[] | undefined {
    return this.dataModel?.getProducts();
  }
}
```

```

deleteProduct(key: number) {
    this.dataModel?.deleteProduct(key);
}

showTable: boolean = true;
}

```

The component now defines an input property that will be assigned the value expression assigned to the `model` attribute on the host element. The `getProduct`, `getProducts`, and `deleteProduct` methods use the `input` property to provide access to the data model to bindings in the component's template, which is modified in Listing 15-11. The `showTable` property is used when I enhance the template in Listing 15-14 later in the chapter.

Listing 15-11. Adding a Data Binding in the `productTable.component.html` File in the `src/app` Folder

There are {{getProducts()?.length}} items in the model

Providing the child component with the data that it requires means adding a binding to its host element, which is defined in the template of the parent component, as shown in Listing 15-12.

Listing 15-12. Adding a Data Binding in the `template.html` File in the `src/app` Folder

```

<div class="container-fluid">
    <div class="row p-2">
        <div class="col-4 p-2 bg-success text-white">
            <paProductForm></paProductForm>
        </div>
        <div class="col p-2 bg-primary text-white">
            <paProductTable [model]="model"></paProductTable>
        </div>
    </div>
</div>

```

The effect of this binding is to provide the child component with access to the parent component's `model` property. This can be a confusing feature because it relies on the fact that the host element is defined in the parent component's template but that the input property is defined by the child component, as illustrated by Figure 15-8.

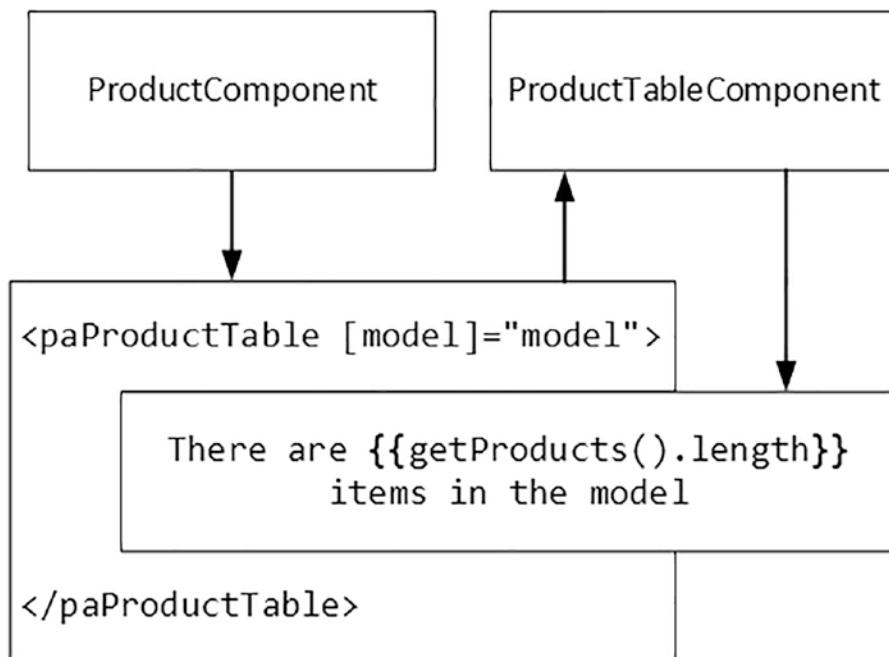


Figure 15-8. Sharing data between parent and child components

The child component's host element acts as the bridge between the parent and child components, and the input property allows the component to provide the child with the data it needs, producing the result shown in Figure 15-9.

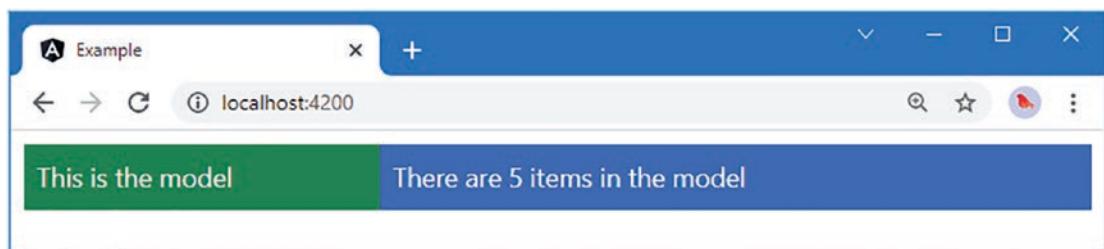


Figure 15-9. Sharing data from a parent to a child component

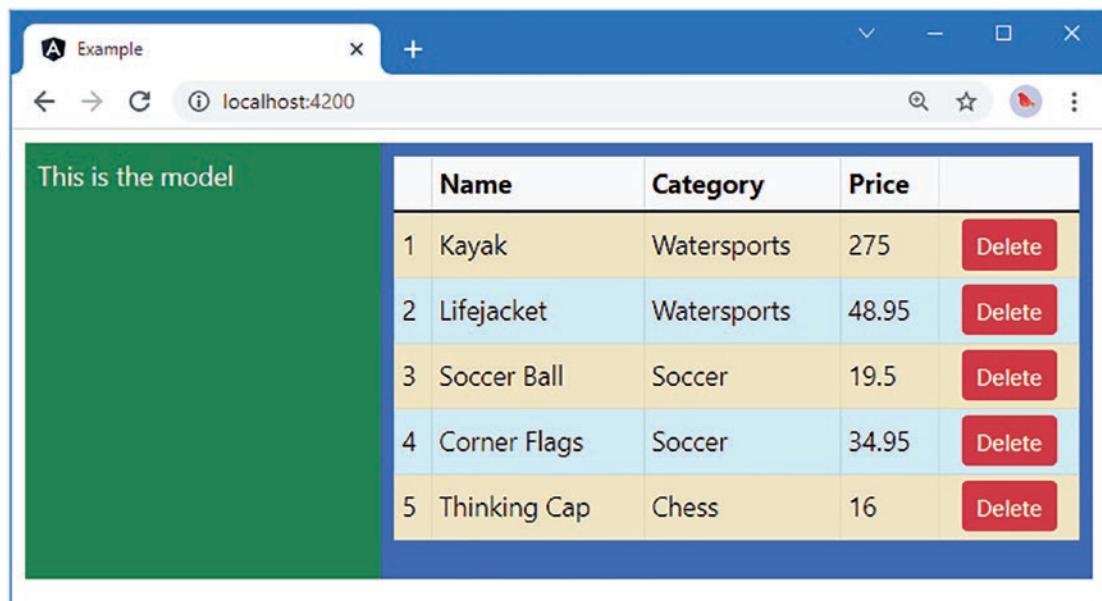
Using Directives in a Child Component Template

Once the input property has been defined, the child component can use the full range of data bindings and directives, either by using the data provided through the parent component or by defining its own. In Listing 15-13, I have restored the original table functionality from earlier chapters that displays a list of the Product objects in the data model, along with a checkbox that determines whether the table is displayed. This functionality was previously managed by the root component and its template.

Listing 15-13. Restoring the Table in the productTable.component.html File in the src/app Folder

```
<table class="table table-sm table-bordered table-striped">
  <thead class="table-light">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of getProducts(); let i = index; let odd = odd;
         let even = even" [class.table-info]="odd" [class.table-warning]="even"
         class="align-middle">
      <td>{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price}}</td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm"
               (click)="deleteProduct(item.id)">
          Delete
        </button>
      </td>
    </tr>
  </tbody>
</table>
```

The same HTML elements, data bindings, and directives (including custom directives like `paIf` and `paFor`) are used, producing the result shown in Figure 15-10. The key difference is not in the appearance of the table but in the way that it is now managed by a dedicated component.



The screenshot shows a web browser window with the URL `localhost:4200`. On the left, there is a green sidebar with the text "This is the model". To the right is a table component with the following data:

	Name	Category	Price	
1	Kayak	Watersports	275	<button>Delete</button>
2	Lifejacket	Watersports	48.95	<button>Delete</button>
3	Soccer Ball	Soccer	19.5	<button>Delete</button>
4	Corner Flags	Soccer	34.95	<button>Delete</button>
5	Thinking Cap	Chess	16	<button>Delete</button>

Figure 15-10. Restoring the table display

Using Output Properties to Coordinate Between Components

Child components can use output properties that define custom events that signal important changes and that allow the parent component to respond when they occur. Listing 15-14 changes the form component, adding an external template and an output property that will be triggered when the user creates a new Product object when invoking the submitForm method.

Listing 15-14. Defining an Output Property in the productForm.component.ts File in the src/app Folder

```
import { Component, Output, EventEmitter } from "@angular/core";
import { Product } from "./product.model";

@Component({
  selector: "paProductForm",
  templateUrl: "productForm.component.html"
})
export class ProductFormComponent {
  newProduct: Product = new Product();

  @Output("paNewProduct")
  newProductEvent = new EventEmitter<Product>();

  submitForm(form: any) {
    this.newProductEvent.emit(this.newProduct);
    this.newProduct = new Product();
    form.resetForm();
  }
}
```

The output property is called newProductEvent, and the component triggers it when the submitForm method is called. Aside from the output property, the additions in the listing are based on the logic in the root controller, which previously managed the form. I also removed the inline template and created a file called productForm.component.html in the src/app folder, with the content shown in Listing 15-15.

Listing 15-15. The Contents of the productForm.component.html File in the src/app Folder

```
<form #form="ngForm" (ngSubmit)="submitForm(form)">
  <div class="form-group">
    <label>Name</label>
    <input class="form-control"
      name="name" [(ngModel)]="newProduct.name" />
  </div>
  <div class="form-group">
    <label>Category</label>
    <input class="form-control"
      name="category" [(ngModel)]="newProduct.category" />
  </div>
  <div class="form-group">
    <label>Price</label>
    <input class="form-control"
      name="name" [(ngModel)]="newProduct.price" />
  </div>
```

```

<button class="btn btn-primary mt-2" type="submit">
    Create
</button>
</form>

```

The form contains standard elements, configured using two-way bindings. The child component's host element acts as the bridge to the parent component, which can register interest in the custom event, as shown in Listing 15-16.

Listing 15-16. Registering for the Custom Event in the template.html File in the src/app Folder

```

<div class="container-fluid">
    <div class="row p-2">
        <div class="col-4 p-2 text-dark">
            <paProductForm (paNewProduct)="addProduct($event)"></paProductForm>
        </div>
        <div class="col p-2">
            <paProductTable [model]="model"></paProductTable>
        </div>
    </div>
</div>

```

The new binding handles the custom event by passing the event object to the addProduct method. The child component is responsible for managing the form elements and validating their contents. When the data passes validation, the custom event is triggered, and the data binding expression is evaluated in the context of the parent component, whose addProduct method adds the new object to the model. Since the model has been shared with the table child component through its input property, the new data is displayed to the user, as shown in Figure 15-11. (You may need to restart the Angular development tools to include the new template file in the build process.)

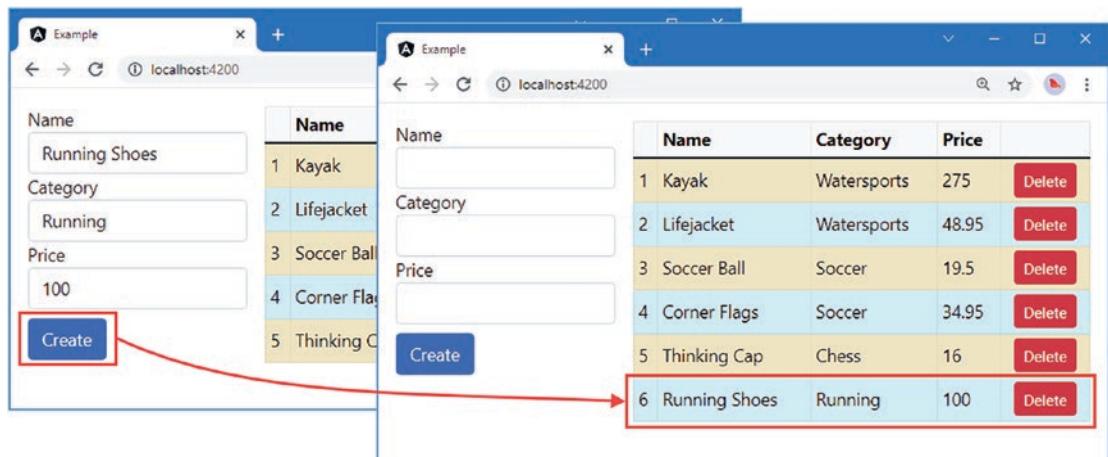


Figure 15-11. Using a custom event in a child component

Projecting Host Element Content

If the host element for a component contains content, it can be included in the template using the special `ng-content` element. This is known as *content projection*, and it allows components to be created that combine the content in their template with the content in the host element. To demonstrate, I added a file called `toggleView.component.ts` to the `src/app` folder and used it to define the component shown in Listing 15-17.

Listing 15-17. The Contents of the `toggleView.component.ts` File in the `src/app` Folder

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

@Component({
  selector: "paToggleView",
  templateUrl: "toggleView.component.html"
})
export class PaToggleView {

  showContent: boolean = true;
}
```

This component defines a `showContent` property that will be used to determine whether the host element's content will be displayed within the template. To provide the template, I added a file called `toggleView.component.html` to the `src/app` folder and added the elements shown in Listing 15-18.

Listing 15-18. The Contents of the `toggleView.component.html` File in the `src/app` Folder

```
<div class="form-check">
  <label class="form-check-label">Show Content</label>
  <input class="form-check-input" type="checkbox" [(ngModel)]="showContent" />
</div>
<ng-content *ngIf="showContent"></ng-content>
```

The important element is `ng-content`, which Angular will replace with the content of the host element. The `ngIf` directive has been applied to the `ng-content` element so that it will be visible only if the checkbox in the template is checked. Listing 15-19 registers the component with the Angular module.

Listing 15-19. Registering the Component in the `app.module.ts` File in the `src/app` Folder

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { PaAttrDirective } from './attr.directive';
import { PaModel } from './twoway.directive';
import { PaStructureDirective } from './structure.directive';
import { PaIteratorDirective } from './iterator.directive';
import { PaCellColor } from './cellColor.directive';
```

```

import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

The final step is to apply the new component to a host element that contains content, as shown in Listing 15-20.

Listing 15-20. Adding a Host Element with Content in the template.html File in the src/app Folder

```

<div class="container-fluid">
  <div class="row p-2">
    <div class="col-4 p-2 text-dark">
      <paProductForm (paNewProduct)="addProduct($event)"></paProductForm>
    </div>
    <div class="col p-2">
      <b><paToggleView></paToggleView></b>
      <paProductTable [model]="model"></paProductTable>
    </paToggleView>
    </div>
  </div>
</div>

```

The `paToggleView` element is the host for the new component, and it contains the `paProductTable` element, which applies the component that creates the product table. The result is that there is a checkbox that controls the visibility of the table, as shown in Figure 15-12. The new component has no knowledge of the content of its host element, and its inclusion in the template is possible only through the `ng-content` element.

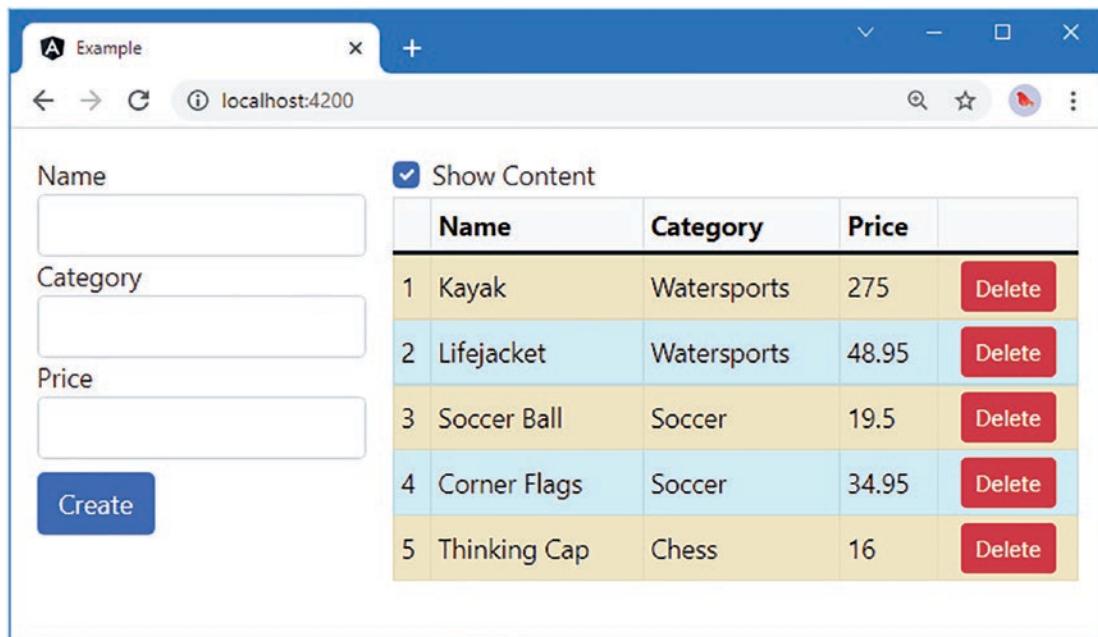


Figure 15-12. Including host element content in the template

SELECTING COMPONENTS DYNAMICALLY

The ViewContainerRef class, introduced in Chapter 14, defines the createComponent method, which can be used to select a component programmatically, without having to specify a fixed element in a template. I have not demonstrated this feature because it has serious limitations and causes more problems than it addresses, at least in my experience. If you want to explore this feature, see the Angular documentation at <https://angular.io/guide/dynamic-component-loader>, but proceed with caution.

Completing the Component Restructure

The functionality that was previously contained in the root component has been distributed to the new child components. All that remains is to tidy up the root component to remove the code that is no longer required, as shown in Listing 15-21.

Listing 15-21. Removing Obsolete Code in the component.ts File in the src/app Folder

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
```

```

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  model: Model = new Model();
  // showTable: boolean = false;
  // darkColor: boolean = false;

  // getProduct(key: number): Product | undefined {
  //   return this.model.getProduct(key);
  // }

  // getProducts(): Product[] {
  //   return this.model.getProducts();
  // }

  // newProduct: Product = new Product();

  addProduct(p: Product) {
    this.model.saveProduct(p);
  }

  // deleteProduct(key: number) {
  //   this.model.deleteProduct(key);
  // }

  // submitForm() {
  //   this.addProduct(this.newProduct);
  // }
}

```

Many of the responsibilities of the root component have been moved elsewhere in the application. Of the original list from the start of the chapter, only the following remain the responsibility of the root component:

- Providing Angular with an entry point into the application, as the root component
- Providing access to the application's data model so that it can be used in data bindings

The child components have assumed the rest of the responsibilities, providing self-contained blocks of functionality that are simpler, easier to develop, and easier to maintain and that can be reused as required.

Using Component Styles

Components can define styles that apply only to the content in their templates, which allows content to be styled by a component without it being affected by the styles defined by its parents or other antecedents and without affecting the content in its child and other descendant components. Styles can be defined inline using the `styles` property of the `@Component` decorator, as shown in Listing 15-22.

Listing 15-22. Defining Inline Styles in the productForm.component.ts File in the src/app Folder

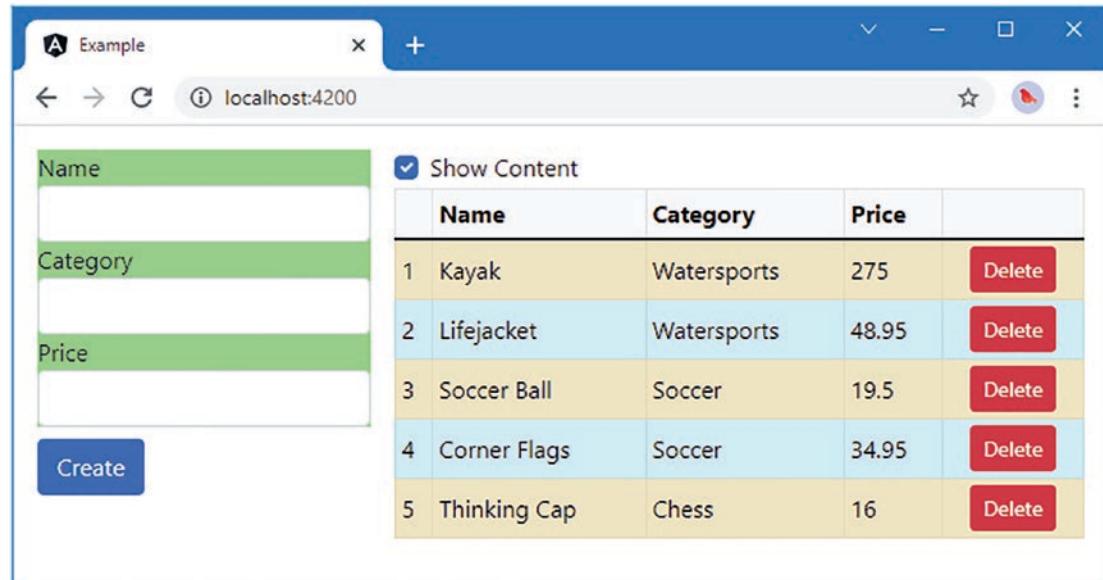
```
import { Component, Output, EventEmitter } from "@angular/core";
import { Product } from "./product.model";

@Component({
  selector: "paProductForm",
  templateUrl: "productForm.component.html",
  styles: ["div { background-color: lightgreen }"]
})
export class ProductFormComponent {
  newProduct: Product = new Product();

  @Output("paNewProduct")
  newProductEvent = new EventEmitter<Product>();

  submitForm(form: any) {
    this.newProductEvent.emit(this.newProduct);
    this.newProduct = new Product();
    form.resetForm();
  }
}
```

The styles property is set to an array, where each item contains a CSS selector and one or more properties. In the listing, I have specified styles that set the background color of div elements to lightgreen. Even though there are div elements throughout the combined HTML document, this style will affect only the elements in the template of the component that defines them, which is the form component in this case, as shown in Figure 15-13.

**Figure 15-13.** Defining inline component styles

Tip The styles included in the bundles created by the development tools are still applied, which is why the elements are still styled using Bootstrap.

Defining External Component Styles

Inline styles offer the same benefits and drawbacks as inline templates: they are simple and keep everything in one file, but they can be hard to read, can be hard to manage, and can confuse code editors.

The alternative is to define styles in a separate file and associate them with a component using the `styleUrls` property in its decorator. External style files follow the same naming convention as templates and code files. I added a file called `productForm.component.css` to the `src/app` folder and used it to define the styles shown in Listing 15-23.

Listing 15-23. The Contents of the `productForm.component.css` File in the `src/app` Folder

```
div {
    background-color: lightcoral;
}
```

This is the same style that was defined inline but with a different color value to confirm that this is the CSS being used by the component. In Listing 15-24, the component's decorator has been updated to specify the styles file.

Listing 15-24. Using External Styles in the `productForm.component.ts` File in the `src/app` Folder

```
import { Component, Output, EventEmitter } from "@angular/core";
import { Product } from "./product.model";

@Component({
    selector: "paProductForm",
    templateUrl: "productForm.component.html",
    //styles: ["div { background-color: lightgreen }"],
    styleUrls: ["productForm.component.css"]
})
export class ProductFormComponent {
    newProduct: Product = new Product();

    @Output("paNewProduct")
    newProductEvent = new EventEmitter<Product>();

    submitForm(form: any) {
        this.newProductEvent.emit(this.newProduct);
        this.newProduct = new Product();
        form.resetForm();
    }
}
```

The `styleUrls` property is set to an array of strings, each of which specifies a CSS file. Figure 15-14 shows the effect of adding the external styles file.

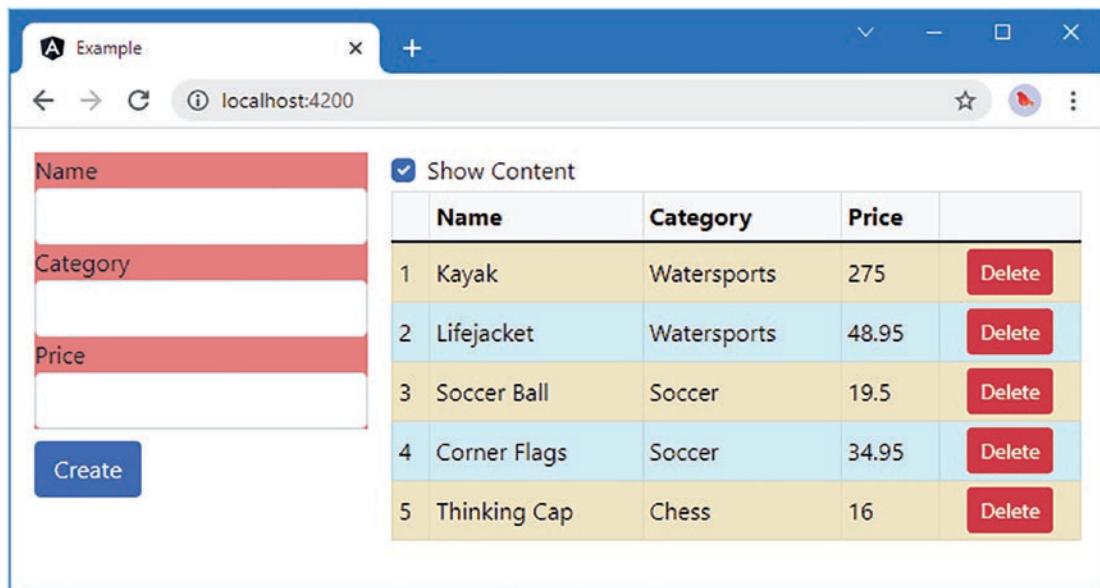


Figure 15-14. Defining external component styles

Using Advanced Style Features

Defining styles in components is a useful feature, but you won't always get the results you expect. Some advanced features allow you to take control of how component styles work.

Setting View Encapsulation

By default, component-specific styles are implemented by writing the CSS that has been applied to the component so that it targets special attributes, which Angular then adds to all of the top-level elements contained in the component's template. If you inspect the DOM using the browser's F12 developer tools, you will see that the contents of the external CSS file in Listing 15-23 have been rewritten like this:

```
...
<style>
  div[_ngcontent-oni-c43] {
    background-color: lightcoral;
  }
</style>
...
```

The selector has been modified so that it matches `div` elements with an attribute called `_ngcontent-oni-c43` (although you may see a different name in your browser since the name of the attribute is generated dynamically by Angular).

To ensure that the CSS in the style element affects only the HTML elements managed by the component, the elements in the template are modified so they have the same dynamically generated attribute, like this:

```
...
<div _ngcontent-oni-c43="" class="form-group">
  <label _ngcontent-oni-c43="">Name</label>
  <input _ngcontent-oni-c43="" name="name" class="form-control ng-untouched
    ng-pristine ng-valid" ng-reflect-name="name">
</div>

<div _ngcontent-jwe-c40="" class="form-group">
  <label _ngcontent-jwe-c40="">Name</label>
  <input _ngcontent-jwe-c40="" name="name" class="form-control ng-untouched
    ng-pristine ng-valid" ng-reflect-name="name">
</div>
...
```

This is known as the component's *view encapsulation* behavior, and what Angular is doing is emulating a feature known as the *shadow DOM*, which allows sections of the Domain Object Model to be isolated so they have their own scope, meaning that JavaScript, styles, and templates can be applied to part of the HTML document.

The shadow DOM feature is supported by most modern browsers, but its path to adoption has been messy, and Angular emulates the feature to ensure consistency. There are two other encapsulation options in addition to emulation, which are set using the `encapsulation` property in the `@Component` decorator.

Tip You can learn more about the shadow DOM at http://developer.mozilla.org/en-US/docs/Web/Web_Components/Shadow_DOM. You can see which browsers support the shadow DOM feature at <https://caniuse.com/shadowdomv1>.

The `encapsulation` property is assigned a value from the `ViewEncapsulation` enumeration, which is defined in the `@angular/core` module, and it defines the values described in Table 15-4.

Table 15-4. The `ViewEncapsulation` Values

Name	Description
Emulated	When this value is specified, Angular emulates the shadow DOM by writing content and styles to add attributes, as described earlier. This is the default behavior if no <code>encapsulation</code> value is specified in the <code>@Component</code> decorator.
ShadowDom	When this value is specified, Angular uses the browser's shadow DOM feature. This will work only if the browser implements the shadow DOM.
None	When this value is specified, Angular simply adds the unmodified CSS styles to the head section of the HTML document and lets the browser figure out how to apply the styles using the normal CSS precedence rules.

The `ShadowDom` and `None` values should be used with caution. Browser support for the shadow DOM feature is improving but has been patchy and was made more complex because there was an earlier version of the shadow DOM feature that was abandoned in favor of the current approach.

The `None` option adds all the styles defined by components to the head section of the HTML document and lets the browser figure out how to apply them. This has the benefit of working in all browsers, but the results are unpredictable, and there is no isolation between the styles defined by different components.

One important consideration is that the `Emulated` setting doesn't produce the same results as enabling native shadow DOM support with the `ShadowDom` setting. The `Emulated` setting ensures that styles defined for a specific component are not applied to elements generated by other components, but it still allows elements to be styled by the global CSS styles, such as those defined by the Bootstrap CSS package. When the `ShadowDom` setting is used, the browser completely isolates an element, which prevents elements from being affected by global CSS styles. To demonstrate, Listing 15-25 enables the `ShadowDom` mode.

Listing 15-25. Setting View Encapsulation in the `productForm.component.ts` File in the `src/app` Folder

```
import { Component, Output, EventEmitter, ViewEncapsulation } from "@angular/core";
import { Product } from "./product.model";

@Component({
  selector: "pa-productform",
  templateUrl: "productForm.component.html",
  styleUrls: ["productForm.component.css"],
  encapsulation: ViewEncapsulation.ShadowDom
})
export class ProductFormComponent {
  newProduct: Product = new Product();

  @Output("paNewProduct")
  newProductEvent = new EventEmitter<Product>();

  submitForm(form: any) {
    this.newProductEvent.emit(this.newProduct);
    this.newProduct = new Product();
    form.resetForm();
  }
}
```

The selectors for components that use the `ShadowDom` setting must be all lowercase and contain a hyphen, which is why I changed the selector to `pa-productform`. Listing 15-26 updates the template to match the new selector.

Listing 15-26. Updating an Element in the `template.html` File in the `src/app` Folder

```
<div class="container-fluid">
  <div class="row p-2">
    <div class="col-4 p-2 text-dark">
      <pa-productform (paNewProduct)="addProduct($event)"></pa-productform>
    </div>
    <div class="col p-2">
      <paToggleView>
        <paProductTable [model]="model"></paProductTable>
      </paToggleView>
    </div>
  </div>
</div>
```

```

    </div>
  </div>
</div>

```

Figure 15-15 shows how the styles defined in the `productForm.component.css` file are applied to the HTML elements generated by the component, but the globally defined Bootstrap styles, added to the project in Chapter 9, are not.

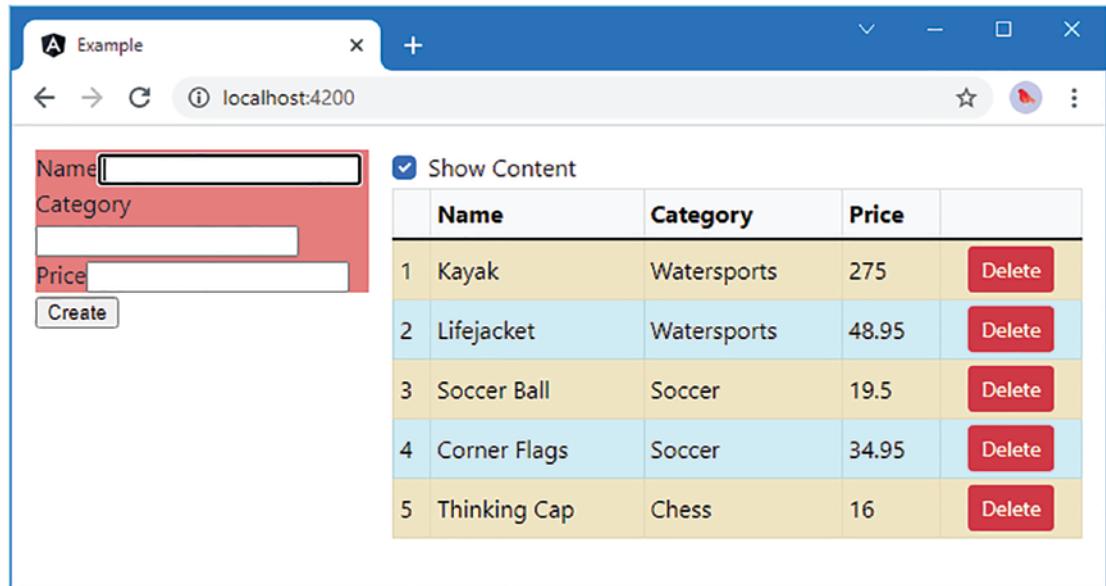


Figure 15-15. Enabling the native shadow DOM feature

Using the Shadow DOM CSS Selectors

Using the shadow DOM means that there are boundaries that regular CSS selectors do not operate across. To help address this, there are special CSS selectors that are useful when using styles that rely on the shadow DOM (even when it is being emulated), as described in Table 15-5 and demonstrated in the sections that follow.

Table 15-5. The Shadow DOM CSS Selectors

Name	Description
<code>:host</code>	This selector is used to match the component's host element.
<code>:host-context(classSelector)</code>	This selector is used to match the ancestors of the host element that are members of a specific class.
<code>/deep/</code> , <code>>>></code> , or <code>::ng-deep</code>	These selectors are used by a parent component to define styles that affect the elements in child component templates. This selector should be used only when the <code>@Component</code> decorator's <code>encapsulation</code> property is set to <code>emulated</code> , as described in the "Setting View Encapsulation" section.

Selecting the Host Element

A component's host element appears outside of its template, which means that the selectors in its styles apply only to elements that the host element contains and not the element itself. This can be addressed by using the `:host` selector, which matches the host element. Listing 15-27 defines a style that is applied only when the mouse pointer is hovering over the host element, which is specified by combining the `:host` and `:hover` selectors.

Listing 15-27. Matching the Host Element in the productForm.component.css File in the src/app Folder

```
div {
    background-color: lightcoral;
}
:host :hover {
    font-size: 25px;
}
```

When the mouse pointer is over the host element, its `font-size` property will be set to `25px`, which increases the text size to 25 points for all the elements in the form, as shown in Figure 15-16.

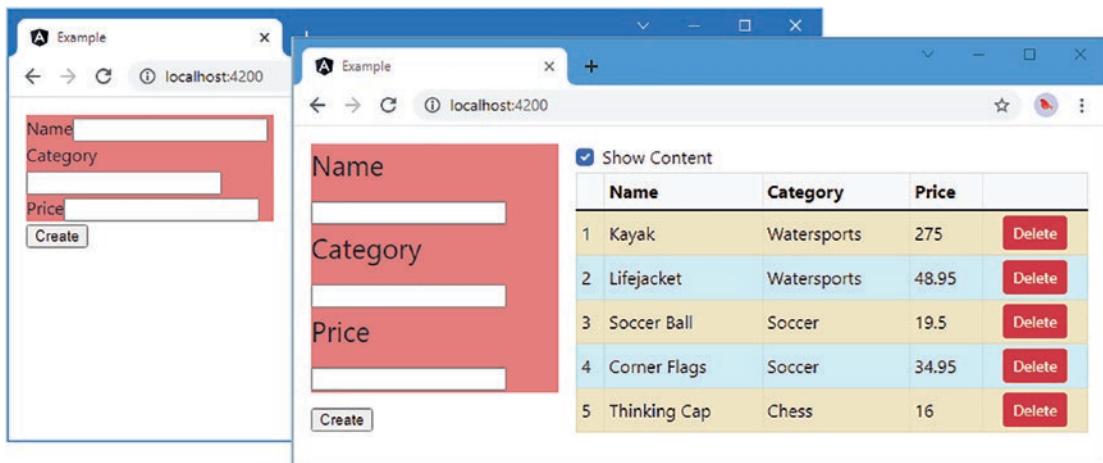


Figure 15-16. Selecting the host element in a component style

Selecting the Host Element's Ancestors

The `:host-context` selector is used to style elements within the component's template based on the class membership of the host element's ancestors (which are outside the template). This is a more limited selector than `:host` and cannot be used to specify anything other than a class selector, without support for matching tag types, attributes, or any other selector. Listing 15-28 shows the use of the `:host-context` selector.

Listing 15-28. Selecting Ancestors in the productForm.component.css File in the src/app Folder

```
div {
    background-color: lightcoral;
}
```

```
:host:hover {
    font-size: 25px;
}
:host-context(.angularApp) input {
    background-color: lightgray;
}
```

The selector in the listing sets the background-color property of input elements within the component's template to lightgrey only if one of the host element's ancestor elements is a member of a class called angularApp. In Listing 15-29, I have added a div element in the template.html file to the angularApp class.

Listing 15-29. Adding the Host Element to a Class in the template.html File in the src/app Folder

```
<div class="container-fluid angularApp">
    <div class="row p-2">
        <div class="col-4 p-2 text-dark">
            <pa-productform (paNewProduct)="addProduct($event)"></pa-productform>
        </div>
        <div class="col p-2">
            <paToggleView>
                <paProductTable [model]="model"></paProductTable>
            </paToggleView>
        </div>
    </div>
</div>
```

Figure 15-17 shows the effect of the selector before and after the changes in Listing 15-29.

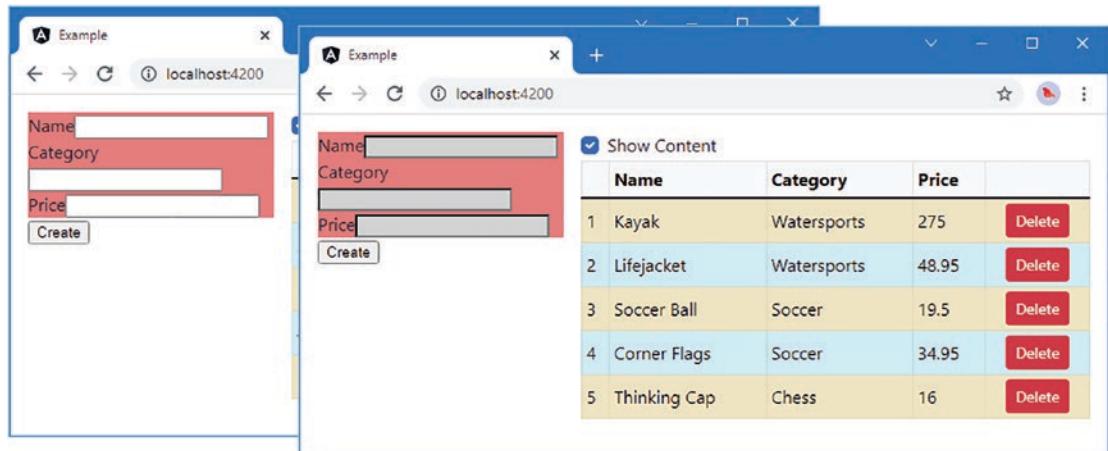


Figure 15-17. Selecting the host element's ancestors

Pushing a Style into the Child Component's Template

Styles defined by a component are not automatically applied to the elements in the child component's templates. As a demonstration, Listing 15-30 adds a style to the `@Component` decorator of the root component.

Listing 15-30. Defining Styles in the component.ts File in the src/app Folder

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
  selector: "app",
  templateUrl: "template.html",
  styles: ["div { border: 2px black solid; font-style:italic }"]
})
export class ProductComponent {
  model: Model = new Model();

  addProduct(p: Product) {
    this.model.saveProduct(p);
  }
}
```

The selector matches all `div` elements and applies a border and changes the font style. Figure 15-18 shows the result.

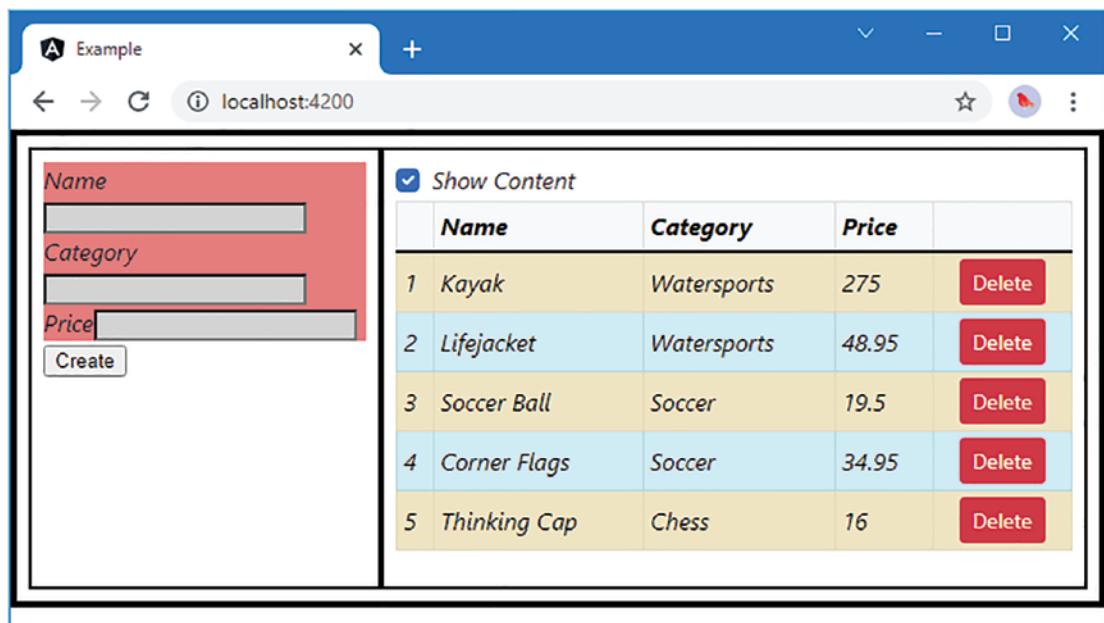


Figure 15-18. Applying regular CSS styles

Some CSS style properties, such as `font-style`, are inherited by default, which means that setting such a property in a parent component will affect the elements in child component templates because the browser automatically applies the style.

Other properties, such as `border`, are not inherited by default, and setting such a property in a parent component does not affect child component templates unless the `/deep/` or `>>>` selector is used, as shown in Listing 15-31. (These selectors are aliases of one another and have the same effect.)

Listing 15-31. Pushing a Style into Child Templates in the component.ts File in the src/app Folder

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
  selector: "app",
  templateUrl: "template.html",
  styles: ["/deep/ div { border: 2px black solid; font-style:italic }"]
})
export class ProductComponent {
  model: Model = new Model();

  addProduct(p: Product) {
    this.model.saveProduct(p);
  }
}
```

The selector for the style uses `/deep/` to push the styles into the child components' templates, which means that all the `div` elements are given a border, as shown in Figure 15-19.

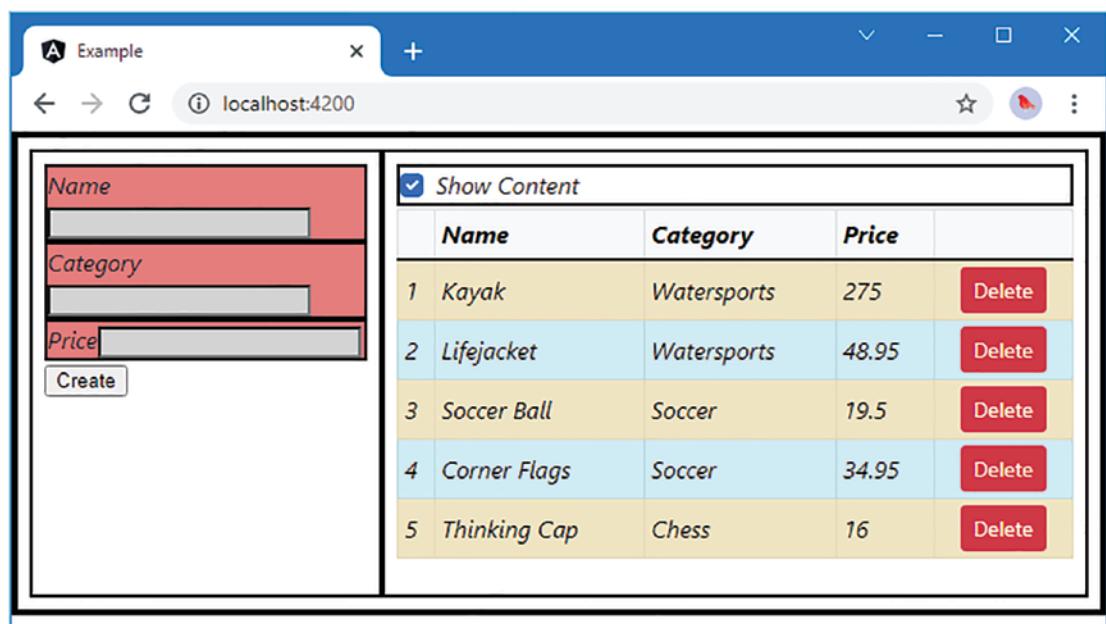


Figure 15-19. Pushing a style into child component templates

Querying Template Content

Components can query the content of their templates to locate instances of directives or components, which are known as *view children*. These are similar to the directive content children queries that were described in Chapter 14 but with some important differences.

In Listing 15-32, I have added some code to the component that manages the table that queries for the PaCellColor directive that was created to demonstrate directive content queries. This directive is still registered in the Angular module and selects td elements, so Angular will have applied it to the cells in the table component's content.

Listing 15-32. Selecting View Children in the productTable.component.ts File in the src/app Folder

```
import { Component, Input, QueryList, ViewChildren } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { PaCellColor } from "./cellColor.directive";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {

  @Input("model")
  dataModel: Model | undefined;

  getProduct(key: number): Product | undefined {
    return this.dataModel?.getProduct(key);
  }

  getProducts(): Product[] | undefined {
    return this.dataModel?.getProducts();
  }

  deleteProduct(key: number) {
    this.dataModel?.deleteProduct(key);
  }

  showTable: boolean = true;

  @ViewChildren(PaCellColor)
  viewChildren: QueryList<PaCellColor> | undefined;

  ngAfterViewInit() {
    this.viewChildren?.changes.subscribe(() => {
      this.updateViewChildren();
    });
    this.updateViewChildren();
  }

  private updateViewChildren() {
```

```

    setTimeout(() => {
      this.viewChildren?.forEach((child, index) => {
        child.setColor(index % 2 ? true : false);
      })
    }, 0);
}

```

Two property decorators are used to query for directives or components defined in the template, as described in Table 15-6.

Table 15-6. The View Children Query Property Decorators

Name	Description
@ViewChild(class)	This decorator tells Angular to query for the first directive or component object of the specified type and assign it to the property. The class name can be replaced with a template variable. Multiple classes or variable names can be separated by commas.
@ViewChildren(class)	This decorator assigns all the directive and component objects of the specified type to the property. Template variables can be used instead of classes, and multiple values can be separated by commas. The results are provided in a <code>QueryList</code> object, described in Chapter 14.

In the listing, I used the `@ViewChildren` decorator to select all the `PaCellColor` objects from the component's template. Aside from the different property decorators, components have two different lifecycle methods that are used to provide information about how the template has been processed, as described in Table 15-7.

Table 15-7. The Additional Component Lifecycle Methods

Name	Description
<code>ngAfterViewInit</code>	This method is called when the component's view has been initialized. The results of the view queries are set before this method is invoked.
<code>ngAfterViewChecked</code>	This method is called after the component's view has been checked as part of the change detection process.

In the listing, I implement the `ngAfterViewInit` method to ensure that Angular has processed the component's template and set the result of the query. Within the method I perform the initial call to the `updateViewChildren` method, which operates on the `PaCellColor` objects, and I set up the function that will be called when the query results change, using the `QueryList.changes` property, as described in Chapter 14. The view children are updated within a call to the `setTimeout` function, which ensures that the changes are applied without triggering the change detection guard. The result is that the color of every second table cell is changed, as shown in Figure 15-20.

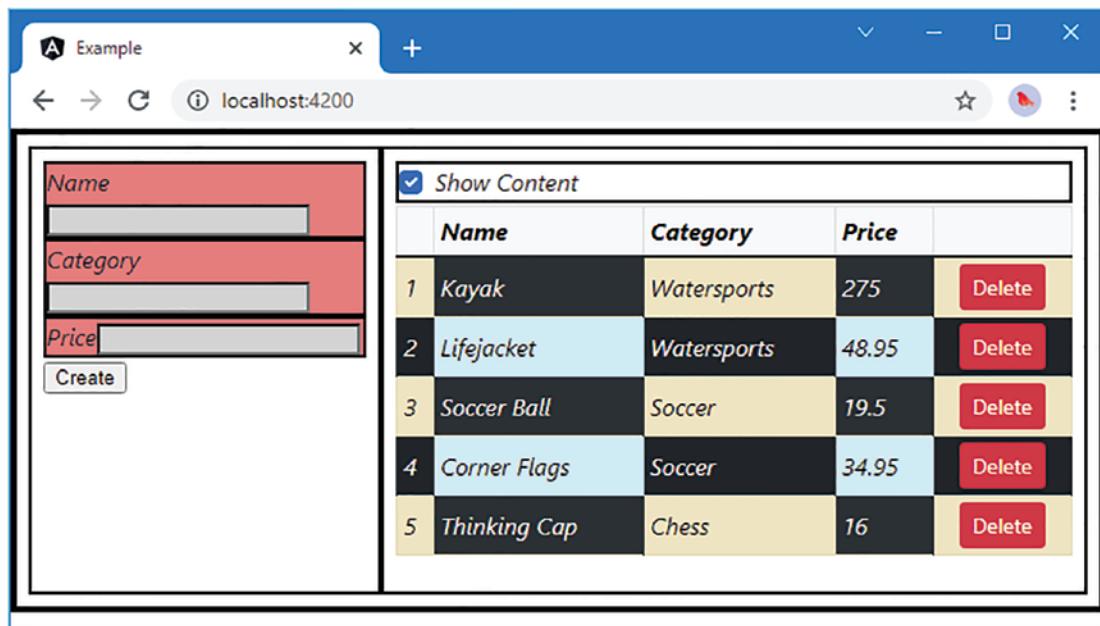


Figure 15-20. Querying for view children

Tip You may need to combine view child and content child queries if you have used the `ng-content` element. The content defined in the template is queried using the technique shown in Listing 15-32, but the project content—which replaces the `ng-content` element—is queried using the child queries described in Chapter 14.

Summary

In this chapter, I revisited the topic of components and explained how to combine all the features of directives with the ability to provide their own templates. I explained how to structure an application to create small module components and how components can coordinate between themselves using input and output properties. I also showed how components can define CSS styles that are applied only to their templates and no other parts of the application. In the next chapter, I introduce pipes, which are used to prepare data for display in templates.

CHAPTER 16



Using and Creating Pipes

Pipes are small fragments of code that transform data values so they can be displayed to the user in templates. Pipes allow transformation logic to be defined in self-contained classes so that it can be applied consistently throughout an application. Table 16-1 puts pipes in context.

Table 16-1. Putting Pipes in Context

Question	Answer
What are they?	Pipes are classes that are used to prepare data for display to the user.
Why are they useful?	Pipes allow preparation logic to be defined in a single class that can be used throughout an application, ensuring that data is presented consistently.
How are they used?	The @Pipe decorator is applied to a class and used to specify a name by which the pipe can be used in a template.
Are there any pitfalls or limitations?	Pipes should be simple and focused on preparing data. It can be tempting to let the functionality creep into areas that are the responsibility of other building blocks, such as directives or components.
Are there any alternatives?	You can implement data preparation code in components or directives, but that makes it harder to reuse in other parts of the application.

Table 16-2 summarizes the chapter.

Table 16-2. Chapter Summary

Problem	Solution	Listing
Formatting a data value for inclusion in a template	Use a pipe in a data binding expression	1–6
Creating a custom pipe	Apply the @Pipe decorator to a class	7–9
Formatting a data value using multiple pipes	Chain the pipe names together using the bar character	10
Specifying when Angular should reevaluate the output from a pipe	Use the pure property of the @Pipe decorator	11–14
Formatting numerical values	Use the number pipe	15, 16

(continued)

Table 16-2. (continued)

Problem	Solution	Listing
Formatting currency values	Use the currency pipe	17, 18
Formatting percentage values	Use the percent pipe	19
Formatting dates	Use the date pipe	20-22
Changing the case of strings	Use the uppercase or lowercase pipe	23, 24
Serializing objects into the JSON format	Use the json pipe	25
Selecting elements from an array	Use the slice pipe	26
Formatting an object or map as key-value pairs	Use the keyvalue pipe	27
Selecting a value to display for a string or number value	Use the i18nSelect or i18nPlural pipe	28-31
Display events from an observable	Use the async pipe	32-34

Preparing the Example Project

I am going to continue working with the example project that was first created in Chapter 9 and that has been expanded and modified in the chapters since. In the final examples in the previous chapter, component styles and view children queries left the application with a strikingly garish appearance that I am going to tone down for this chapter. In Listing 16-1, I have disabled the inline component styles applied to the form elements.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 16-1. Disabling CSS Styles in the productForm.component.ts File in the src/app Folder

```
import { Component, Output, EventEmitter, ViewEncapsulation } from "@angular/core";
import { Product } from "./product.model";

@Component({
  selector: "pa-productform",
  templateUrl: "productForm.component.html",
  // styleUrls: ["productForm.component.css"],
  // encapsulation: ViewEncapsulation.ShadowDom
})
export class ProductFormComponent {
  newProduct: Product = new Product();

  @Output("paNewProduct")
  newProductEvent = new EventEmitter<Product>();
```

```

    submitForm(form: any) {
        this.newProductEvent.emit(this.newProduct);
        this.newProduct = new Product();
        form.resetForm();
    }
}

```

To disable the checkerboard coloring of the table cells, I changed the selector for the `PaCellColor` directive so that it matches an attribute that is not currently applied to the HTML elements, as shown in Listing 16-2.

Listing 16-2. Changing the Selector in the `cellColor.directive.ts` File in the `src/app` Folder

```

import { Directive, HostBinding } from "@angular/core";

@Directive({
    selector: "td[paApplyColor]"
})
export class PaCellColor {

    @HostBinding("class")
    bgClass: string = "";

    setColor(dark: Boolean) {
        this.bgClass = dark ? "table-dark" : "";
    }
}

```

Listing 16-3 disables the deep styles defined by the root component.

Listing 16-3. Disabling CSS Styles in the `component.ts` File in the `src/app` Folder

```

import { Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
    selector: "app",
    templateUrl: "template.html",
    //styles: ["/deep/ div { border: 2px black solid; font-style:italic }"]
})
export class ProductComponent {
    model: Model = new Model();

    addProduct(p: Product) {
        this.model.saveProduct(p);
    }
}

```

The next change is to simplify the `ProductTableComponent` class to remove methods and properties that are no longer required and add new properties that will be used in later examples, as shown in Listing 16-4.

Listing 16-4. Simplifying the Code in the productTable.component.ts File in the src/app Folder

```

import { Component, Input, QueryList, ViewChildren } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
//import { PaCellColor } from "./cellcolor.directive";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {

  @Input("model")
  dataModel: Model | undefined;

  getProduct(key: number): Product | undefined {
    return this.dataModel?.getProduct(key);
  }

  getProducts(): Product[] | undefined {
    return this.dataModel?.getProducts();
  }

  deleteProduct(key: number) {
    this.dataModel?.deleteProduct(key);
  }

  // showTable: boolean = true;

  // @ViewChildren(PaCellColor)
  // viewChildren: QueryList<PaCellColor> | undefined;

  // ngAfterViewInit() {
  //   this.viewChildren?.changes.subscribe(() => {
  //     this.updateViewChildren();
  //   });
  //   this.updateViewChildren();
  // }

  // private updateViewChildren() {
  //   setTimeout(() => {
  //     this.viewChildren?.forEach((child, index) => {
  //       child.setColor(index % 2 ? true : false);
  //     })
  //   }, 0);
  // }

  taxRate: number = 0;
  categoryFilter: string | undefined;
  itemCount: number = 3;
}

```

Finally, I have removed one of the component elements from the root component's template to disable the checkbox that shows and hides the table, as shown in Listing 16-5.

Listing 16-5. Simplifying the Elements in the template.html File in the src/app Folder

```
<div class="container-fluid angularApp">
  <div class="row p-2">
    <div class="col-4 p-2 text-dark">
      <pa-productform (paNewProduct)="addProduct($event)"></pa-productform>
    </div>
    <div class="col p-2">
      <!-- <paToggleView> -->
      <paProductTable [model]="model"></paProductTable>
      <!-- </paToggleView> -->
    </div>
  </div>
</div>
```

Run the following command in the example folder to start the Angular development tools:

```
ng serve
```

Open a new browser tab and navigate to <http://localhost:4200> to see the content shown in Figure 16-1.

	Name	Category	Price	
1	Kayak	Watersports	275	<button>Delete</button>
2	Lifejacket	Watersports	48.95	<button>Delete</button>
3	Soccer Ball	Soccer	19.5	<button>Delete</button>
4	Corner Flags	Soccer	34.95	<button>Delete</button>
5	Thinking Cap	Chess	16	<button>Delete</button>

Figure 16-1. Running the example application

Understanding Pipes

Pipes are classes that transform data before it is received by a directive or component. That may not sound like an important job, but pipes can be used to perform some of the most commonly required development tasks easily and consistently.

As a quick example to demonstrate how pipes are used, Listing 16-6 applies one of the built-in pipes to transform the values in the Price column of the table displayed by the application.

Listing 16-6. Using a Pipe in the productTable.component.html File in the src/app Folder

```
<table class="table table-sm table-bordered table-striped">
  <thead class="table-light">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of getProducts(); let i = index; let odd = odd;
      let even = even" [class.table-info]="odd" [class.table-warning]="even"
      class="align-middle">
      <td>{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price | currency:"USD": "symbol" }}</td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm"
          (click)="deleteProduct(item.id)">
          Delete
        </button>
      </td>
    </tr>
  </tbody>
</table>
```

The syntax for applying a pipe is similar to the style used by command prompts, where a value is “piped” for transformation using the vertical bar symbol (the | character). Figure 16-2 shows the structure of the data binding that contains the pipe.

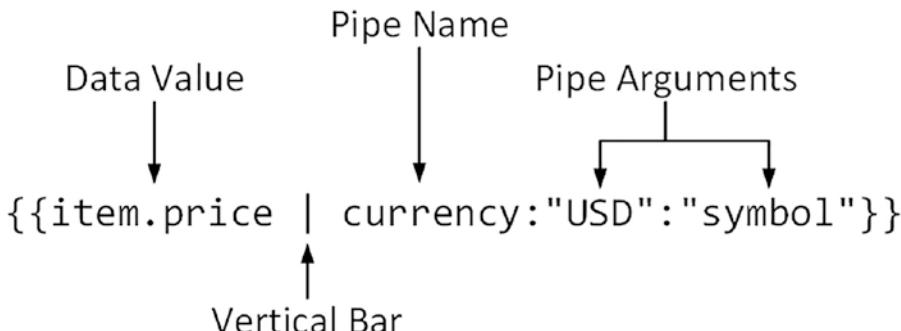
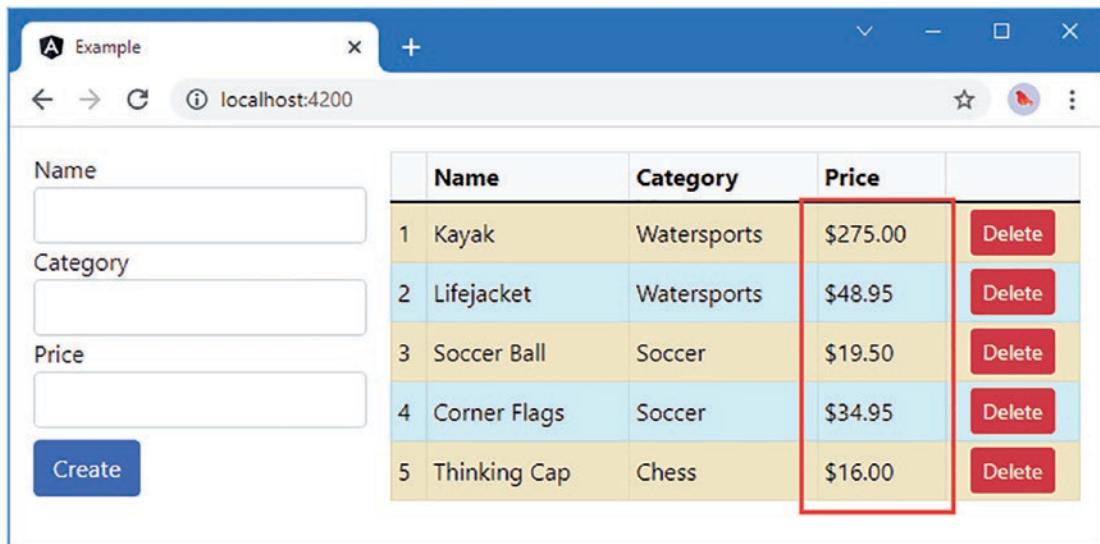


Figure 16-2. The anatomy of data binding with a pipe

The name of the pipe used in Listing 16-6 is currency, and it formats numbers into currency values. Arguments to the pipe are separated by colons (the : character). The first pipe argument specifies the currency code that should be used, which is USD in this case, representing U.S. dollars. The second pipe argument, which is symbol, specifies whether the currency symbol, rather than its code, should be displayed.

When Angular processes the expression, it obtains the data value and passes it to the pipe for transformation. The result produced by the pipe is then used as the expression result for the data binding. In the example, the bindings are string interpolations, and Figure 16-3 shows the results.



A screenshot of a web browser window titled "Example". The address bar shows "localhost:4200". The page displays a form with three input fields: "Name", "Category", and "Price", followed by a "Create" button. To the right is a table with columns "Name", "Category", and "Price". The "Price" column is highlighted with a red border. The table data is as follows:

	Name	Category	Price	
1	Kayak	Watersports	\$275.00	Delete
2	Lifejacket	Watersports	\$48.95	Delete
3	Soccer Ball	Soccer	\$19.50	Delete
4	Corner Flags	Soccer	\$34.95	Delete
5	Thinking Cap	Chess	\$16.00	Delete

Figure 16-3. The effect of using the currency pipe

Creating a Custom Pipe

I will return to the built-in pipes that Angular provides later in the chapter, but the best way to understand how pipes work and what they are capable of is to create a custom pipe. I added a file called `addTax.pipe.ts` in the `src/app` folder and defined the class shown in Listing 16-7.

Listing 16-7. The Contents of the `addTax.pipe.ts` File in the `src/app` Folder

```
import { Pipe } from "@angular/core";

@Pipe({
  name: "addTax"
})
export class PaAddTaxPipe {

  defaultRate: number = 10;

  transform(value: any, rate?: any): number {
    let valueNumber = Number.parseFloat(value);
    let rateNumber = rate == undefined ?
      this.defaultRate : rate;
    return valueNumber + (valueNumber * rateNumber);
  }
}
```

```

        this.defaultRate : Number.parseInt(rate);
        return valueNumber + (valueNumber * (rateNumber / 100));
    }
}

```

Pipes are classes to which the Pipe decorator has been applied and that implement a method called `transform`. The Pipe decorator defines two properties, which are used to configure pipes, as described in Table 16-3.

Table 16-3. The Pipe Decorator Properties

Name	Description
<code>name</code>	This property specifies the name by which the pipe is applied in templates.
<code>pure</code>	When <code>true</code> , this pipe is reevaluated only when its input value or its arguments are changed. This is the default value. See the “Creating Impure Pipes” section for details.

The example pipe is defined in a class called `PaAddTaxPipe`, and its decorator `name` property specifies that the pipe will be applied using `addTax` in templates. The `transform` method must accept at least one argument, which Angular uses to provide the data value that the pipe formats. The pipe does its work in the `transform` method, and its result is used by Angular in the binding expression. In this example, the `transform` method accepts a number value, and its result is the received value plus sales tax.

The `transform` method can also define additional arguments that are used to configure the pipe. In the example, the optional `rate` argument can be used to specify the sales tax rate, which defaults to 10 percent.

Caution Be careful when dealing with the arguments received by the `transform` method and make sure that you parse or convert them to the types you need. The TypeScript type annotations are not enforced at runtime, and Angular will pass you whatever data values it is working with.

Registering a Custom Pipe

Pipes are registered using the `declarations` property of the Angular module, as shown in Listing 16-8.

Listing 16-8. Registering a Custom Pipe in the `app.module.ts` File in the `src/app` Folder

```

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";

```

```

import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

Applying a Custom Pipe

Once a custom pipe has been registered, it can be used in data binding expressions. In Listing 16-9, I have applied the pipe to the price value in the tables and added a select element that allows the tax rate to be specified.

Listing 16-9. Applying the Custom Pipe in the productTable.component.html File in the src/app Folder

```

<div class="my-2">
  <label>Tax Rate:</label>
  <select class="form-select" [value]="taxRate || 0"
    (change)="taxRate=$any($event).target.value">
    <option value="0">None</option>
    <option value="10">10%</option>
    <option value="20">20%</option>
    <option value="50">50%</option>
  </select>
</div>

<table class="table table-sm table-bordered table-striped">
  <thead class="table-light">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of getProducts(); let i = index; let odd = odd;
      let even = even" [class.table-info]="odd" [class.table-warning]="even"
      class="align-middle">
      <td>{{i + 1}}</td>
      <td>{{item.name}}</td>
  
```

```

<td>{{item.category}}</td>
<td>{{item.price | addTax:(taxRate || 0)}}</td>
<td class="text-center">
    <button class="btn btn-danger btn-sm"
        (click)="deleteProduct(item.id)">
        Delete
    </button>
</td>
</tr>
</tbody>
</table>

```

Just for variety, I defined the tax rate entirely within the template. The select element has a binding that sets its value property to a component variable called taxRate or defaults to 0 if the property has not been defined. The event binding handles the change event and sets the value of the taxRate property. You cannot specify a fallback value when using the ngModel directive, which is why I have split up the bindings.

In applying the custom pipe, I have used the vertical bar character, followed by the value specified by the name property in the pipe's decorator. The name of the pipe is followed by a colon, which is followed by an expression that is evaluated to provide the pipe with its argument. In this case, the taxRate property will be used if it has been defined, with a fallback value of zero.

Pipes are part of the dynamic nature of Angular data bindings, and the pipe's transform method will be called to get an updated value if the underlying data value changes or if the expression used for the arguments changes. The dynamic nature of pipes can be seen by changing the value displayed by the select element, which will define or change the taxRate property, which will, in turn, update the amount added to the price property by the custom pipe, as shown in Figure 16-4.

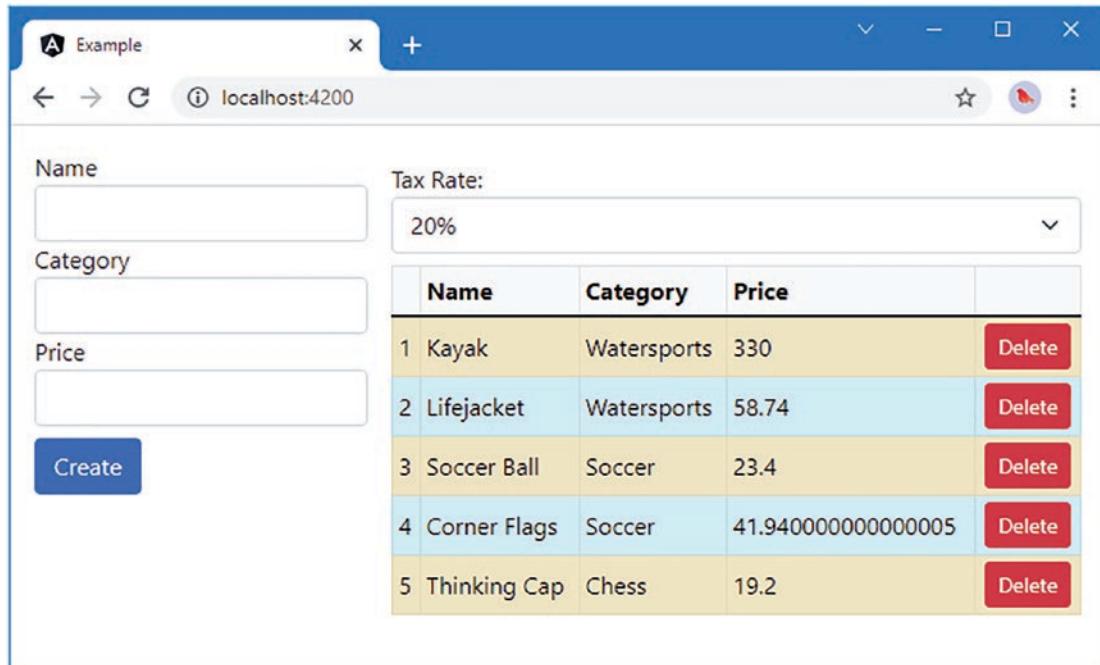


Figure 16-4. Using a custom pipe

Combining Pipes

The `addTax` pipe is applying the tax rate, but the fractional amounts that are produced by the calculation are unsightly—and unhelpful since few tax authorities insist on accuracy to 15 fractional digits.

I could fix this by adding support to the custom pipe to format the number values as currencies, but that would require duplicating the functionality of the built-in currency pipe that I used earlier in the chapter. A better approach is to combine the functionality of both pipes so that the output from the custom `addTax` pipe is fed into the built-in currency pipe, which is then used to produce the value displayed to the user.

Pipes are chained together in this way using the vertical bar character, and the names of the pipes are specified in the order that data should flow, as shown in Listing 16-10.

Listing 16-10. Combining Pipes in the `productTable.component.html` File in the `src/app` Folder

```
...
<td>{{item.price | addTax:(taxRate || 0) | currency:"USD":"symbol" }}</td>
...
```

The value of the `item.price` property is passed to the `addTax` pipe, which adds the sales tax, and then to the `currency` pipe, which formats the number value into a currency amount, as shown in Figure 16-5.

	Name	Category	Price	
1	Kayak	Watersports	\$275.00	<button>Delete</button>
2	Lifejacket	Watersports	\$48.95	<button>Delete</button>
3	Soccer Ball	Soccer	\$19.50	<button>Delete</button>
4	Corner Flags	Soccer	\$34.95	<button>Delete</button>
5	Thinking Cap	Chess	\$16.00	<button>Delete</button>

Figure 16-5. Combining the functionality of pipes

Creating Impure Pipes

The `pure` decorator property is used to tell Angular when to call the pipe's `transform` method. The default value for the `pure` property is `true`, which tells Angular that the pipe's `transform` method will generate a new value only if the input data value—the data value before the vertical bar character in the template—changes or when one or more of its arguments is modified. This is known as a *pure* pipe because it has no independent internal state and all its dependencies can be managed using the Angular change detection process.

Setting the pure decorator property to `false` creates an *impure pipe* and tells Angular that the pipe has its own state data or that it depends on data that may not be picked up in the change detection process when there is a new value.

When Angular performs its change detection process, it treats impure pipes as sources of data values in their own right and invokes the `transform` methods even when there has been no data value or argument changes.

The most common need for impure pipes is when they process the contents of arrays and the elements in the array change. As you saw in Chapter 14, Angular doesn't automatically detect changes that occur within arrays and won't invoke a pure pipe's transform method when an array element is added, edited, or deleted because it just sees the same array object being used as the input data value.

Caution Impure pipes should be used sparingly because Angular has to call the `transform` method whenever there is any data change or user interaction in the application, just in case it might result in a different result from the pipe. If you do create an impure pipe, then keep it as simple as possible. Performing complex operations, such as sorting an array, can devastate the performance of an Angular application.

As a demonstration, I added a file called `categoryFilter.pipe.ts` in the `src/app` folder and used it to define the pipe shown in Listing 16-11.

Listing 16-11. The Contents of the `categoryFilter.pipe.ts` File in the `src/app` Folder

```
import { Pipe } from "@angular/core";
import { Product } from "./product.model";

@Pipe({
  name: "filter",
  pure: true
})
export class PaCategoryFilterPipe {

  transform(products: Product[] | undefined, category: string | undefined):
    Product[] {
    if (products == undefined) {
      return [];
    }
    return category == undefined ?
      products : products.filter(p => p.category == category);
  }
}
```

This is a pure filter that receives an array of `Product` objects and returns only the ones whose `category` property matches the `category` argument. Listing 16-12 shows the new pipe registered in the Angular module.

Listing 16-12. Registering a Pipe in the `app.module.ts` File in the `src/app` Folder

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
```

```

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoWay.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

Listing 16-13 shows the application of the new pipe to the binding expression that targets the `ngFor` directive as well as a new select element that allows the filter category to be selected.

Listing 16-13. Applying a Pipe in the productTable.component.html File in the src/app Folder

```

<div class="my-2">
  <label>Tax Rate:</label>
  <select class="form-select" [value]="taxRate || 0"
    (change)="taxRate=$any($event).target.value">
    <option value="0">None</option>
    <option value="10">10%</option>
    <option value="20">20%</option>
    <option value="50">50%</option>
  </select>
</div>

<div class="my-2">
  <label>Category Filter:</label>
  <select class="form-select" [(ngModel)]="categoryFilter">
    <option>Watersports</option>
    <option>Soccer</option>
    <option>Chess</option>
  </select>
</div>

```

```

</select>
</div>






```

To see the problem, use the select element to filter the products in the table so that only those in the Soccer category are shown. Then use the form elements to create a new product in that category. Clicking the Create button will add the product to the data model, but the new product won't be shown in the table, as illustrated in Figure 16-6.

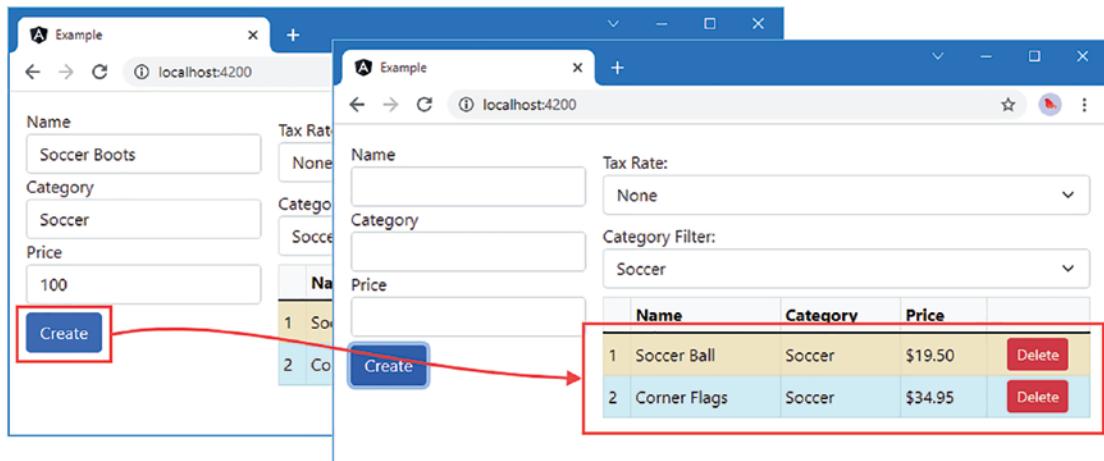


Figure 16-6. A problem caused by a pure pipe

The table isn't updated because, as far as Angular is concerned, none of the inputs to the filter pipe has changed. The component's `getProducts` method returns the same array object, and the `categoryFilter` property is still set to Soccer. The fact that there is a new object inside the array returned by the `getProducts` method isn't recognized by Angular.

The solution is to set the pipe's `pure` property to `false`, as shown in Listing 16-14.

Listing 16-14. Marking a Pipe as Impure in the `categoryFilter.pipe.ts` File in the `src/app` Folder

```
import { Pipe } from "@angular/core";
import { Product } from "./product.model";

@Pipe({
  name: "filter",
  pure: false
})
export class PaCategoryFilterPipe {

  transform(products: Product[] | undefined, category: string | undefined): Product[] {
    if (products == undefined) {
      return [];
    }
    return category == undefined ?
      products : products.filter(p => p.category == category);
  }
}
```

If you repeat the test, you will see that the new product is now correctly displayed in the table, as shown in Figure 16-7.

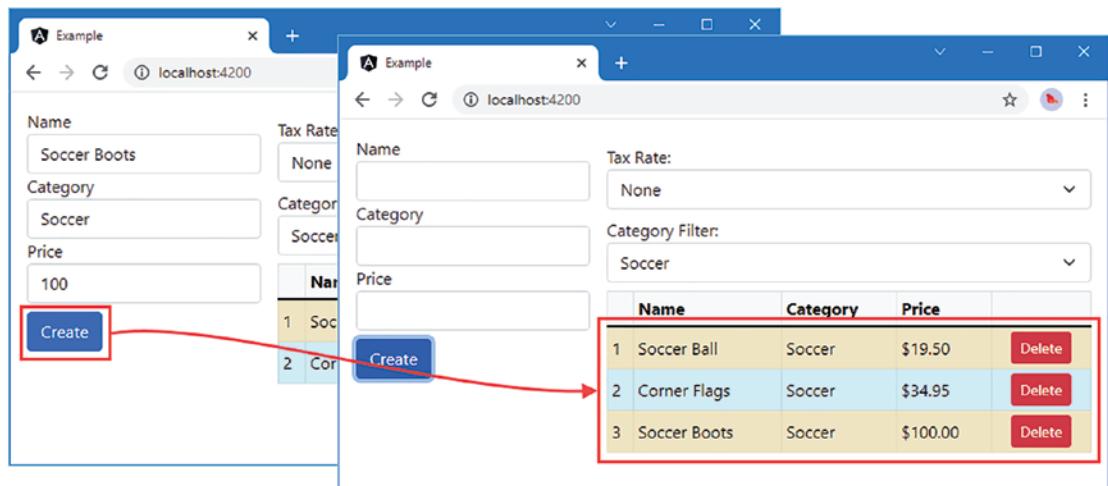


Figure 16-7. Using an impure pipe

Using the Built-in Pipes

Angular includes a set of built-in pipes that perform commonly required tasks. These pipes are described in Table 16-4 and demonstrated in the sections that follow.

Table 16-4. *The Built-in Pipes*

Name	Description
number	This pipe performs location-sensitive formatting of number values. See the “Formatting Numbers” section for details.
currency	This pipe performs location-sensitive formatting of currency amounts. See the “Formatting Currency Values” section for details.
percent	This pipe performs location-sensitive formatting of percentage values. See the “Formatting Percentages” section for details.
date	This pipe performs location-sensitive formatting of dates. See the “Formatting Dates” section for details.
uppercase	This pipe transforms all the characters in a string to uppercase. See the “Changing String Case” section for details.
lowercase	This pipe transforms all the characters in a string to lowercase. See the “Changing String Case” section for details.
titlecase	This pipe transforms all the characters in a string to title case. See the “Changing String Case” section for details.
json	This pipe transforms an object into a JSON string. See the “Serializing Data as JSON” section for details.
slice	This pipe selects items from an array or characters from a string, as described in the “Slicing Data Arrays” section.
keyvalue	This pipe transforms an object or map into a series of key-value pairs, as described in the “Formatting Key-Value Pairs” section.
i18nSelect	This pipe selects a text value to display for a set of values, as described in the “Selecting Values” section.
i18nPlural	This pipe selects a pluralized string for a value, as described in the “Pluralizing Values” section.
async	This pipe subscribes to an observable or a promise and displays the most recent value it produces.

Formatting Numbers

The `number` pipe formats `number` values using locale-sensitive rules. Listing 16-15 shows the use of the `number` pipe, along with the argument that specifies the formatting that will be used. I have removed the custom pipes and the associated select elements from the template.

Listing 16-15. Using the number Pipe in the productTable.component.html File in the src/app Folder

```
<!-- <div class="my-2">
    <label>Tax Rate:</label>
    <select class="form-select" [value]="taxRate || 0"
        (change)="taxRate=$any($event).target.value">
        <option value="0">None</option>
        <option value="10">10%</option>
        <option value="20">20%</option>
        <option value="50">50%</option>
    </select>
</div>

<div class="my-2">
    <label>Category Filter:</label>
    <select class="form-select" [(ngModel)]="categoryFilter">
        <option>Watersports</option>
        <option>Soccer</option>
        <option>Chess</option>
    </select>
</div> -->

<table class="table table-sm table-bordered table-striped">
    <thead class="table-light">
        <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    </thead>
    <tbody>
        <tr *paFor="let item of getProducts() | filter:categoryFilter;
            let i = index; let odd = odd;
            let even = even" [class.table-info]="odd" [class.table-warning]="even"
            class="align-middle">
            <td>{{i + 1}}</td>
            <td>{{item.name}}</td>
            <td>{{item.category}}</td>
            <td>{{item.price | number:"3.2-2" }}</td>
            <td class="text-center">
                <button class="btn btn-danger btn-sm"
                    (click)="deleteProduct(item.id)">
                    Delete
                </button>
            </td>
        </tr>
    </tbody>
</table>
```

The number pipe accepts a single argument that specifies the number of digits that are included in the formatted result. The argument is in the following format (note the period and hyphen that separate the values and that the entire argument is quoted as a string):

"<minIntegerDigits>.<minFractionDigits>-<maxFractionDigits>"

Table 16-5 describes each element of the formatting argument.

Table 16-5. The Elements of the number Pipe Argument

Name	Description
minIntegerDigits	This value specifies the minimum number of digits. The default value is 1.
minFractionDigits	This value specifies the minimum number of fractional digits. The default value is 0.
maxFractionDigits	This value specifies the maximum number of fractional digits. The default value is 3.

The argument used in the listing is "3.2-2", which specifies that at least three digits should be used to display the integer portion of the number and that two fractional digits should always be used. This produces the result shown in Figure 16-8.

	Name	Category	Price	
1	Kayak	Watersports	275.00	<button>Delete</button>
2	Lifejacket	Watersports	048.95	<button>Delete</button>
3	Soccer Ball	Soccer	019.50	<button>Delete</button>
4	Corner Flags	Soccer	034.95	<button>Delete</button>
5	Thinking Cap	Chess	016.00	<button>Delete</button>

Figure 16-8. Formatting number values

The number pipe is location-sensitive, which means that the same format argument will produce differently formatted results based on the user's locale setting. Angular applications default to the en-US locale by default and require other locales to be loaded explicitly, as shown in Listing 16-16.

Listing 16-16. Setting the Locale in the app.module.ts File in the src/app Folder

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
```

```

import { PaModel } from "./twoWay.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

registerLocaleData(localeFr);

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [{ provide: LOCALE_ID, useValue: "fr-FR" }],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

Setting the locale consists of importing the locale you require from the modules that contain each region's data and registering it by calling the `registerLocaleData` function, which is imported from the `@angular/common` module. In the listing, I have imported the `fr-FR` locale, which is for French as it is spoken in France. The final step is to configure the `providers` property, which I describe in Chapter 17, but the effect of the configuration in Listing 16-16 is to enable the `fr-FR` locale, which changes the formatting of the numerical values, as shown in Figure 16-9.

The screenshot shows a web application interface. On the left, there is a form with three input fields: 'Name' (empty), 'Category' (empty), and 'Price' (empty). Below the form is a blue 'Create' button. On the right is a table with five rows of data. The table has columns for 'Name', 'Category', and 'Price'. The 'Price' column contains values like '275,00', '048,95', '019,50', '034,95', and '016,00'. The last row ('Thinking Cap') has a red border around its 'Price' cell. Each row also has a 'Delete' button.

	Name	Category	Price	
1	Kayak	Watersports	275,00	Delete
2	Lifejacket	Watersports	048,95	Delete
3	Soccer Ball	Soccer	019,50	Delete
4	Corner Flags	Soccer	034,95	Delete
5	Thinking Cap	Chess	016,00	Delete

Figure 16-9. Locale-sensitive formatting

You can override the application's locale setting by specifying a locale as a configuration option for the pipe, like this:

```
...
<td>{{item.price | number:"3.2-2":"en-US" }}</td>
...
```

Formatting Currency Values

The currency pipe formats number values that represent monetary amounts. Listing 16-6 used this pipe to introduce the topic, and Listing 16-17 shows another application of the same pipe but with the addition of number format specifiers.

Listing 16-17. Using the currency Pipe in the productTable.component.html File in the src/app Folder

```
<table class="table table-sm table-bordered table-striped">
  <thead class="table-light">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of getProducts() | filter:categoryFilter;
      let i = index; let odd = odd;
      let even = even" [class.table-info]="odd" [class.table-warning]="even"
      class="align-middle">
      <td>{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price | currency:"USD":"symbol": "2.2-2" }}</td>
      <td class="text-center">
```

```

<button class="btn btn-danger btn-sm"
       (click)="deleteProduct(item.id)">
    Delete
</button>
</td>
</tr>
</tbody>
</table>

```

The currency pipe can be configured using four arguments, which are described in Table 16-6.

Table 16-6. The Types of Web Forms Code Nuggets

Name	Description
currencyCode	This string argument specifies the currency using an ISO 4217 code. The default value is USD if this argument is omitted. You can see a list of currency codes at http://en.wikipedia.org/wiki/ISO_4217 .
display	This string indicates whether the currency symbol or code should be displayed. The supported values are code (use the currency code), symbol (use the currency symbol), and symbol-narrow (which shows the concise form when a currency has narrow and wide symbols). You can also specify a string to use. The default value is symbol.
digitInfo	This string argument specifies the formatting for the number, using the same formatting instructions supported by the number pipe, as described in the “Formatting Numbers” section.
locale	This string argument specifies the locale for the currency. This defaults to the LOCALE_ID value, the configuration of which is shown in Listing 16-16.

The arguments specified in Listing 16-17 tell the pipe to use the U.S. dollar as the currency (which has the ISO code USD), to display the symbol rather than the code in the output, and to format the number so that it has at least two integer digits and exactly two fraction digits.

This pipe relies on the Internationalization API to get details of the currency—especially its symbol—but doesn’t select the currency automatically to reflect the user’s locale setting.

This means that the formatting of the number and the position of the currency symbol are affected by the application’s locale setting, regardless of the currency that has been specified by the pipe. The example application is still configured to use the fr-FR locale, which produces the results shown in Figure 16-10.

The screenshot shows a web application interface. On the left side, there are three input fields: 'Name' (empty), 'Category' (empty), and 'Price' (empty). Below these fields is a blue 'Create' button. On the right side, there is a table with five rows of data. The table has columns for 'Name', 'Category', 'Price', and a red 'Delete' button. The data in the table is as follows:

	Name	Category	Price	
1	Kayak	Watersports	275,00 \$US	<button>Delete</button>
2	Lifejacket	Watersports	48,95 \$US	<button>Delete</button>
3	Soccer Ball	Soccer	19,50 \$US	<button>Delete</button>
4	Corner Flags	Soccer	34,95 \$US	<button>Delete</button>
5	Thinking Cap	Chess	16,00 \$US	<button>Delete</button>

Figure 16-10. Location-sensitive currency formatting

To revert to the default locale, Listing 16-18 removes the fr-FR setting from the application's root module.

Listing 16-18. Removing the locale Setting in the app.module.ts File in the src/app Folder

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

registerLocaleData(localeFr);
```

```

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  //providers: [{ provide: LOCALE_ID, useValue: "fr-FR" }],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

Figure 16-11 shows the result.

	Name	Category	Price	
1	Kayak	Watersports	\$275.00	<button>Delete</button>
2	Lifejacket	Watersports	\$48.95	<button>Delete</button>
3	Soccer Ball	Soccer	\$19.50	<button>Delete</button>
4	Corner Flags	Soccer	\$34.95	<button>Delete</button>
5	Thinking Cap	Chess	\$16.00	<button>Delete</button>

Figure 16-11. Formatting currency values

Formatting Percentages

The percent pipe formats number values as percentages, where values between 0 and 1 are formatted to represent 0 to 100 percent. This pipe has optional arguments that are used to specify the number formatting options, using the same format as the number pipe, and override the default locale. Listing 16-19 re-introduces the custom sales tax filter and populates the associated select element with option elements whose content is formatted with the percent filter.

Listing 16-19. Formatting Percentages in the productTable.component.html File in the src/app Folder

```
<div class="my-2">
  <label>Tax Rate:</label>
  <select class="form-select" [value]="taxRate || 0"
    (change)="taxRate=$any($event).target.value">
    <option value="0">None</option>
    <option value="10">{{ 0.1 | percent }}</option>
    <option value="20">{{ 0.2 | percent }}</option>
    <option value="50">{{ 0.5 | percent }}</option>
    <option value="150">{{ 1.5 | percent }}</option>
  </select>
</div>

<table class="table table-sm table-bordered table-striped">
  <thead class="table-light">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of getProducts() | filter:categoryFilter;
      let i = index; let odd = odd;
      let even = even" [class.table-info]="odd" [class.table-warning]="even"
      class="align-middle">
      <td>{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price | addTax:(taxRate ?? 0)
        | currency:"USD": "symbol": "2.2-2" }}</td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm"
          (click)="deleteProduct(item.id)">
          Delete
        </button>
      </td>
    </tr>
  </tbody>
</table>
```

Values that are greater than 1 are formatted into percentages greater than 100 percent. You can see this in the last item shown in Figure 16-12, where the value 1.5 produces a formatted value of 150 percent.

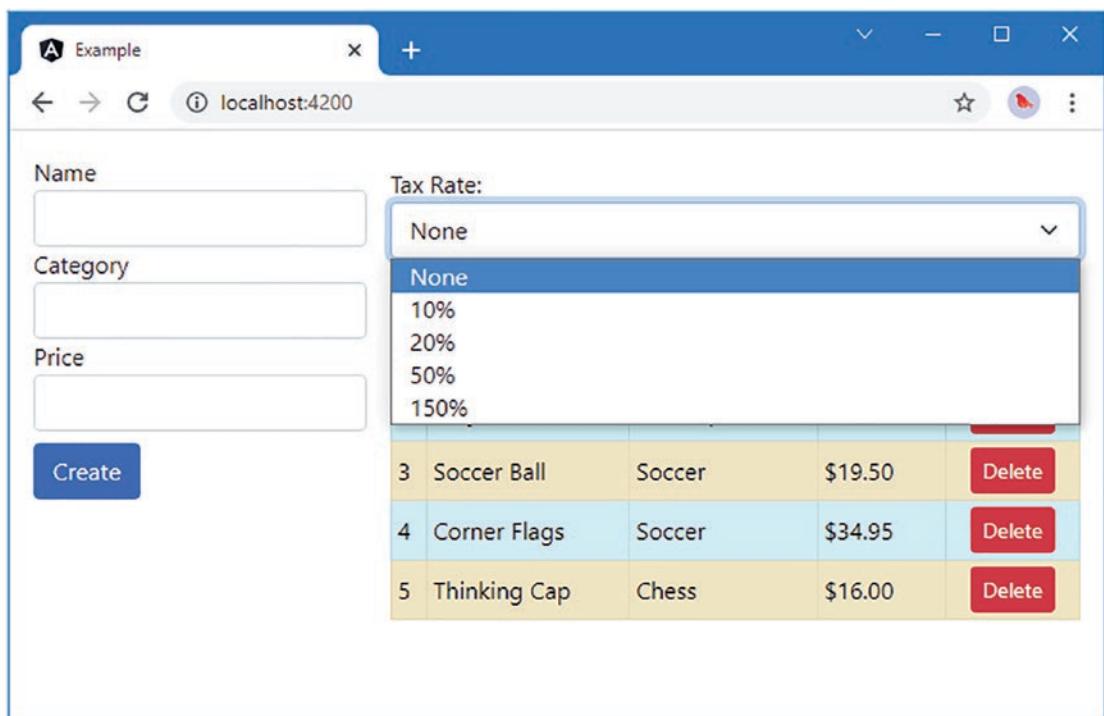


Figure 16-12. Formatting percentage values

The formatting of percentage values is location-sensitive, although the differences between locales can be subtle. As an example, while the en-US locale produces a result such as 10 percent, with the numerals and the percent sign next to one another, many locales, including fr-FR, will produce a result such as 10 %, with a space between the numerals and the percent sign.

Formatting Dates

The date pipe performs location-sensitive formatting of dates. Dates can be expressed using JavaScript Date objects, as a number value representing milliseconds since the beginning of 1970 or as a well-formatted string. Listing 16-20 adds three properties to the ProductTableComponent class, each of which encodes a date in one of the formats supported by the date pipe.

Listing 16-20. Defining Dates in the productTable.component.ts File in the src/app Folder

```
import { Component, Input, QueryList, ViewChildren } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {
```

```

@Input("model")
dataModel: Model | undefined;

getProduct(key: number): Product | undefined {
    return this.dataModel?.getProduct(key);
}

getProducts(): Product[] | undefined {
    return this.dataModel?.getProducts();
}

deleteProduct(key: number) {
    this.dataModel?.deleteProduct(key);
}

taxRate: number = 0;
categoryFilter: string | undefined;
itemCount: number = 3;

dateObject: Date = new Date(2020, 1, 20);
dateString: string = "2020-02-20T00:00:00.000Z";
dateNumber: number = 1582156800000;
}

```

All three properties describe the same date, which is February 20, 2020. No time has been specified, which means that these values will represent midnight, with no time specified. In Listing 16-21, I have used the date pipe to format all three properties.

Listing 16-21. Formatting Dates in the productTable.component.html File in the src/app Folder

```

<div class="bg-info p-2 text-white">
    <div>Date formatted from object: {{ dateObject | date }}</div>
    <div>Date formatted from string: {{ dateString | date }}</div>
    <div>Date formatted from number: {{ dateNumber | date }}</div>
</div>

<table class="table table-sm table-bordered table-striped">
    <thead class="table-light">
        <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    </thead>
    <tbody>
        <tr *paFor="let item of getProducts() | filter:categoryFilter;
            let i = index; let odd = odd;
            let even = even" [class.table-info]="odd" [class.table-warning]="even"
            class="align-middle">
            <td>{{i + 1}}</td>
            <td>{{item.name}}</td>
            <td>{{item.category}}</td>
            <td>{{item.price | addTax:(taxRate ?? 0)
                | currency:"USD":"symbol":"2.2-2" }}</td>
            <td class="text-center">
                <button class="btn btn-danger btn-sm">

```

```

        (click)="deleteProduct(item.id)">
      Delete
    </button>
  </td>
</tr>
</tbody>
</table>

```

The pipe works out which data type it is working with, parses the value to get a date, and then formats it, as shown in Figure 16-13.

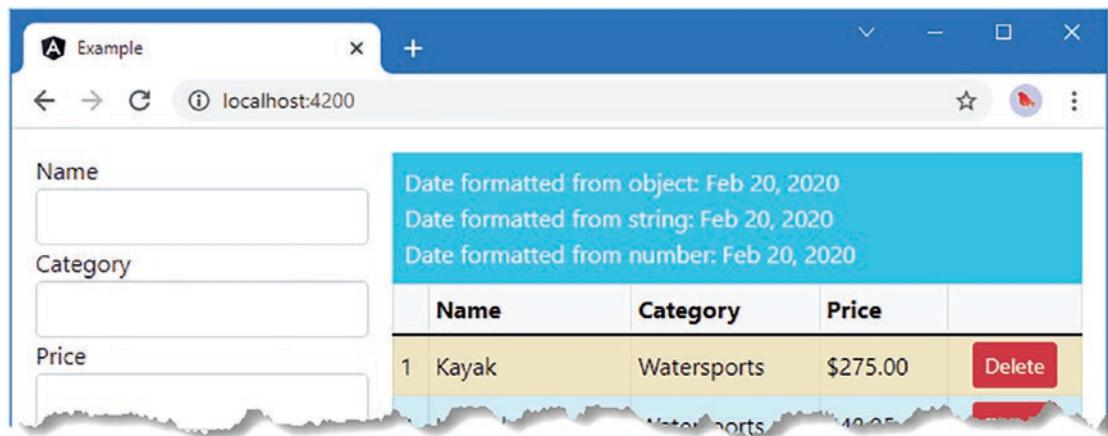


Figure 16-13. Formatting dates

If you are in a time zone that is to the left of GMT, then you will see Feb 19, 2020, for two of the dates. The first date is expressed relative to the application's time zone, but the others are expressed in the UTC time zone, which means that the dates will be adjusted, as shown in Figure 16-14.

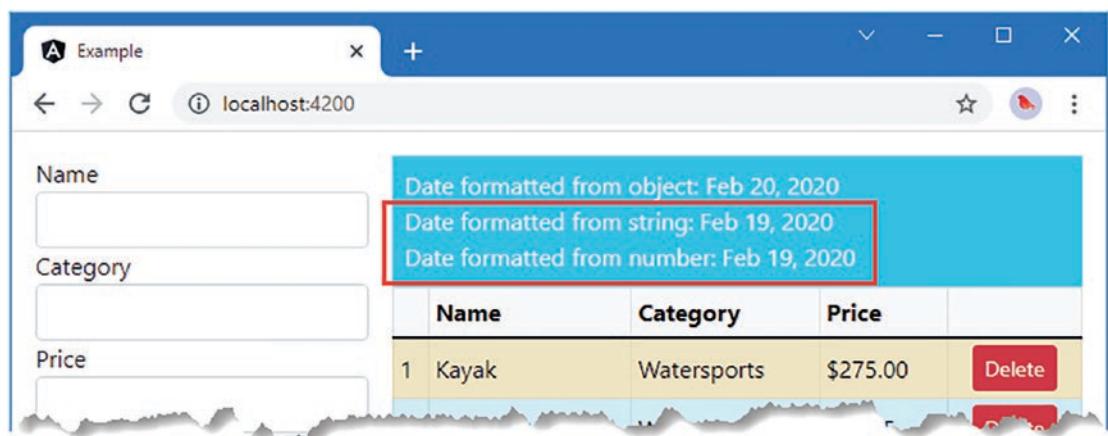


Figure 16-14. The effect of a different time zone

The date pipe accepts an argument that specifies the date format that should be used. Individual date components can be selected for the output using the symbols described in Table 16-7. A complete set of supported symbols can be found in the Angular API documentation at <https://angular.io/api/common/DatePipe>.

Table 16-7. Useful Date Pipe Format Symbols

Name	Description
y, yy, yyyy	These symbols select the year.
M, MMM, MMMM	These symbols select the month.
d, dd	These symbols select the day (as a number).
E, EE, EEE, EEEE, EEEEE	These symbols select the day (as a name).
h, hh, H, HH	These symbols select the hour in 12- and 24-hour forms.
m, mm	These symbols select the minutes.
s, ss	These symbols select the seconds.
Z	This symbol selects the time zone.

The symbols in Table 16-7 provide access to the date components in differing levels of brevity so that M will return 2 if the month is February, MM will return 02, MMM will return Feb, and MMMM will return February, assuming that you are using the en-US locale. The date pipe also supports predefined date formats for commonly used combinations, the most useful of which are described in Table 16-8.

Table 16-8. Useful Predefined date Pipe Formats

Name	Description
short	This format is equivalent to the component string M/d/yy, h:mm a. It presents the date in a concise format, including the time component.
medium	This format is equivalent to the component string MMM d, y, h:mm:ss a. It presents the date as a more expansive format, including the time component.
shortDate	This format is equivalent to the component string M/d/yy. It presents the date in a concise format and excludes the time component.
mediumDate	This format is equivalent to the component string MMM d, y. It presents the date in a more expansive format and excludes the time component.
longDate	This format is equivalent to the component string MMMM d, y. It presents the date and excludes the time component.
fullDate	This format is equivalent to the component string EEEE, MMMM d, y. It presents the date fully and excludes the date format.
shortTime	This format is equivalent to the component string h:mm a.
mediumTime	This format is equivalent to the component string h:mm:ss a.

The date pipe also accepts arguments that specify a time zone and a locale. Listing 16-22 shows the use of the predefined formats as arguments to the date pipe, rendering the same date in different ways and with different locale settings.

Tip The time zone argument has to be specified in order to set the locale. Use the empty string ("") as the time zone if you want to use the application's default time zone.

Listing 16-22. Formatting Dates in the productTable.component.html File in the src/app Folder

```
<div class="bg-info p-2 text-white">
    <div>Date formatted from object: {{ dateObject | date:"shortDate" }}</div>
    <div>Date formatted from string: {{ dateString | date:"mediumDate" }}</div>
    <div>Date formatted from number: {{ dateNumber | date:"longDate" }}</div>
</div>

<div class="bg-info p-2 text-white">
    <div>
        Date formatted from object: {{ dateObject | date:"shortDate":"UTC":"fr-FR" }}
    </div>
    <div>
        Date formatted from string: {{ dateString | date:"mediumDate":"UTC":"fr-FR" }}
    </div>
    <div>
        Date formatted from number: {{ dateNumber | date:"longDate":"UTC":"fr-FR" }}
    </div>
</div>

<table class="table table-sm table-bordered table-striped">
    <thead class="table-light">
        <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    </thead>
    <tbody>
        <tr *paFor="let item of getProducts() | filter:categoryFilter;
            let i = index; let odd = odd;
            let even = even" [class.table-info]="odd" [class.table-warning]="even"
            class="align-middle">
            <td>{{i + 1}}</td>
            <td>{{item.name}}</td>
            <td>{{item.category}}</td>
            <td>{{item.price | addTax:(taxRate ?? 0)
                | currency:"USD": "symbol": "2.2-2" }}</td>
            <td class="text-center">
                <button class="btn btn-danger btn-sm"
                    (click)="deleteProduct(item.id)">
                    Delete
                </button>
            </td>
        </tr>
    </tbody>
</table>
```

Formatting arguments are specified as literal strings. Take care to capitalize the format string correctly because `shortDate` will be interpreted as one of the predefined formats from Table 16-8, but `shortdate` (with a lowercase letter `d`) will be interpreted as a series of characters from Table 16-7 and produce nonsensical output.

Caution Date parsing/formatting is a complex and time-consuming process. As a consequence, the `pure` property for the date pipe is `true`; as a result, changes to individual components of a `Date` object won't trigger an update. If you need to reflect changes in the way that a date is displayed, then you must change the reference to the `Date` object that the binding containing the date pipe refers to.

Figure 16-15 shows the formatted dates, in the `en-US` and `fr-FR` locales.

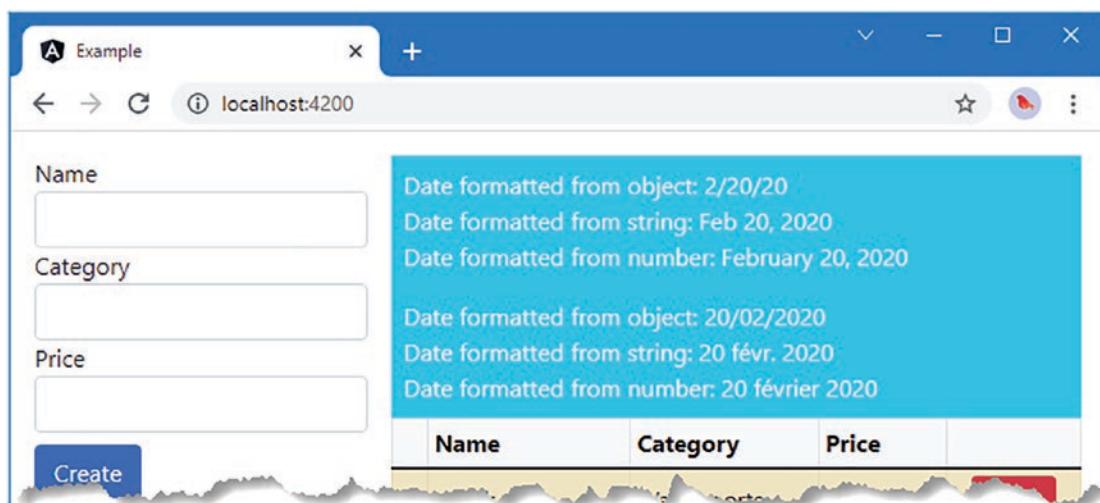


Figure 16-15. Location-sensitive date formatting

UNDERSTANDING THE IMPACT OF LAZY LOCALIZATION

Localizing a product takes time, effort, and resources, and it needs to be done by someone who understands the linguistic, cultural, and monetary conventions of the target country or region. If you don't localize properly, then the result can be worse than not localizing at all.

It is for this reason that I don't describe localization features in detail in this book—or any of my books. Describing features outside of the context in which they will be used feels like setting up readers for a self-inflicted disaster. At least if a product isn't localized, the user knows where they stand and doesn't have to try to figure out whether you just forgot to change the currency code or whether those prices are really in U.S. dollars. (This is an issue that I see all the time living in the United Kingdom.)

You *should* localize your products. Your users *should* be able to do business or perform other operations in a way that makes sense to them. But you *must* take it seriously and allocate the time and effort required to do it properly.

Changing String Case

The uppercase, lowercase, and titlecase pipes convert all the characters in a string to uppercase or lowercase, respectively. Listing 16-23 shows the first two pipes applied to cells in the product table. This listing also removes the dates used in the previous section.

Listing 16-23. Changing Character Case in the productTable.component.html File in the src/app Folder

```
<!-- <div class="bg-info p-2 text-white">
    <div>Date formatted from object: {{ dateObject | date:"shortDate" }}</div>
    <div>Date formatted from string: {{ dateString | date:"mediumDate" }}</div>
    <div>Date formatted from number: {{ dateNumber | date:"longDate" }}</div>
</div>

<div class="bg-info p-2 text-white">
    <div>Date formatted from object: {{ dateObject | date:"shortDate": "UTC": "fr-FR" }}</div>
    <div>Date formatted from string: {{ dateString | date:"mediumDate": "UTC": "fr-FR" }}</div>
    <div>Date formatted from number: {{ dateNumber | date:"longDate": "": "fr-FR" }}</div>
</div> -->

<table class="table table-sm table-bordered table-striped">
    <thead class="table-light">
        <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
    </thead>
    <tbody>
        <tr *paFor="let item of getProducts() | filter:categoryFilter;
            let i = index; let odd = odd;
            let even = even" [class.table-info] = "odd" [class.table-warning] = "even"
            class="align-middle">
            <td>{{i + 1}}</td>
            <td>{{item.name | uppercase }}</td>
            <td>{{item.category | lowercase }}</td>
            <td>
                {{item.price | addTax:(taxRate ?? 0)
                    | currency:"USD": "symbol": "2.2-2" }}
            </td>
            <td class="text-center">
                <button class="btn btn-danger btn-sm"
                    (click)="deleteProduct(item.id)">
                    Delete
                </button>
            </td>
        </tr>
    </tbody>
</table>
```

These pipes use the standard JavaScript string methods `toUpperCase` and `toLowerCase`, which are not sensitive to locale settings, as shown in Figure 16-16.

	Name	Category	Price	
1	KAYAK	watersports	\$275.00	<button>Delete</button>
2	LIFEJACKET	watersports	\$48.95	<button>Delete</button>
3	SOCcer BALL	soccer	\$19.50	<button>Delete</button>
4	CORNER FLAGS	soccer	\$34.95	<button>Delete</button>
5	THINKING CAP	chess	\$16.00	<button>Delete</button>

Figure 16-16. Changing character case

The `titlecase` pipe capitalizes the first character of each word and uses lowercase for the remaining characters. Listing 16-24 applies the `titlecase` pipe to the table cells.

Listing 16-24. Applying the Pipe in the `productTable.component.html` File in the `src/app` Folder

```
<table class="table table-sm table-bordered table-striped">
  <thead class="table-light">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of getProducts() | filter:categoryFilter;
        let i = index; let odd = odd;
        let even = even" [class.table-info]="odd" [class.table-warning]="even"
        class="align-middle">
      <td>{{i + 1}}</td>
      <td>{{item.name | titlecase }}</td>
      <td>{{item.category | lowercase }}</td>
      <td>
        {{item.price | addTax:(taxRate ?? 0)
          | currency:"USD": "symbol": "2.2-2" }}
      </td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm"
              (click)="deleteProduct(item.id)">
          Delete
        </button>
      </td>
    </tr>
  </tbody>
</table>
```

Figure 16-17 shows the effect of the pipe.

	Name	Category	Price	
1	Kayak	watersports	\$275.00	Delete
2	Lifejacket	watersports	\$48.95	Delete
3	Soccer Ball	soccer	\$19.50	Delete
4	Corner Flags	soccer	\$34.95	Delete
5	Thinking Cap	chess	\$16.00	Delete

Figure 16-17. Using the `titlecase` pipe

Serializing Data as JSON

The `json` pipe creates a JSON representation of a data value. No arguments are accepted by this pipe, which uses the browser's `JSON.stringify` method to create the JSON string. Listing 16-25 applies this pipe to create a JSON representation of the objects in the data model.

Listing 16-25. Creating a JSON String in the `productTable.component.html` File in the `src/app` Folder

```
<div class="bg-info p-2 text-white">
  <div>{{ getProducts() | json }}</div>
</div>

<table class="table table-sm table-bordered table-striped">
  <thead class="table-light">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of getProducts() | filter:categoryFilter;
      let i = index; let odd = odd;
      let even = even" [class.table-info]="odd" [class.table-warning]="even"
      class="align-middle">
      <td>{{i + 1}}</td>
      <td>{{item.name | titlecase }}</td>
      <td>{{item.category | lowercase }}</td>
      <td>{{item.price | addTax:(taxRate ?? 0)
        | currency:"USD":"symbol":"2.2-2" }}</td>
      <td>
```

```

<td class="text-center">
    <button class="btn btn-danger btn-sm"
        (click)="deleteProduct(item.id)">
        Delete
    </button>
</td>
</tr>
</tbody>
</table>

```

This pipe is useful during debugging, and its decorator's pure property is `false` so that any change in the application will cause the pipe's `transform` method to be invoked, ensuring that even collection-level changes are shown. Figure 16-18 shows the JSON generated from the objects in the example application's data model.

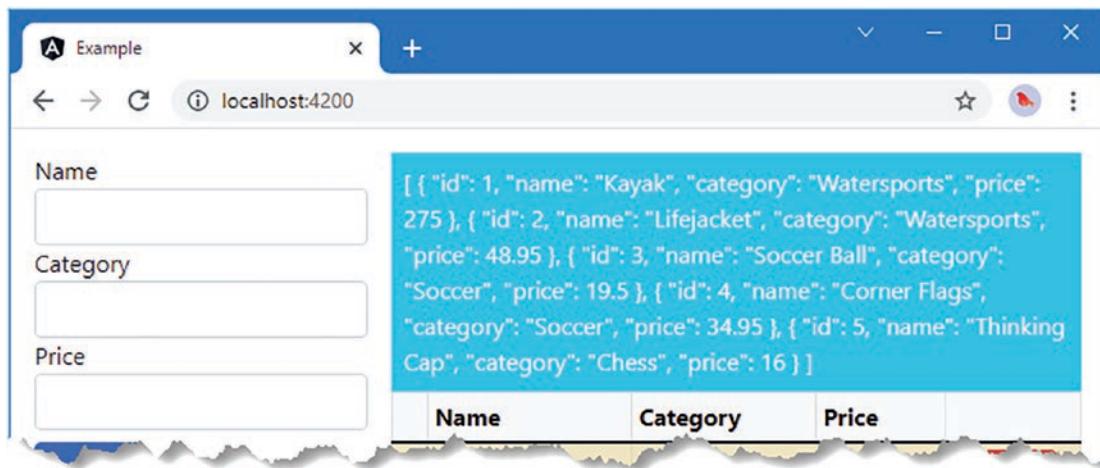


Figure 16-18. Generating JSON strings for debugging

Slicing Data Arrays

The `slice` pipe operates on an array or string and returns a subset of the elements or characters it contains. This is an impure pipe, which means it will reflect any changes that occur within the data object it is operating on but also means that the slice operation will be performed after any change in the application, even if that change was not related to the source data.

The objects or characters selected by the `slice` pipe are specified using two arguments, which are described in Table 16-9.

Table 16-9. The Slice Pipe Arguments

Name	Description
start	This argument must be specified. If the value is positive, the start index for items to be included in the result counts from the first position in the array. If the value is negative, then the pipe counts back from the end of the array.
end	This optional argument is used to specify how many items from the start index should be included in the result. If this value is omitted, all the items after the start index (or before in the case of negative values) will be included.

Listing 16-26 demonstrates the use of the slice pipe in combination with a select element that specifies how many items should be displayed in the product table.

Listing 16-26. Using the slice Pipe in the productTable.component.html File in the src/app Folder

```
<div class="form-group my-2">
  <label>Number of items:</label>
  <select class="form-select" [value]="itemCount ?? 1"
    (change)="itemCount=$any($event).target.value">
    <option *ngFor="let item of getProducts(); let i = index" [value]="i + 1"
      [selected]="(i + 1) === itemCount">
      {{i + 1}}
    </option>
  </select>
</div>

<table class="table table-sm table-bordered table-striped">
  <thead class="table-light">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of getProducts() | slice:0:(itemCount ?? 1);
        let i = index; let odd = odd;
        let even = even" [class.table-info]="'odd'" [class.table-warning]="'even'"
        class="align-middle">
      <td>{{i + 1}}</td>
      <td>{{item.name | titlecase }}</td>
      <td>{{item.category | lowercase }}</td>
      <td>{{item.price | addTax:(taxRate ?? 0)
          | currency:"USD": "symbol": "2.2-2" }}</td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm"
          (click)="deleteProduct(item.id)">
          Delete
        </button>
      </td>
    </tr>
  </tbody>
</table>
```

The select element is populated with option elements created with the ngFor directive. This directive doesn't directly support iterating a specific number of times, so I have used the index variable to generate the values that are required. The select element sets a property called itemCount, which is used as the second argument of the slice pipe, like this:

```
...
<tr *paFor="let item of getProducts() | slice:0:(itemCount ?? 1);
  let i = index; let odd = odd;
  let even = even" [class.table-info]="odd" [class.table-warning]="even"
  class="align-middle">
...

```

The effect is that changing the value displayed by the select element changes the number of items displayed in the product table, as shown in Figure 16-19.

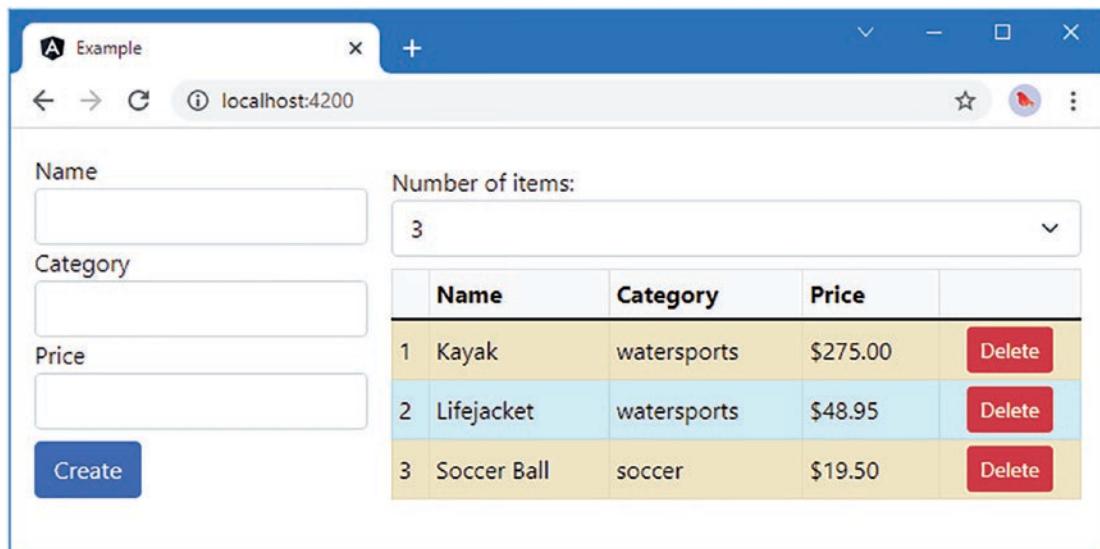


Figure 16-19. Using the slice pipe

Formatting Key-Value Pairs

The keyvalue pipe operates on an object or a map and returns a sequence of key-value pairs. Each object in the sequence is represented as an object with key and value properties, and Listing 16-27 replaces the contents of the productTable.component.html file to demonstrate the use of the pipe to enumerate the contents of the array returned by the getProducts method.

Listing 16-27. Using the keyvalue Pipe in the productTable.component.html File in the src/app Folder

```
<table class="table table-sm table-bordered table-striped">
  <thead><tr><th>Key</th><th>Value</th></tr></thead>
  <tbody>
    <tr *paFor="let item of getProducts() | keyvalue">
      <td>{{ item.key }}</td>
```

```

        <td>{{ item.value | json }}</td>
    </tr>
</tbody>
</table>

```

When used on an array, the keys are the array indexes, and the values are the objects in the array. The objects in the array are formatted using the `json` filter, producing the results shown in Figure 16-20.

Key	Value
0	{ "id": 1, "name": "Kayak", "category": "Watersports", "price": 275 }
1	{ "id": 2, "name": "Lifejacket", "category": "Watersports", "price": 48.95 }
2	{ "id": 3, "name": "Soccer Ball", "category": "Soccer", "price": 19.5 }
3	{ "id": 4, "name": "Corner Flags", "category": "Soccer", "price": 34.95 }
4	{ "id": 5, "name": "Thinking Cap", "category": "Chess", "price": 16 }

Figure 16-20. Using the `keyvalue` pipe

Selecting Values

The `i18nSelect` pipe selects a string based on a value, allowing context-sensitive values to be displayed to the user. The mapping between values and strings is defined as a simple map, as shown in Listing 16-28.

Listing 16-28. Mapping Values to Strings in the `productTable.component.ts` File in the `src/app` Folder

```

import { Component, Input, QueryList, ViewChildren } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {

  @Input("model")
  dataModel: Model | undefined;
}

```

```

getProduct(key: number): Product | undefined {
    return this.dataModel?.getProduct(key);
}

getProducts(): Product[] | undefined {
    return this.dataModel?.getProducts();
}

deleteProduct(key: number) {
    this.dataModel?.deleteProduct(key);
}

taxRate: number = 0;
categoryFilter: string | undefined;
itemCount: number = 3;

// dateObject: Date = new Date(2020, 1, 20);
// dateString: string = "2020-02-20T00:00:00.000Z";
// dateNumber: number = 1582156800000;

selectMap = {
    "Watersports": "stay dry",
    "Soccer": "score goals",
    "other": "have fun"
}
}

```

The other mapping is used as a fallback when there is no match with the other values. In Listing 16-29, I have applied the pipe to select a message to display to the user.

Listing 16-29. Using the Pipe in the productTable.component.html File in the src/app Folder

```

<table class="table table-sm table-bordered table-striped">
    <thead><tr><th>Name</th><th>Category</th><th>Message</th></tr></thead>
    <tbody>
        <tr *paFor="let item of getProducts()">
            <td>{{ item.name }}</td>
            <td>{{ item.category }}</td>
            <td>Helps you {{ item.category | i18nSelect:selectMap }} </td>
        </tr>
    </tbody>
</table>

```

The pipe is provided with the map as an argument and produces the response shown in Figure 16-21.

Name	Category	Message
Kayak	Watersports	Helps you stay dry
Lifejacket	Watersports	Helps you stay dry
Soccer Ball	Soccer	Helps you score goals
Corner Flags	Soccer	Helps you score goals
Thinking Cap	Chess	Helps you have fun

Figure 16-21. Selecting values using the `i18nSelect` pipe

Pluralizing Values

The `i18nPlural` pipe is used to select an expression that describes a numeric value. The mapping between values and expressions is expressed as a simple map, as shown in Listing 16-30.

Listing 16-30. Mapping Numbers to Strings in the `productTable.component.ts` File in the `src/app` Folder

```
import { Component, Input, QueryList, ViewChildren } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {

  // ...other statements omitted for brevity...

  selectMap = {
    "Watersports": "stay dry",
    "Soccer": "score goals",
    "other": "have fun"
  }

  numberMap = {
    "=1": "one product",
    "=2": "two products",
    "=3": "three products",
    "=4": "four products",
    "=5": "five products",
    "=6": "six products",
    "=7": "seven products",
    "=8": "eight products",
    "=9": "nine products",
    "=10": "ten products"
  }
}
```

```

    "=2": "two products",
    "other": "# products"
}
}

```

Each mapping is expressed with an equals sign followed by the number. The other value is a fallback, and the result it produces can refer to the number value using the # placeholder character. Listing 16-31 shows the results that can be produced using the example mappings.

Listing 16-31. Using the Pipe in the productTable.component.html File in the src/app Folder

```


| Name            | Category            | Message                                              |
|-----------------|---------------------|------------------------------------------------------|
| {{ item.name }} | {{ item.category }} | Helps you {{ item.category   i18nSelect:selectMap }} |



There are {{ 1 | i18nPlural:numberMap }}



There are {{ 2 | i18nPlural:numberMap }}



There are {{ 100 | i18nPlural:numberMap }}


```

The mapping is specified as the argument to the pipe, and the values in Listing 16-31 produce the result shown in Figure 16-22.

The screenshot shows a Microsoft Edge browser window. At the top, there are standard window controls: a plus sign for new tabs, a downward arrow, a minus sign, a square, and an X. To the right of these are icons for a star, a red bird, and three dots. Below the header is a table with five rows. The columns are labeled 'Name', 'Category', and 'Message'. The data is as follows:

Name	Category	Message
Kayak	Watersports	Helps you stay dry
Lifejacket	Watersports	Helps you stay dry
Soccer Ball	Soccer	Helps you score goals
Corner Flags	Soccer	Helps you score goals
Thinking Cap	Chess	Helps you have fun

Below the table, there is a large blue rectangular area containing three messages from an i18nPlural pipe:

- There are one product.
- There are two products.
- There are 100 products.

Figure 16-22. Selecting values using the `i18nPlural` pipe

Using the Async Pipe

Angular includes the `async` pipe, which can be used to consume Observable objects directly in a view, selecting the last object received from the event sequence. This is an impure pipe because its changes are driven from outside of the view in which it is used, meaning that its `transform` method will be called often, even if a new event has not been received from the Observable.

You can see this pipe used in later chapters to receive events from observables provided by the Angular API, but for this chapter, I am going to generate test events. Listing 16-32 adds a `Subject<number>` property to the `ProductTableComponent` class and uses it to generate a sequence of events.

Listing 16-32. Adding a Subject in the `productTable.component.ts` File in the `src/app` Folder

```
import { Component, Input, QueryList, ViewChildren } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { Subject } from "rxjs";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
```

```

export class ProductTableComponent {
    @Input("model")
    dataModel: Model | undefined;

    getProduct(key: number): Product | undefined {
        return this.dataModel?.getProduct(key);
    }

    getProducts(): Product[] | undefined {
        return this.dataModel?.getProducts();
    }

    deleteProduct(key: number) {
        this.dataModel?.deleteProduct(key);
    }

    taxRate: number = 0;
    categoryFilter: string | undefined;
    itemCount: number = 3;

    selectMap = {
        "Watersports": "stay dry",
        "Soccer": "score goals",
        "other": "have fun"
    }

    numberMap = {
        "=1": "one product",
        "=2": "two products",
        "other": "# products"
    }
}

numbers: Subject<number> = new Subject<number>();

ngOnInit() {
    let counter = 100;
    setInterval(() => {
        this.numbers.next(counter += 10)
    }, 1000);
}
}

```

Listing 16-33 applies the `async` pipe to display the values received from the observable.

Listing 16-33. Using the Async Pipe in the `productTable.component.html` File in the `src/app` Folder

```

<table class="table table-sm table-bordered table-striped">
    <thead>    <tr><th>Name</th><th>Category</th><th>Message</th></tr></thead>
    <tbody>
        <tr *paFor="let item of getProducts()">
            <td>{{ item.name }}</td>

```

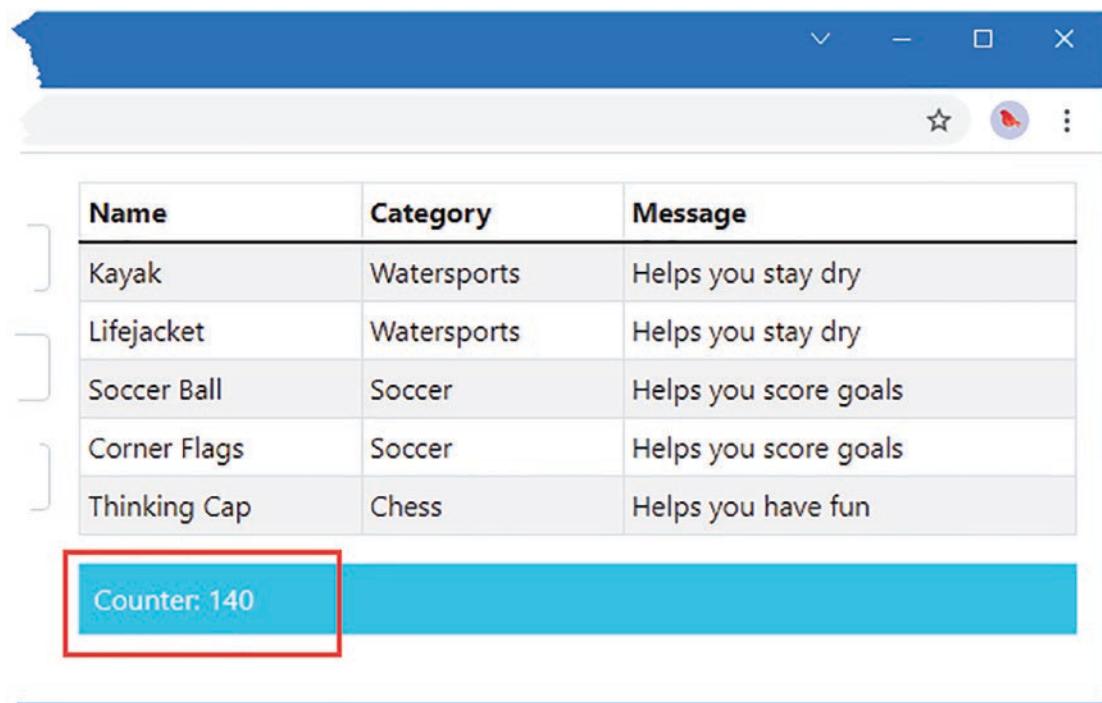
```

<td>{{ item.category }}</td>
<td>Helps you {{ item.category | i18nSelect:selectMap }} </td>
</tr>
</tbody>
</table>

<div class="bg-info text-white p-2">
  <div> Counter: {{ numbers | async }} </div>
</div>

```

The string interpolation binding expression gets the `numbers` property from the component and passes it to the `async` pipe, which keeps track of the most recent event that has been received, as shown in Figure 16-23.



A screenshot of a web browser window displaying a table. The table has three columns: Name, Category, and Message. The data is as follows:

Name	Category	Message
Kayak	Watersports	Helps you stay dry
Lifejacket	Watersports	Helps you stay dry
Soccer Ball	Soccer	Helps you score goals
Corner Flags	Soccer	Helps you score goals
Thinking Cap	Chess	Helps you have fun

Below the table is a blue footer bar containing the text "Counter: 140". This text is highlighted with a red rectangular box.

Figure 16-23. Using the `async` pipe

The `async` pipe can be used with other pipes, such as the currency pipe shown in Listing 16-34.

Listing 16-34. Combining Pipes in the `productTable.component.html` File in the `src/app` Folder

```

<table class="table table-sm table-bordered table-striped">
  <thead>    <tr><th>Name</th><th>Category</th><th>Message</th></tr></thead>
  <tbody>
    <tr *paFor="let item of getProducts()">
      <td>{{ item.name }}</td>
      <td>{{ item.category }}</td>

```

```
<td>Helps you {{ item.category | i18nSelect:selectMap }} </td>
</tr>
</tbody>
</table>

<div class="bg-info text-white p-2">
  <div> Counter: {{ numbers | async | currency:"USD":symbol:"2.2-2" }} </div>
</div>
```

As each event is received, it is passed from the `async` pipe to the `currency` pipe, producing the result shown in Figure 16-24.

The screenshot shows a web browser window with a blue header bar. Below the header is a toolbar with icons for star, refresh, and more. The main content area displays a table with three columns: Name, Category, and Message. The table has five rows of data. At the bottom of the table, there is a blue horizontal bar containing the text "Counter: \$120.00". This entire "Counter" section is highlighted with a red rectangular border.

Name	Category	Message
Kayak	Watersports	Helps you stay dry
Lifejacket	Watersports	Helps you stay dry
Soccer Ball	Soccer	Helps you score goals
Corner Flags	Soccer	Helps you score goals
Thinking Cap	Chess	Helps you have fun

Figure 16-24. Combining pipes

Summary

In this chapter, I introduced pipes and explained how they are used to transform data values so they can be presented to the user in the template. I demonstrated the process for creating custom pipes, explained how some pipes are pure and others are not, and demonstrated the built-in pipes that Angular provides for handling common tasks. In the next chapter, I introduce services, which can be used to simplify the design of Angular applications and allow building blocks to easily collaborate.

CHAPTER 17



Using Services

Services are objects that provide common functionality to support other building blocks in an application, such as directives, components, and pipes. What's important about services is the way that they are used, which is through a process called *dependency injection*. Using services can increase the flexibility and scalability of an Angular application, but dependency injection can be a difficult topic to understand. To that end, I start this chapter slowly and explain the problems that services and dependency injection can be used to solve, how dependency injection works, and why you should consider using services in your own projects. In Chapter 18, I introduce some more advanced features that Angular provides for service. Table 17-1 puts services in context.

Table 17-1. Putting Services in Context

Question	Answer
What are they?	Services are objects that define the functionality required by other building blocks such as components or directives. What separates services from regular objects is that they are provided to building blocks by an external provider, rather than being created directly using the <code>new</code> keyword or received by an input property.
Why are they useful?	Services simplify the structure of applications, make it easier to move or reuse functionality, and make it easier to isolate building blocks for effective unit testing.
How are they used?	Classes declare dependencies on services using constructor parameters, which are then resolved using the set of services for which the application has been configured. Services are classes to which the <code>@Injectable</code> decorator has been applied.
Are there any pitfalls or limitations?	Dependency injection is a contentious topic, and not all developers like using it. If you don't perform unit tests or if your applications are relatively simple, the extra work required to implement dependency injection is unlikely to pay any long-term dividends.
Are there any alternatives?	Services and dependency injection are hard to avoid because Angular uses them to provide access to built-in functionality. But you are not required to define services for your own custom functionality if that is your preference.

Table 17-2 summarizes the chapter.

Table 17-2. Chapter Summary

Problem	Solution	Listing
Avoiding the need to distribute shared objects manually	Use services	1-14, 21-28
Declaring a dependency on a service	Add a constructor parameter with the type of the service you require	15-20

Preparing the Example Project

I continue using the example project in this chapter that I have been working with since Chapter 9. To prepare for this chapter, I have replaced the contents of the template for the `ProductTable` component with the elements shown in Listing 17-1.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 17-1. Replacing the Contents of the `productTable.component.html` File in the `src/app` Folder

```
<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of getProducts(); let i = index">
      <td>{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price | currency:"USD": "symbol" }}</td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm"
               (click)="deleteProduct(item.id)">
          Delete
        </button>
      </td>
    </tr>
  </tbody>
</table>
```

Run the following command in the `example` folder to start the TypeScript compiler and the development HTTP server:

```
ng serve
```

Open a new browser window and navigate to `http://localhost:4200` to see the content shown in Figure 17-1.

	Name	Category	Price	
1	Kayak	Watersports	\$275.00	<button>Delete</button>
2	Lifejacket	Watersports	\$48.95	<button>Delete</button>
3	Soccer Ball	Soccer	\$19.50	<button>Delete</button>
4	Corner Flags	Soccer	\$34.95	<button>Delete</button>
5	Thinking Cap	Chess	\$16.00	<button>Delete</button>

Figure 17-1. Running the example application

Understanding the Object Distribution Problem

In Chapter 15, I added components to the project to help break up the monolithic structure of the application. In doing this, I used input and output properties to connect components, using host elements to bridge the isolation that Angular enforces between a parent component and its children. I also showed you how to query the contents of the template for view children, which complements the content children feature described in Chapter 14.

These techniques for coordinating between directives and components can be powerful and useful if applied carefully. But they can also end up as a general tool for distributing shared objects throughout an application, where the result is to increase the complexity of the application and to tightly bind components together.

Demonstrating the Problem

To help demonstrate the problem, I am going to add a shared object to the project and two components that rely on it. I created a file called `discount.service.ts` to the `src/app` folder and defined the class shown in Listing 17-2. I'll explain the significance of the service part of the filename later in the chapter.

Listing 17-2. The Contents of the `discount.service.ts` File in the `src/app` Folder

```
export class DiscountService {
    private discountValue: number = 10;

    public get discount(): number {
        return this.discountValue;
    }
}
```

```

public set discount(newValue: number) {
    this.discountValue = newValue ?? 0;
}

public applyDiscount(price: number) {
    return Math.max(price - this.discountValue, 5);
}

}

```

The `DiscountService` class defines a private property called `discountValue` that is used to store a number that will be used to reduce the product prices in the data model. This value is exposed through getters and setters called `discount`, and there is a convenience method called `applyDiscount` that reduces a price while ensuring that a price is never less than \$5.

For the first component that makes use of the `DiscountService` class, I added a file called `discountDisplay.component.ts` to the `src/app` folder and added the code shown in Listing 17-3.

Listing 17-3. The Contents of the `discountDisplay.component.ts` File in the `src/app` Folder

```

import { Component, Input } from "@angular/core";
import { DiscountService } from "./discount.service";

@Component({
    selector: "paDiscountDisplay",
    template: `<div class="bg-info text-white p-2 my-2">
        The discount is {{discounter?.discount }}
    </div>`
})
export class PaDiscountDisplayComponent {

    @Input("discounter")
    discounter?: DiscountService;
}

```

The `DiscountDisplayComponent` uses an inline template to display the discount amount, which is obtained from a `DiscountService` object received through an `input` property.

For the second component that makes use of the `DiscountService` class, I added a file called `discountEditor.component.ts` to the `src/app` folder and added the code shown in Listing 17-4.

Listing 17-4. The Contents of the `discountEditor.component.ts` File in the `src/app` Folder

```

import { Component, Input } from "@angular/core";
import { DiscountService } from "./discount.service";

@Component({
    selector: "paDiscountEditor",
    template: `<div class="form-group">
        <label>Discount</label>
        <ng-template [ngIf]="discounter?.discount ?? false">
            <input [(ngModel)]="discounter!.discount"
                class="form-control" type="number" />
        </ng-template>
    </div>`
})

```

```

        `
```

`})
export class PaDiscountEditorComponent {

 @Input("discounter")
 discouter?: DiscountService;
}`

The `DiscountEditorComponent` uses an inline template with an `input` element that allows the discount amount to be edited. The `input` element has a two-way binding on the `DiscountService.discount` property that targets the `ngModel` directive. Listing 17-5 shows the new components being enabled in the Angular module.

Listing 17-5. Enabling the Components in the `app.module.ts` File in the `src/app` Folder

```

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoWay.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";

registerLocaleData(localeFr);

@NgModule({
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaToggleView, PaAddTaxPipe,
        PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent],

```

```

imports: [
  BrowserModule,
  BrowserAnimationsModule,
  FormsModule, ReactiveFormsModule
],
// providers: [{ provide: LOCALE_ID, useValue: "fr-FR" }],
bootstrap: [ProductComponent]
})
export class AppModule { }

```

To get the new components working, I added them to the parent component's template, positioning the new content underneath the table that lists the products, which means that I need to edit the `productTable.component.html` file, as shown in Listing 17-6.

Listing 17-6. Adding Component Elements in the `productTable.component.html` File in the `src/app` Folder

```


|           | Name          | Category          | Price                                      |                                                                                                                           |
|-----------|---------------|-------------------|--------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| {{i + 1}} | {{item.name}} | {{item.category}} | {{item.price   currency:"USD": "symbol" }} | <button class="btn btn-danger btn-sm"                (click)="deleteProduct(item.id)">           Delete         </button> |


<paDiscountEditor [discounter]="discouter"></paDiscountEditor>
<paDiscountDisplay [discouter]="discouter"></paDiscountDisplay>

```

These elements correspond to the components' selector properties in Listing 17-3 and Listing 17-4 and use data bindings to set the value of the input properties. The final step is to create an object in the parent component that will provide the value for the data binding expressions, as shown in Listing 17-7.

Listing 17-7. Creating the Shared Object in the `productTable.component.ts` File in the `src/app` Folder

```

import { Component, Input, QueryList, ViewChildren } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
//import { Subject } from "rxjs";
import { DiscountService } from "./discount.service";

```

```

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {
  discounter: DiscountService = new DiscountService();

  @Input("model")
  dataModel: Model | undefined;

  getProduct(key: number): Product | undefined {
    return this.dataModel?.getProduct(key);
  }

  getProducts(): Product[] | undefined {
    return this.dataModel?.getProducts();
  }

  deleteProduct(key: number) {
    this.dataModel?.deleteProduct(key);
  }

  taxRate: number = 0;
  categoryFilter: string | undefined;
  itemCount: number = 3;

  // selectMap = {
  //   "Watersports": "stay dry",
  //   "Soccer": "score goals",
  //   "other": "have fun"
  // }

  // numberMap = {
  //   "=1": "one product",
  //   "=2": "two products",
  //   "other": "# products"
  // }

  // numbers: Subject<number> = new Subject<number>();

  // ngOnInit() {
  //   let counter = 100;
  //   setInterval(() => {
  //     this.numbers.next(counter += 10)
  //   }, 1000);
  // }
}

```

Figure 17-2 shows the content from the new components. Changes to the value in the `input` element provided by one of the components will be reflected in the content presented by the other component, reflecting the use of the shared `DiscountService` object and its `discount` property.

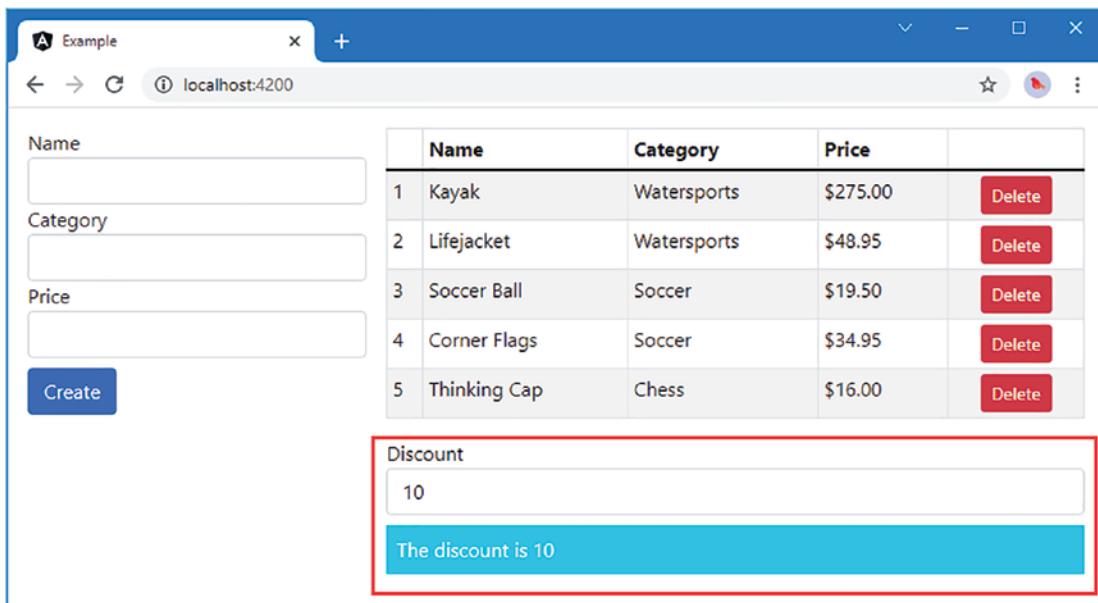


Figure 17-2. Adding components to the example application

The process for adding the new components and the shared object was straightforward and logical, until the final stage. The problem arises in the way that I had to create and distribute the shared object: the instance of the `DiscountService` class.

Because Angular isolates components from one another, I had no way to share the `DiscountService` object directly between the `DiscountEditorComponent` and `DiscountDisplayComponent`. Each component could have created its own `DiscountService` object, but that means changes from the editor component wouldn't be shown in the display component.

That is what led me to create the `DiscountService` object in the `ProductTableComponent` class, which is the first shared ancestor of the discount editor and display components. This allowed me to distribute the `DiscountService` object through the `ProductTableComponent`'s template, ensuring that a single object was shared with both of the components that need it.

But there are a couple of problems. The first is that the `ProductTableComponent` class doesn't actually need or use a `DiscountService` object to deliver its own functionality. It just happens to be the first common ancestor of the components that do need the object. And creating the shared object in the `ProductTableComponent` class makes that class slightly more complex and slightly more difficult to test effectively. This is a modest increment of complexity, but it will occur for every shared object that the application requires—and a complex application can depend on a lot of shared objects, each of which ends up being created by components that just happen to be the first common ancestor of the classes that depend on them.

The second problem is hinted at by the term *first common ancestor*. The `ProductTableComponent` class happens to be the parent of both of the classes that depend on the `DiscountService` object, but think about what would happen if I wanted to move the `DiscountEditorComponent` so that it was displayed under the form rather than the table. In this situation, I have to work my way up the tree of components until I find a common ancestor, which would end up being the root component. Then I would have to work my way down the component tree adding input properties and modifying templates so that each intermediate component could receive the `DiscountService` object from its parent and pass it on to any children that have

descendants that need it. The same applies to any directives that depend on receiving a `DiscountService` object, where any component whose template contains data bindings that target that directive must make sure they are part of the distribution chain, too.

The result is that the components and directives in the application become tightly bound together. A major refactoring is required if you need to move or reuse a component in a different part of the application and the management of the input properties and data bindings becomes unmanageable.

Distributing Objects as Services Using Dependency Injection

There is a better way to distribute objects to the classes that depend on them, which is to use *dependency injection*, where objects are provided to classes from an external source. Angular includes a built-in dependency injection system and supplies the external source of objects, known as *providers*. In the sections that follow, I rework the example application to provide the `DiscountService` object without needing to use the component hierarchy as a distribution mechanism.

Preparing the Service

Any object that is managed and distributed through dependency injection is called a *service*, which is why I selected the name `DiscountService` for the class that defines the shared object and why that class is defined in a file called `discount.service.ts`. Angular denotes service classes using the `@Injectable` decorator, as shown in Listing 17-8. The `@Injectable` decorator doesn't define any configuration properties.

Listing 17-8. Preparing a Class as a Service in the `discount.service.ts` File in the `src/app` Folder

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

@Injectable()
export class DiscountService {
    private discountValue: number = 10;

    public get discount(): number {
        return this.discountValue;
    }

    public set discount(newValue: number) {
        this.discountValue = newValue || 0;
    }

    public applyDiscount(price: number) {
        return Math.max(price - this.discountValue, 5);
    }
}
```

■ **Tip** Strictly speaking, the `@Injectable` decorator is required only when a class has its own constructor arguments to resolve, but it is a good idea to apply it anyway because it serves as a signal that the class is intended for use as a service.

Preparing the Dependent Components

A class declares dependencies using its constructor. When Angular needs to create an instance of the class—such as when it finds an element that matches the `selector` property defined by a component—its constructor is inspected, and the type of each argument is examined. Angular then uses the services that have been defined to try to satisfy the dependencies. The term *dependency injection* arises because each dependency is *injected* into the constructor to create the new instance.

For the example application, it means that the components that depend on a `DiscountService` object no longer require input properties and can declare a constructor dependency instead. Listing 17-9 shows the changes to the `DiscountDisplayComponent` class.

Listing 17-9. Declaring a Dependency in the `discountDisplay.component.ts` File in the `src/app` Folder

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "./discount.service";

@Component({
  selector: "paDiscountDisplay",
  template: `<div class="bg-info text-white p-2 my-2">
    The discount is {{discouter?.discount }}
  </div>`
})
export class PaDiscountDisplayComponent {

  constructor(public discouter: DiscountService) { }

  // @Input("discouter")
  // discouter?: DiscountService;
}
```

The same change can be applied to the `DiscountEditorComponent` class, replacing the input property with a dependency declared through the constructor, as shown in Listing 17-10.

Listing 17-10. Declaring a Dependency in the `discountEditor.component.ts` File in the `src/app` Folder

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "./discount.service";

@Component({
  selector: "paDiscountEditor",
  template: `<div class="form-group">
    <label>Discount</label>
    <input [(ngModel)]="discouter.discount"
      class="form-control" type="number" />
  </div>`
})
export class PaDiscountEditorComponent {

  constructor(public discouter: DiscountService) { }

  // @Input("discouter")
  // discouter?: DiscountService;
}
```

These are small changes, but they avoid the need to distribute objects using templates and input properties and produce a more flexible application. And, since the value of the `discounter` property is set in the constructor, I can simplify the template so that it doesn't need to deal with undefined values.

I can now remove the `DiscountService` object from the product table component, as shown in Listing 17-11.

Listing 17-11. Removing the Shared Object in the `productTable.component.ts` File in the `src/app` Folder

```
import { Component, Input, QueryList, ViewChildren } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
//import { Subject } from "rxjs";
import { DiscountService } from "./discount.service";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {
  //discounter: DiscountService = new DiscountService();

  @Input("model")
  dataModel: Model | undefined;

  getProduct(key: number): Product | undefined {
    return this.dataModel?.getProduct(key);
  }

  getProducts(): Product[] | undefined {
    return this.dataModel?.getProducts();
  }

  deleteProduct(key: number) {
    this.dataModel?.deleteProduct(key);
  }

  taxRate: number = 0;
  categoryFilter: string | undefined;
  itemCount: number = 3;
}
```

Since the parent component is no longer providing the shared object through data bindings, I can remove them from the template, as shown in Listing 17-12.

Listing 17-12. Removing the Data Bindings in the `productTable.component.html` File in the `src/app` Folder

```
<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of getProducts(); let i = index">
```

```

<td>{{i + 1}}</td>
<td>{{item.name}}</td>
<td>{{item.category}}</td>
<td>{{item.price | currency:"USD": "symbol" }}</td>
<td class="text-center">
    <button class="btn btn-danger btn-sm"
           (click)="deleteProduct(item.id)">
        Delete
    </button>
</td>
</tr>
</tbody>
</table>

<paDiscountEditor></paDiscountEditor>
<paDiscountDisplay></paDiscountDisplay>

```

Registering the Service

The final change is to configure the dependency injection feature so that it can provide `DiscountService` objects to the components that require them. To make the service available throughout the application, it is registered in the Angular module, as shown in Listing 17-13.

Listing 17-13. Registering a Service in the app.module.ts File in the src/app Folder

```

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { PaAttrDirective } from './attr.directive';
import { PaModel } from './twoway.directive';
import { PaStructureDirective } from './structure.directive';
import { PaIteratorDirective } from './iterator.directive';
import { PaCellColor } from './cellColor.directive';
import { PaCellColorSwitcher } from './cellColorSwitcher.directive';
import { ProductTableComponent } from './productTable.component';
import { ProductFormComponent } from './productForm.component';
import { PaToggleView } from './toggleView.component';
import { PaAddTaxPipe } from './addTax.pipe';
import { PaCategoryFilterPipe } from './categoryFilter.pipe';

import { LOCALE_ID } from '@angular/core';
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

```

```

import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";

registerLocaleData(localeFr);

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [DiscountService],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

The NgModule decorator's `providers` property is set to an array of the classes that will be used as services. There is only one service at the moment, which is provided by the `DiscountService` class.

When you save the changes to the application, there won't be any visual changes, but the dependency injection feature will be used to provide the components with the `DiscountService` object they require.

Reviewing the Dependency Injection Changes

Angular seamlessly integrates dependency injection into its feature set. Each time that Angular encounters an element that requires a new building block, such as a component or a pipe, it examines the class constructor to check what dependencies have been declared and uses its services to try to resolve them. The set of services used to resolve dependencies includes the custom services defined by the application, such as the `DiscountService` service that was registered in Listing 17-13, and a set of built-in services provided by Angular that will be described in later chapters.

The changes to introduce dependency injection in the previous section didn't result in a big-bang change in the way that the application works—or any visible change at all. But there is a profound difference in the way that the application is put together that makes it more flexible and fluid. The best demonstration of this is to add the components that require the `DiscountService` to a different part of the application, as shown in Listing 17-14.

Listing 17-14. Adding Components in the productForm.component.html File in the src/app Folder

```

<form #form="ngForm" (ngSubmit)="submitForm(form)">
  <div class="form-group">
    <label>Name</label>
    <input class="form-control"
      name="name" [(ngModel)]="newProduct.name" />
  </div>

```

```

<div class="form-group">
    <label>Category</label>
    <input class="form-control"
        name="category" [(ngModel)]="newProduct.category" />
</div>
<div class="form-group">
    <label>Price</label>
    <input class="form-control"
        name="name" [(ngModel)]="newProduct.price" />
</div>
<button class="btn btn-primary mt-2" type="submit">
    Create
</button>
</form>

<paDiscountEditor></paDiscountEditor>
<paDiscountDisplay></paDiscountDisplay>

```

These new elements duplicate the discount display and editor components so they appear below the form used to create new products, as shown in Figure 17-3.

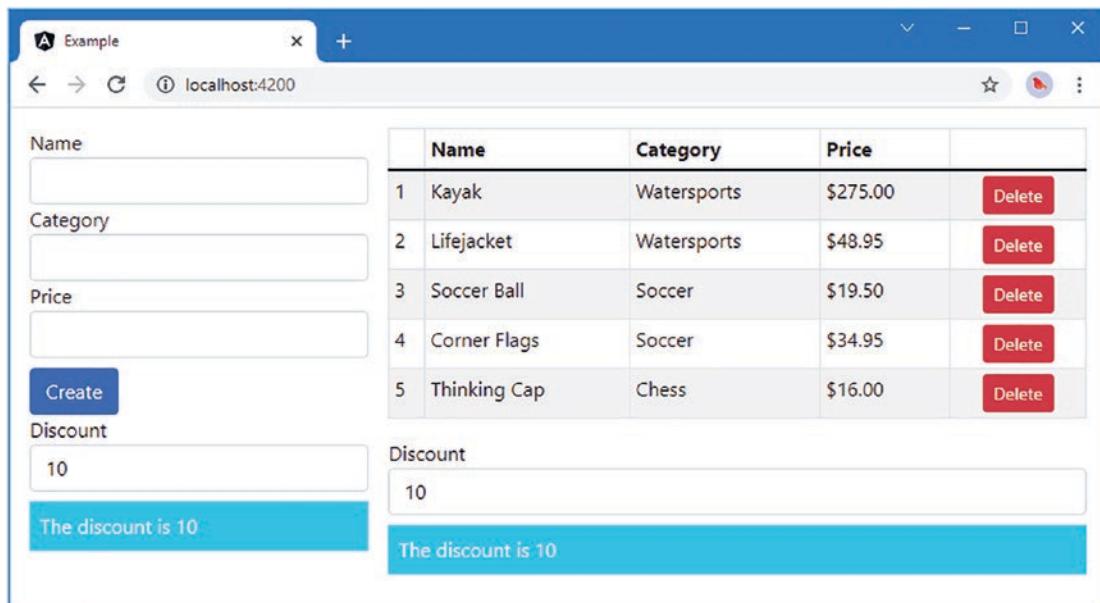


Figure 17-3. Duplicating components with dependencies

There are two points of note. First, using dependency injection made this a simple process of adding elements to a template, without needing to modify the ancestor components to provide a `DiscountService` object using input properties.

The second point of note is that all the components in the application that have declared a dependency on `DiscountService` have received the same object. If you edit the value in either of the input elements, the changes will be reflected in the other input element and in the string interpolation bindings, as shown in Figure 17-4.

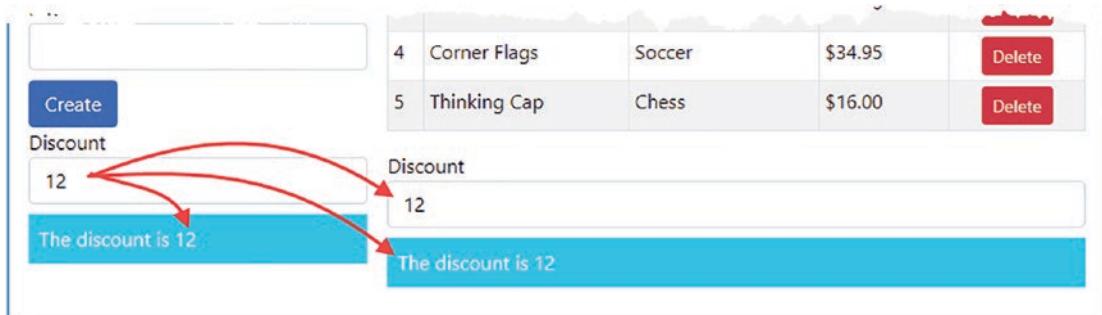


Figure 17-4. Checking that the dependency is resolved using a shared object

Declaring Dependencies in Other Building Blocks

It isn't just components that can declare constructor dependencies. Once you have defined a service, you can use it more widely, including other building blocks in the application, such as pipes and directives, as demonstrated in the sections that follow.

Declaring a Dependency in a Pipe

Pipes can declare dependencies on services by defining a constructor with arguments for each required service. To demonstrate, I added a file called `discount.pipe.ts` to the `src/app` folder and used it to define the pipe shown in Listing 17-15.

Listing 17-15. The Contents of the `discount.pipe.ts` File in the `src/app` Folder

```
import { Pipe } from "@angular/core";
import { DiscountService } from "./discount.service";

@Pipe({
  name: "discount",
  pure: false
})
export class PaDiscountPipe {

  constructor(private discount: DiscountService) { }

  transform(price: number): number {
    return this.discount.applyDiscount(price);
  }
}
```

The `PaDiscountPipe` class is a pipe that receives a price and generates a result by calling the `DiscountService.applyDiscount` method, where the service is received through the constructor. The `pure` property in the `Pipe` decorator is `false`, which means that the pipe will be asked to update its result when the value stored by the `DiscountService` changes, which won't be recognized by the Angular change-detection process.

Tip This feature should be used with caution because it means that the `transform` method will be called after every change in the application, not just when the service is changed.

Listing 17-16 shows the new pipe being registered in the application's Angular module.

Listing 17-16. Registering a Pipe in the `app.module.ts` File in the `src/app` Folder

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";

registerLocaleData(localeFr);

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
    PaDiscountPipe],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule,
    ReactiveFormsModule
  ],
  providers: [DiscountService]
})
```

```

imports: [
  BrowserModule,
  BrowserAnimationsModule,
  FormsModule, ReactiveFormsModule
],
providers: [DiscountService],
bootstrap: [ProductComponent]
})
export class AppModule { }

```

Listing 17-17 shows the new pipe applied to the Price column in the product table.

Listing 17-17. Applying a Pipe in the productTable.component.html File in the src/app Folder

```

<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of getProducts(); let i = index">
      <td>{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price | discount | currency:"USD": "symbol" }}</td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm"
               (click)="deleteProduct(item.id)">
          Delete
        </button>
      </td>
    </tr>
  </tbody>
</table>

<paDiscountEditor></paDiscountEditor>
<paDiscountDisplay></paDiscountDisplay>

```

The discount pipe processes the price to apply the discount and then passes on the value to the currency pipe for formatting. You can see the effect of using the service in the pipe by changing the value in one of the discount input elements, as shown in Figure 17-5.

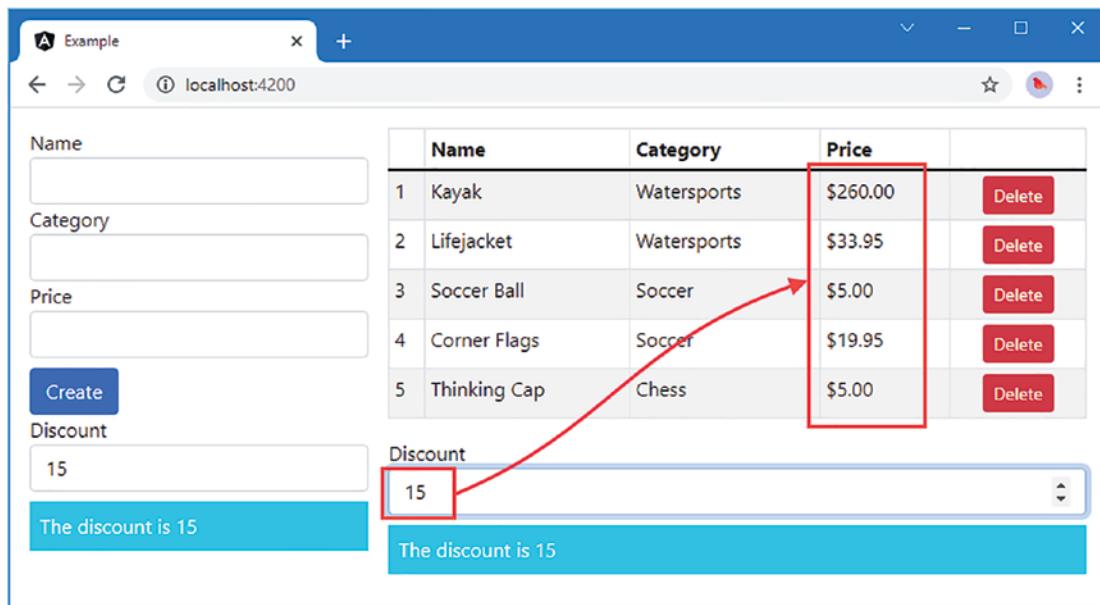


Figure 17-5. Using a service in a pipe

Declaring Dependencies in Directives

Directives can also use services. As I explained in Chapter 15, components are just directives with templates, so anything that works in a component will also work in a directive.

To demonstrate using a service in a directive, I added a file called `discountAmount.directive.ts` to the `src/app` folder and used it to define the directive shown in Listing 17-18.

Listing 17-18. The Contents of the `discountAmount.directive.ts` File in the `src/app` Folder

```
import { Directive, Input, SimpleChange, KeyValueDiffer,
  KeyValueDiffers } from "@angular/core";
import { DiscountService } from "./discount.service";

@Directive({
  selector: "td[pa-price]",
  exportAs: "discount"
})
export class PaDiscountAmountDirective {
  private differ?: KeyValueDiffer<any, any>;
  constructor(private keyValueDiffers: KeyValueDiffers,
    private discount: DiscountService) { }

  @Input("pa-price")
  originalPrice?: number;

  discountAmount?: number;
}
```

```

ngOnInit() {
  this.differ =
    this.keyValueDifferers.find(this.discount).create();
}

ngOnChanges(changes: { [property: string]: SimpleChange }) {
  if (changes["originalPrice"] != null) {
    this.updateValue();
  }
}

ngDoCheck() {
  if (this.differ?.diff(this.discount) != null) {
    this.updateValue();
  }
}

private updateValue() {
  this.discountAmount = this.discount.applyDiscount(this.originalPrice ?? 0);
}
}

```

Directives don't have an equivalent to the pure property used by pipes and must take direct responsibility for responding to changes propagated through services. This directive displays the discounted amount for a product. The selector property matches `td` elements that have a `pa-price` attribute, which is also used as an input property to receive the price that will be discounted. The directive exports its functionality using the `exportAs` property and provides a property called `discountAmount` whose value is set to the discount that has been applied to the product.

There are two other points to note about this directive. The first is that the `DiscountService` object isn't the only constructor parameter in the directive's class.

```

...
constructor(private keyValueDiffer: KeyValueDiffer,
            private discount: DiscountService) { }
...

```

The `KeyValueDiffer` parameter is also a dependency that Angular will have to resolve when it creates a new instance of the directive class. This is an example of the built-in services that Angular provides that deliver commonly required functionality.

The second point of note is what the directive does with the services it receives. The components and the pipe that use the `DiscountService` service don't have to worry about tracking updates, either because Angular automatically evaluates the expressions of the data bindings and updates them when the discount rate change (for the components) or because any change in the application triggers an update (for the impure pipe). The data binding for this directive is on the `price` property, which will trigger a change if it is altered. But there is also a dependency on the `discount` property defined by the `DiscountService` class. Changes in the `discount` property are detected using the service received through the constructor, which tracks changes as described in Chapter 14. When Angular invokes the `ngDoCheck` method, the directive uses the key-value pair differ to see whether there has been a change. (This change detection could also have been handled by keeping track of the previous update in the directive class, but I wanted to provide an example of using the key-value differ feature.)

The directive also implements the `ngOnChanges` method so that it can respond to changes in the value of the input property. For both types of update, the `updateValue` method is called, which calculates the discounted price and assigns it to the `discountAmount` property.

[Listing 17-19](#) registers the new directive in the application's Angular module.

Listing 17-19. Registering a Directive in the `app.module.ts` File in the `src/app` Folder

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";

registerLocaleData(localeFr);

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
    PaDiscountPipe, PaDiscountAmountDirective],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
})
```

```

    providers: [DiscountService],
    bootstrap: [ProductComponent]
})
export class AppModule { }

```

To apply the new directive, Listing 17-20 adds a new column to the table, using a string interpolation binding to access the property provided by the directive and to pass it to the currency pipe.

Listing 17-20. Creating a New Column in the productTable.component.html File in the src/app Folder

```

<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of getProducts(); let i = index">
      <td>{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td [pa-price]="item.price" #discount="discount">
        {{ discount.discountAmount | currency:"USD":"symbol" }}</td>
      </td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm"
               (click)="deleteProduct(item.id)">
          Delete
        </button>
      </td>
    </tr>
  </tbody>
</table>

<paDiscountEditor></paDiscountEditor>
<paDiscountDisplay></paDiscountDisplay>

```

The directive could have created a host binding on the `textContent` property to set the contents of its host element, but that would have prevented the currency pipe from being used. Instead, the directive is assigned to the `discount` template variable, which is then used in the string interpolation binding to access and then format the `discountAmount` value. Figure 17-6 shows the results. Changes to the discount amount in either of the discount editor input elements will be reflected in the new table column.

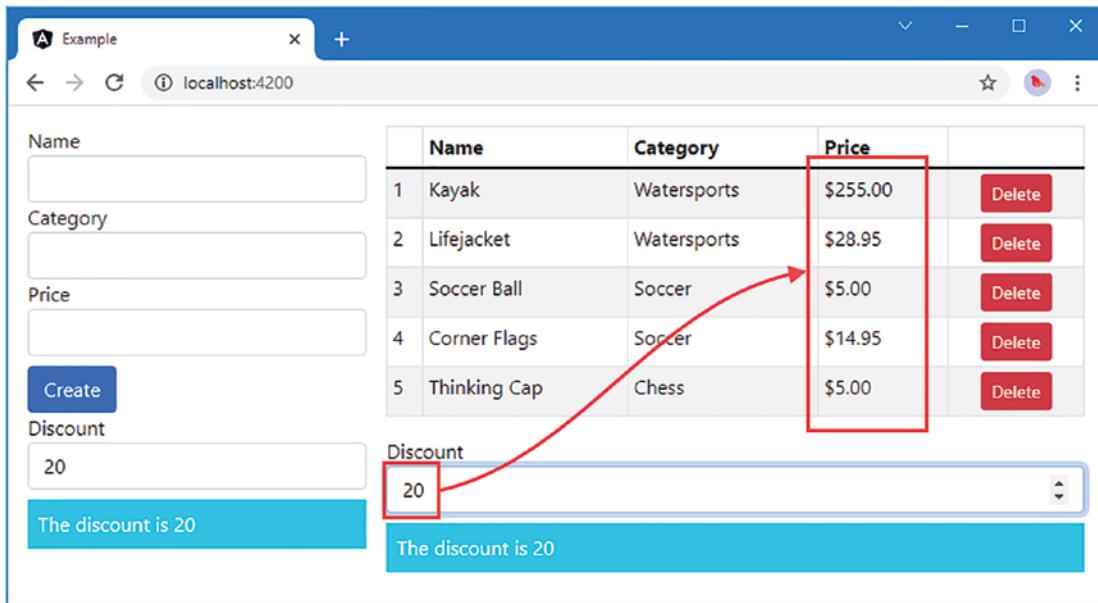


Figure 17-6. Using a service in a directive

Understanding the Test Isolation Problem

The example application contains a related problem that services and dependency injection can be used to solve. Consider how the Model class is used in the root component:

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
  selector: "app",
  templateUrl: "template.html",
})
export class ProductComponent {
  model: Model = new Model();

  addProduct(p: Product) {
    this.model.saveProduct(p);
  }
}
```

The root component is defined as the `ProductComponent` class, and it sets up a value for its `model` property by creating a new instance of the `Model` class. This works—and is a legitimate way to create an object—but it makes it harder to perform unit testing effectively.

Unit testing works best when you can isolate one small part of the application and focus on it to perform tests. But when you create an instance of the `ProductComponent` class, you are implicitly creating an instance of the `Model` class as well. If you were to run tests on the root component's `addProduct` method and find a problem, you would have no indication of whether the problem was in the `ProductComponent` or `Model` class.

Isolating Components Using Services and Dependency Injection

The underlying problem is that the `ProductComponent` class is tightly bound to the `Model` class, which is, in turn, tightly bound to the `SimpleDataSource` class. Dependency injection can be used to tease apart the building blocks in an application so that each class can be isolated and tested on its own. In the sections that follow, I walk through the process of breaking up these tightly coupled classes, following essentially the same process as in the previous section but delving deeper into the example application.

Preparing the Services

The `@Injectable` decorator is used to denote services, just as in the previous example. Listing 17-21 shows the decorator applied to the `SimpleDataSource` class.

Listing 17-21. Denoting a Service in the `datasource.model.ts` File in the `src/app` Folder

```
import { Product } from "./product.model";
import { Injectable } from "@angular/core";

@Injectable()
export class SimpleDataSource {
  private data: Product[];

  constructor() {
    this.data = new Array<Product>(
      new Product(1, "Kayak", "Watersports", 275),
      new Product(2, "Lifejacket", "Watersports", 48.95),
      new Product(3, "Soccer Ball", "Soccer", 19.50),
      new Product(4, "Corner Flags", "Soccer", 34.95),
      new Product(5, "Thinking Cap", "Chess", 16));
  }

  getData(): Product[] {
    return this.data;
  }
}
```

No other changes are required. Listing 17-22 shows the same decorator being applied to the data repository, and since this class has a dependency on the `SimpleDataSource` class, it declares it as a constructor dependency rather than creating an instance directly.

Listing 17-22. Denoting a Service and Dependency in the repository.model.ts File in the src/app Folder

```
import { Product } from "./product.model";
import { SimpleDataSource } from "./datasource.model";
import { Injectable } from "@angular/core";

@Injectable()
export class Model {
    //private dataSource: SimpleDataSource;
    private products: Product[];
    private locator = (p: Product, id: number | any) => p.id == id;

    constructor(private dataSource: SimpleDataSource) {
        //this.dataSource = new SimpleDataSource();
        this.products = new Array<Product>();
        this.dataSource.getData().forEach(p => this.products.push(p));
    }

    // ...other members omitted for brevity...
}
```

The important point to note in this listing is that services can declare dependencies on other services. When Angular comes to create a new instance of a service class, it inspects the constructor and tries to resolve the services in the same way as when dealing with a component or directive.

Registering the Services

These services must be registered so that Angular knows how to resolve dependencies on them, as shown in Listing 17-23.

Listing 17-23. Registering the Services in the app.module.ts File in the src/app Folder

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { PaAttrDirective } from './attr.directive';
import { PaModel } from './twoway.directive';
import { PaStructureDirective } from './structure.directive';
import { PaIteratorDirective } from './iterator.directive';
import { PaCellColor } from './cellColor.directive';
import { PaCellColorSwitcher } from './cellColorSwitcher.directive';
import { ProductTableComponent } from './productTable.component';
import { ProductFormComponent } from './productForm.component';
import { PaToggleView } from './toggleView.component';
import { PaAddTaxPipe } from './addTax.pipe';
import { PaCategoryFilterPipe } from './categoryFilter.pipe';
```

```

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";

registerLocaleData(localeFr);

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
    PaDiscountPipe, PaDiscountAmountDirective],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [DiscountService, SimpleDataSource, Model],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

Preparing the Dependent Component

Rather than create a Model object directly, the root component can declare a constructor dependency that Angular will resolve using dependency injection when the application starts, as shown in Listing 17-24.

Listing 17-24. Declaring a Service Dependency in the component.ts File in the src/app Folder

```

import { Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
  selector: "app",
  templateUrl: "template.html",
})
export class ProductComponent {
  //model: Model = new Model();
}

```

```
constructor(public model: Model) { }

addProduct(p: Product) {
    this.model.saveProduct(p);
}
}
```

There is now a chain of dependencies that Angular has to resolve. When the application starts, the Angular module specifies that the `ProductComponent` class needs a `Model` object. Angular inspects the `Model` class and finds that it needs a `SimpleDataSource` object. Angular inspects the `SimpleDataSource` object and finds that there are no declared dependencies and therefore knows that this is the end of the chain. It creates a `SimpleDataSource` object and passes it as an argument to the `Model` constructor to create a `Model` object, which can then be passed to the `ProductComponent` class constructor to create the object that will be used as the root component. All of this happens automatically, based on the constructors defined by each class and the use of the `@Injectable` decorator.

These changes don't create any visible changes in the way that the application works, but they do allow a completely different way of performing unit tests. The `ProductComponent` class requires that a `Model` object is provided as a constructor argument, which allows for a mock object to be used.

Breaking up the direct dependencies between the classes in the application means that each of them can be isolated for the purposes of unit testing and provided with mock objects through their constructor, allowing the effect of a method or some other feature to be consistently and independently assessed.

Completing the Adoption of Services

Once you start using services in an application, the process generally takes on a life of its own, and you start to examine the relationships between the building blocks you have created. The extent to which you introduce services is—at least in part—a matter of personal preference.

A good example is the use of the `Model` class in the root component. Although the component does implement a method that uses the `Model` object, it does so because it needs to handle a custom event from one of its child components. The only other reason that the root component has for needing a `Model` object is to pass it on via its template to the other child component using an input property.

This situation isn't an enormous problem, and your preference may be to have these kinds of relationships in a project. After all, each of the components can be isolated for unit testing, and there is some purpose, however limited, to the relationships between them. This kind of relationship between components can help make sense of the functionality that an application provides.

On the other hand, the more you use services, the more the building blocks in your project become self-contained and reusable blocks of functionality, which can ease the process of adding or changing functionality as the project matures.

There is no absolute right or wrong, and you must find the balance that suits you, your team, and, ultimately, your users and customers. Not everyone likes using dependency injection, and not everyone performs unit testing.

My preference is to use dependency injection as widely as possible. I find that the final structure of my applications can differ significantly from what I expect when I start a new project and that the flexibility offered by dependency injection helps me avoid repeated periods of refactoring. So, to complete this chapter, I am going to push the use of the `Model` service into the rest of the application, breaking the coupling between the root component and its immediate children.

Updating the Root Component and Template

The first changes I will make are to remove the `Model` object from the root component, along with the method that uses it and the input property in the template that distributes the model to one of the child components. Listing 17-25 shows the changes to the component class.

Listing 17-25. Removing the Model Object from the component.ts File in the src/app Folder

```
import { Component } from "@angular/core";
// import { Model } from "./repository.model";
// import { Product } from "./product.model";

@Component({
  selector: "app",
  templateUrl: "template.html",
})
export class ProductComponent {
  // constructor(public model: Model) { }

  // addProduct(p: Product) {
  //   this.model.saveProduct(p);
  // }
}
```

The revised root component class doesn't define any functionality and now exists only to provide the top-level application content in its template. Listing 17-26 shows the corresponding changes in the root template to remove the custom event binding and the input property.

Listing 17-26. Removing the Data Bindings in the template.html File in the src/app Folder

```
<div class="container-fluid angularApp">
  <div class="row p-2">
    <div class="col-4 p-2 text-dark">
      <pa-productform></pa-productform>
    </div>
    <div class="col p-2">
      <paProductTable></paProductTable>
    </div>
  </div>
</div>
```

Updating the Child Components

The component that provides the form for creating new `Product` objects relied on the root component to handle its custom event and update the model. Without this support, the component must now declare a `Model` dependency and perform the update itself, as shown in Listing 17-27.

Listing 17-27. Working with the Model in the productForm.component.ts File in the src/app Folder

```

import { Component, Output, EventEmitter, ViewEncapsulation } from "@angular/core";
import { Product } from "./product.model";
import { Model } from "./repository.model";

@Component({
  selector: "pa-productform",
  templateUrl: "productForm.component.html",
  // styleUrls: ["productForm.component.css"],
  // encapsulation: ViewEncapsulation.ShadowDom
})
export class ProductFormComponent {
  newProduct: Product = new Product();

  constructor(private model: Model) { }

  // @Output("paNewProduct")
  // newProductEvent = new EventEmitter<Product>();

  submitForm(form: any) {
    // this.newProductEvent.emit(this.newProduct);
    this.model.saveProduct(this.newProduct);
    this.newProduct = new Product();
    form.resetForm();
  }
}

```

The component that manages the table of product objects used an input property to receive a Model object from its parent but must now obtain it directly by declaring a constructor dependency, as shown in Listing 17-28.

Listing 17-28. Declaring a Model Dependency in the productTable.component.ts File in the src/app Folder

```

import { Component, Input, QueryList, ViewChildren } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
//import { Subject } from "rxjs";
import { DiscountService } from "./discount.service";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {
  //discouter: DiscountService = new DiscountService();

  constructor(private dataModel: Model) { }

  // @Input("model")
  // dataModel: Model | undefined;

```

```
getProduct(key: number): Product | undefined {
    return this.dataModel?.getProduct(key);
}

getProducts(): Product[] | undefined {
    return this.dataModel?.getProducts();
}

deleteProduct(key: number) {
    this.dataModel?.deleteProduct(key);
}

taxRate: number = 0;
categoryFilter: string | undefined;
itemCount: number = 3;
}
```

You will see the same functionality displayed in the browser window when all the changes have been saved and the browser reloads the Angular application—but the way that the functionality is wired up has changed substantially, with each component obtaining the share objects it needs through the dependency injection feature, rather than relying on its parent component to provide it.

Summary

In this chapter, I explained the problems that dependency injection can be used to address and demonstrated the process of defining and consuming services. I described how services can be used to increase the flexibility in the structure of an application and how dependency injection makes it possible to isolate building blocks so they can be unit tested effectively. In the next chapter, I describe the advanced features that Angular provides for working with services.

CHAPTER 18



Using Service Providers

In the previous chapter, I introduced services and explained how they are distributed using dependency injection. When using dependency injection, the objects that are used to resolve dependencies are created by *service providers*, known more commonly as *providers*. In this chapter, I explain how providers work, describe the different types of providers available, and demonstrate how providers can be created in different parts of the application to change the way that services behave. Table 18-1 puts providers in context.

WHY YOU SHOULD CONSIDER SKIPPING THIS CHAPTER

Dependency injection provokes strong reactions in developers and polarizes opinion. If you are new to dependency injection and have yet to form your own opinion, then you might want to skip this chapter and just use the features that I described in Chapter 17. That's because features like the ones I describe in this chapter are exactly why many developers dread using dependency injection and form a strong preference against its use.

The basic Angular dependency injection features are easy to understand and have an immediate and obvious benefit in making applications easier to write and maintain. The features described in this chapter provide fine-grained control over how dependency injection works, but they also make it possible to sharply increase the complexity of an Angular application and, ultimately, undermine many of the benefits that the basic features offer.

If you decide that you want all of the gritty details, then read on. But if you are new to the world of dependency injection, you may prefer to skip this chapter until you find that the basic features from Chapter 17 don't deliver the functionality you require.

Table 18-1. Putting Service Providers in Context

Question	Answer
What are they?	Providers are classes that create service objects the first time that Angular needs to resolve a dependency.
Why are they useful?	Providers allow the creation of service objects to be tailored to the needs of the application. The simplest provider just creates an instance of a specified class, but there are other providers that can be used to tailor the way that service objects are created and configured.
How are they used?	Providers are defined in the providers property of the Angular module's decorator. They can also be defined by components and directives to provide services to their children, as described in the "Using Local Providers" section.
Are there any pitfalls or limitations?	It is easy to create unexpected behavior, especially when working with local providers. If you encounter problems, check the scope of the local providers you have created and make sure that your dependencies and providers are using the same tokens.
Are there any alternatives?	Many applications will require only the basic dependency injection features described in Chapter 17. You should use the features in this chapter only if you cannot build your application using the basic features and only if you have a solid understanding of dependency injection.

Table 18-2 summarizes the chapter.

Table 18-2. Chapter Summary

Problem	Solution	Listing
Changing the way that services are created	Use a service provider	1-3
Specifying a service using a class	Use the class provider	4-6, 10-13
Defining arbitrary tokens for services	Use the <code>InjectionToken</code> class	7-9
Specifying a service using an object	Use the value provider	14-15
Specifying a service using a function	Use the factory provider	16-18
Specifying one service using another	Use the existing service provider	19
Changing the scope of a service	Use a local service provider	20-28
Controlling the resolution of dependencies	Use the <code>@Host</code> , <code>@Optional</code> , or <code>@SkipSelf</code> decorator	29-30

Preparing the Example Project

As with the other chapters in this part of the book, I am going to continue working with the project created in Chapter 11 and most recently modified in Chapter 19. To prepare for this chapter, I added a file called `log.service.ts` to the `src/app` folder and used it to define the service shown in Listing 18-1.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 18-1. The Contents of the log.service.ts File in the src/app Folder

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

export enum LogLevel {
    DEBUG, INFO, ERROR
}

@Injectable()
export class LogService {
    minimumLevel: LogLevel = LogLevel.INFO;

    logInfoMessage(message: string) {
        this.logMessage(LogLevel.INFO, message);
    }

    logDebugMessage(message: string) {
        this.logMessage(LogLevel.DEBUG, message);
    }

    logErrorMessage(message: string) {
        this.logMessage(LogLevel.ERROR, message);
    }

    logMessage(level: LogLevel, message: string) {
        if (level >= this.minimumLevel) {
            console.log(`Message (${LogLevel[level]}): ${message}`);
        }
    }
}
```

This service writes out log messages, with differing levels of severity, to the browser's JavaScript console. I will register and use this service later in the chapter.

When you have created the service and saved the changes, run the following command in the example folder to start the Angular development tools:

```
ng serve
```

Open a new browser window and navigate to <http://localhost:4200> to see the application, as shown in Figure 18-1.

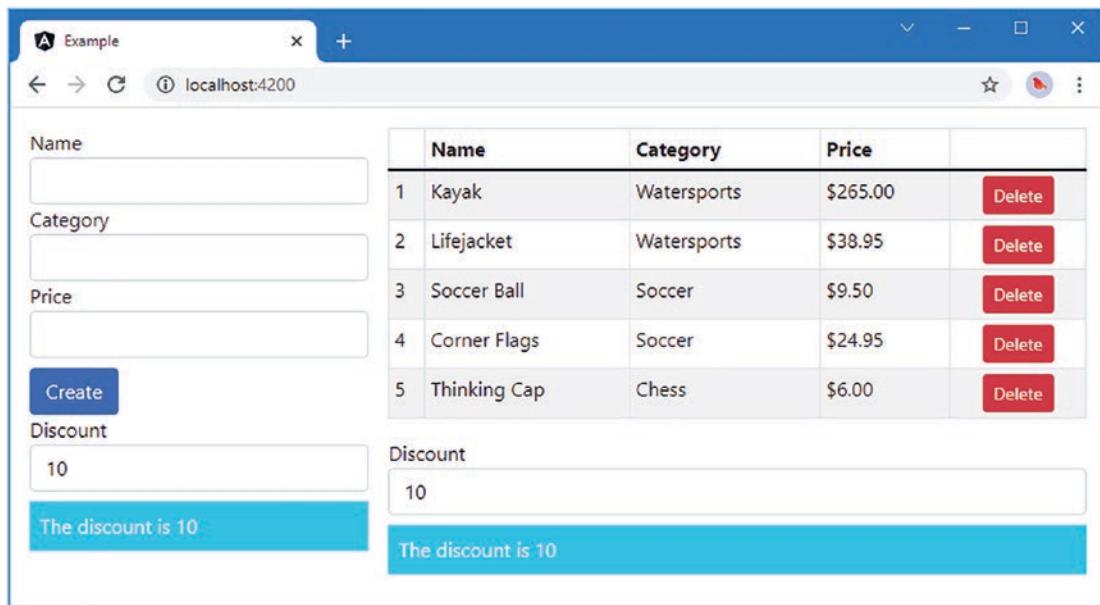


Figure 18-1. Running the example application

Using Service Providers

As I explained in the previous chapters, classes declare dependencies on services using their constructor arguments. When Angular needs to create a new instance of the class, it inspects the constructor and uses a combination of built-in and custom services to resolve each argument. Listing 18-2 updates the `DiscountService` class so that it depends on the `LogService` class created in the previous section.

Listing 18-2. Creating a Dependency in the `discount.service.ts` File in the `src/app` Folder

```
import { Injectable } from "@angular/core";
import { LogService } from "./log.service";

@Injectable()
export class DiscountService {
  private discountValue: number = 10;

  constructor(private logger: LogService) { }

  public get discount(): number {
    return this.discountValue;
  }

  public set discount(newValue: number) {
    this.discountValue = newValue ?? 0;
  }
}
```

```

public applyDiscount(price: number) {
  this.logger.logInfoMessage(`Discount ${this.discount}%
    + applied to price: ${price}`);
  return Math.max(price - this.discountValue, 5);
}
}

```

The changes in Listing 18-2 prevent the application from running. Angular processes the HTML document and starts creating the hierarchy of components, each with their templates that require directives and data bindings, and it encounters the classes that depend on the `DiscountService` class. But it can't create an instance of `DiscountService` because its constructor requires a `LogService` object, and it doesn't know how to handle this class.

When you save the changes in Listing 18-2, you will see an error like this one in the browser's JavaScript console:

```
NullInjectorError: No provider for LogService!
```

Angular delegates responsibility for creating the objects needed for dependency injection to *providers*, each of which managed a single type of dependency. When it needs to create an instance of the `DiscountService` class, it looks for a suitable provider to resolve the `LogService` dependency. Since there is no such provider, Angular can't create the objects it needs to start the application and reports the error.

The simplest way to create a provider is to add the service class to the array assigned to the Angular module's `providers` property, as shown in Listing 18-3. (I have taken the opportunity to remove some of the statements that are no longer required in the module.)

Listing 18-3. Creating a Provider in the `app.module.ts` File in the `src/app` Folder

```

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoWay.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

```

```

import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";
import { LogService } from "./log.service";

registerLocaleData(localeFr);

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
    PaDiscountPipe, PaDiscountAmountDirective],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [DiscountService, SimpleDataSource, Model, LogService],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

When you save the changes, you will have defined the provider that Angular requires to handle the LogService dependency, and you will see messages like this one shown in the browser's JavaScript console:

Message (INFO): Discount 10 applied to price: 16

You might wonder why the configuration step in Listing 18-3 is required. After all, Angular could just assume that it should create a new LogService object the first time it needs one.

In fact, Angular provides a range of different providers, each of which creates objects in a different way to let you take control of the service creation process. Table 18-3 describes the set of available providers, which are described in the sections that follow.

Table 18-3. The Angular Providers

Name	Description
Class provider	This provider is configured using a class. Dependencies on the service are resolved by an instance of the class, which Angular creates.
Value provider	This provider is configured using an object, which is used to resolve dependencies on the service.
Factory provider	This provider is configured using a function. Dependencies on the service are resolved using an object that is created by invoking the function.
Existing service provider	This provider is configured using the name of another service and allows aliases for services to be created.

Using the Class Provider

This provider is the most commonly used and is the one I applied by adding the class names to the module's providers property in Listing 18-3. This listing shows the shorthand syntax, and there is also a literal syntax that achieves the same result, as shown in Listing 18-4.

Listing 18-4. Using the Class Provider Literal Syntax in the app.module.ts File in the src/app Folder

```
...
@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
    PaDiscountPipe, PaDiscountAmountDirective],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [DiscountService, SimpleDataSource, Model,
    { provide: LogService, useClass: LogService }],
  bootstrap: [ProductComponent]
})
...
...
```

Providers are defined as classes, but they can be specified and configured using the JavaScript object literal format, like this:

```
...
{ provide: LogService, useClass: LogService }
...
```

The class provider supports three properties, which are described in Table 18-4 and explained in the sections that follow.

Table 18-4. The Class Provider's Properties

Name	Description
provide	This property is used to specify the token, which is used to identify the provider and the dependency that will be resolved. See the “Understanding the Token” section.
useClass	This property is used to specify the class that will be instantiated to resolve the dependency by the provider. See the “Understanding the useClass Property” section.
multi	This property can be used to deliver an array of service objects to resolve dependencies. See the “Resolving a Dependency with Multiple Objects” section.

Understanding the Token

All providers rely on a token, which Angular uses to identify the dependency that the provider can resolve. The simplest approach is to use a class as the token, which is what I did in Listing 18-4. However, you can use any object as the token, which allows the dependency and the type of the object to be separated. This has the effect of increasing the flexibility of the dependency injection configuration because it allows a provider to supply objects of different types, which can be useful with some of the more advanced providers described later in the chapter. As a simple example, Listing 18-5 uses the class provider to register the log service created at the start of the chapter using a string as a token, rather than a class.

Listing 18-5. Registering a Service with a Token in the app.module.ts File in the src/app Folder

```
...
@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
    PaDiscountPipe, PaDiscountAmountDirective],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [DiscountService, SimpleDataSource, Model,
    { provide: "logger", useClass: LogService }],
  bootstrap: [ProductComponent]
})
...
```

In the listing, the `provide` property of the new provider is set to `logger`. Angular will automatically match providers whose token is a class, but it needs some additional help for other token types. Listing 18-6 shows the `DiscountService` class updated with a dependency on the logging service, accessed using the `logger` token.

Listing 18-6. Using a String Provider Token in the discount.service.ts File in the src/app Folder

```
import { Injectable, Inject } from "@angular/core";
import { LogService } from "./log.service";

@Injectable()
export class DiscountService {
    private discountValue: number = 10;

    constructor(@Inject("logger") private logger: LogService) { }

    public get discount(): number {
        return this.discountValue;
    }

    public set discount(newValue: number) {
        this.discountValue = newValue ?? 0;
    }

    public applyDiscount(price: number) {
        this.logger.logInfoMessage(`Discount ${this.discount}` +
            ` applied to price: ${price}`);
        return Math.max(price - this.discountValue, 5);
    }
}
```

The `@Inject` decorator is applied to the constructor argument and used to specify the token that should be used to resolve the dependency. When Angular needs to create an instance of the `DiscountService` class, it will inspect the constructor and use the `@Inject` decorator argument to select the provider that will be used to resolve the dependency, resolving the dependency on the `LogService` class.

Using Opaque Tokens

When using simple types as provider tokens, there is a chance that two different parts of the application will try to use the same token to identify different services, which means that the wrong type of object may be used to resolve dependencies and cause errors.

To help work around this, Angular provides the `InjectionToken` class, which provides an object wrapper around a string value and can be used to create unique token values. In Listing 18-7, I have used the `InjectionToken` class to create a token that will be used to identify dependencies on the `LogService` class.

Listing 18-7. Using the `InjectionToken` Class in the log.service.ts File in the src/app Folder

```
import { Injectable, InjectionToken } from "@angular/core";

export const LOG_SERVICE = new InjectionToken("logger");

export enum LogLevel {
    DEBUG, INFO, ERROR
}
```

```

@Injectable()
export class LogService {
  minimumLevel: LogLevel = LogLevel.INFO;

  // ...methods omitted for brevity...
}

```

The constructor for the `InjectionToken` class accepts a string value that describes the service, but it is the `InjectionToken` object that will be the token. Dependencies must be declared on the same `InjectionToken` that is used to create the provider in the module; this is why the token has been created using the `const` keyword, which prevents the object from being modified. Listing 18-8 shows the provider configuration using the new token.

Listing 18-8. Creating a Provider Using an `InjectionToken` in the `app.module.ts` File in the `src/app` Folder

```

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoWay.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";
import { LogService, LOG_SERVICE } from "./log.service";

registerLocaleData(localeFr);

```

```

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
    PaDiscountPipe, PaDiscountAmountDirective],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [DiscountService, SimpleDataSource, Model,
    { provide: LOG_SERVICE, useClass: LogService }],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

Finally, Listing 18-9 shows the `DiscountService` class updated to declare a dependency using the `InjectionToken` instead of a string.

Listing 18-9. Declaring a Dependency in the `discount.service.ts` File in the `src/app` Folder

```

import { Injectable, Inject } from "@angular/core";
import { LogService, LOG_SERVICE } from "./log.service";

@Injectable()
export class DiscountService {
  private discountValue: number = 10;

  constructor( @Inject(LOG_SERVICE) private logger: LogService) { }

  public get discount(): number {
    return this.discountValue;
  }

  public set discount(newValue: number) {
    this.discountValue = newValue || 0;
  }

  public applyDiscount(price: number) {
    this.logger.logInfoMessage(`Discount ${this.discount}%
      + applied to price: ${price}`);
    return Math.max(price - this.discountValue, 5);
  }
}

```

There is no difference in the functionality offered by the application, but using the `InjectionToken` means that there will be no confusion between services.

Understanding the useClass Property

The class provider's `useClass` property specifies the class that will be instantiated to resolve dependencies. The provider can be configured with any class, which means you can change the implementation of a service by changing the provider configuration. This feature should be used with caution because the recipients of the service object will be expecting a specific type and a mismatch won't result in an error until the application is running in the browser. (TypeScript type enforcement has no effect on dependency injection because it occurs at runtime after the type annotations have been processed by the TypeScript compiler.)

The most common way to change classes is to use different subclasses. In Listing 18-10, I extended the `LogService` class to create a service that writes a different format of message in the browser's JavaScript console.

Listing 18-10. Creating a Subclassed Service in the `log.service.ts` File in the `src/app` Folder

```
import { Injectable, InjectionToken } from "@angular/core";

export const LOG_SERVICE = new InjectionToken("logger");

export enum LogLevel {
    DEBUG, INFO, ERROR
}

@Injectable()
export class LogService {
    minimumLevel: LogLevel = LogLevel.INFO;

    logInfoMessage(message: string) {
        this.logMessage(LogLevel.INFO, message);
    }

    logDebugMessage(message: string) {
        this.logMessage(LogLevel.DEBUG, message);
    }

    logErrorMessage(message: string) {
        this.logMessage(LogLevel.ERROR, message);
    }

    logMessage(level: LogLevel, message: string) {
        if (level >= this.minimumLevel) {
            console.log(`Message ${LogLevel[level]}: ${message}`);
        }
    }
}

@Injectable()
export class SpecialLogService extends LogService {

    constructor() {
        super()
        this.minimumLevel = LogLevel.DEBUG;
    }
}
```

```

    override logMessage(level: LogLevel, message: string) {
      if (level >= this.minimumLevel) {
        console.log(`Special Message ${LogLevel[level]}: ${message}`);
      }
    }
}

```

The SpecialLogService class extends LogService and provides its own implementation of the logMessage method. Listing 18-11 updates the provider configuration so that the useClass property specifies the new service.

Listing 18-11. Configuring the Provider in the app.module.ts File in the src/app Folder

```

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";
import { LogService, LOG_SERVICE, SpecialLogService } from "./log.service";

registerLocaleData(localeFr);

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,

```

```

ProductFormComponent, PaToggleView, PaAddTaxPipe,
PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
PaDiscountPipe, PaDiscountAmountDirective],
imports: [
  BrowserModule,
  BrowserAnimationsModule,
  FormsModule, ReactiveFormsModule
],
providers: [DiscountService, SimpleDataSource, Model,
  { provide: LOG_SERVICE, useClass: SpecialLogService }],
bootstrap: [ProductComponent]
})
export class AppModule { }

```

The combination of token and class means that dependencies on the LOG_SERVICE opaque token will be resolved using a SpecialLogService object. When you save the changes, you will see messages like this one displayed in the browser's JavaScript console, indicating that the derived service has been used:

Special Message (INFO): Discount 10 applied to price: 275

Care must be taken when setting the useClass property to specify a type that the dependent classes are expecting. Specifying a subclass is the safest option because the functionality of the base class is guaranteed to be available.

Resolving a Dependency with Multiple Objects

The class provider can be configured to deliver an array of objects to resolve a dependency, which can be useful if you want to provide a set of related services that differ in how they are configured. To provide an array, multiple class providers are configured using the same token and with the multi property set as true, as shown in Listing 18-12.

Listing 18-12. Configuring Multiple Service Objects in the app.module.ts File in the src/app Folder

```

...
@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
    PaDiscountPipe, PaDiscountAmountDirective],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
}
),

```

```

providers: [DiscountService, SimpleDataSource, Model,
  { provide: LOG_SERVICE, useClass: LogService, multi: true },
  { provide: LOG_SERVICE, useClass: SpecialLogService, multi: true }],
bootstrap: [ProductComponent]
})
...

```

The Angular dependency injection system will resolve dependencies on the LOG_SERVICE token by creating LogService and SpecialLogService objects, placing them in an array, and passing them to the dependent class's constructor. The class that receives the services must be expecting an array, as shown in Listing 18-13.

Listing 18-13. Receiving Multiple Services in the discount.service.ts File in the src/app Folder

```

import { Injectable, Inject } from "@angular/core";
import { LogService, LOG_SERVICE, LogLevel } from "./log.service";

@Injectable()
export class DiscountService {
  private discountValue: number = 10;
  private logger?: LogService;

  constructor( @Inject(LOG_SERVICE) loggers: LogService[] ) {
    this.logger = loggers.find(l => l.minimumLevel == LogLevel.DEBUG);
  }

  public get discount(): number {
    return this.discountValue;
  }

  public set discount(newValue: number) {
    this.discountValue = newValue ?? 0;
  }

  public applyDiscount(price: number) {
    this.logger?.logInfoMessage(`Discount ${this.discount}` +
      ` applied to price: ${price}`);
    return Math.max(price - this.discountValue, 5);
  }
}

```

The services are received as an array by the constructor, which uses the array's find method to locate the first logger whose minimumLevel property is LogLevel.Debug and assign it to the logger property. The applyDiscount method calls the service's logDebugMessage method, which results in messages like this one being displayed in the browser's JavaScript console:

Special Message (INFO): Discount 10 applied to price: 275

Using the Value Provider

The value provider is used when you want to take responsibility for creating the service objects yourself, rather than leaving it to the class provider. This can also be useful when services are simple types, such as string or number values, which can be a useful way of providing access to common configuration settings. The value provider can be applied using a literal object and supports the properties described in Table 18-5.

Table 18-5. The Value Provider Properties

Name	Description
provide	This property defines the service token, as described in the “Understanding the Token” section earlier in the chapter.
useValue	This property specifies the object that will be used to resolve the dependency.
multi	This property is used to allow multiple providers to be combined to provide an array of objects that will be used to resolve a dependency on the token. See the “Resolving a Dependency with Multiple Objects” section earlier in the chapter for an example.

The value provider works in the same way as the class provider except that it is configured with an object rather than a type. Listing 18-14 shows the use of the value provider to create an instance of the LogService class that is configured with a specific property value.

Listing 18-14. Using the Value Provider in the app.module.ts File in the src/app Folder

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
```

```

import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";
import { LogService, LOG_SERVICE, SpecialLogService, LogLevel } from "./log.service";

let logger = new LogService();
logger.minimumLevel = LogLevel.DEBUG;

registerLocaleData(localeFr);

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
    PaDiscountPipe, PaDiscountAmountDirective],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [DiscountService, SimpleDataSource, Model,
    { provide: LogService, useValue: logger }],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

This value provider is configured to resolve dependencies on the `LogService` token with a specific object that has been created and configured outside of the module class.

The value provider—and, in fact, all of the providers—can use any object as the token, as described in the previous section, but I have returned to using types as tokens because it is the most commonly used technique and because it works so nicely with TypeScript constructor parameter typing. Listing 18-15 shows the corresponding change to the `DiscountService`, which declares a dependency using a typed constructor argument.

Listing 18-15. Declaring a Dependency Using a Type in the `discount.service.ts` File in the `src/app` Folder

```

import { Injectable, Inject } from "@angular/core";
import { LogService, LOG_SERVICE, LogLevel } from "./log.service";

@Injectable()
export class DiscountService {
  private discountValue: number = 10;
  //private logger?: LogService;

  constructor(private logger: LogService) { }

  public get discount(): number {
    return this.discountValue;
  }

```

```

public set discount(newValue: number) {
    this.discountValue = newValue ?? 0;
}

public applyDiscount(price: number) {
    this.logger?.logInfoMessage(`Discount ${this.discount}`
        + ` applied to price: ${price}`);
    return Math.max(price - this.discountValue, 5);
}
}

```

Using the Factory Provider

The factory provider uses a function to create the object required to resolve a dependency. This provider supports the properties described in Table 18-6.

Table 18-6. The Factory Provider Properties

Name	Description
provide	This property defines the service token, as described in the “Understanding the Token” section earlier in the chapter.
deps	This property specifies an array of provider tokens that will be resolved and passed to the function specified by the useFactory property.
useFactory	This property specifies the function that will create the service object. The objects produced by resolving the tokens specified by the deps property will be passed to the function as arguments. The result returned by the function will be used as the service object.
multi	This property is used to allow multiple providers to be combined to provide an array of objects that will be used to resolve a dependency on the token. See the “Resolving a Dependency with Multiple Objects” section earlier in the chapter for an example.

This is the provider that gives the most flexibility in how service objects are created because you can define functions that are tailored to your application’s requirements. Listing 18-16 shows a factory function that creates LogService objects.

Listing 18-16. Using the Factory Provider in the app.module.ts File in the src/app Folder

```

...
@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
    PaDiscountPipe, PaDiscountAmountDirective],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
}

```

```

providers: [DiscountService, SimpleDataSource, Model,
{
  provide: LogService, useFactory: () => {
    let logger = new LogService();
    logger.minimumLevel = LogLevel.DEBUG;
    return logger;
  }
},
bootstrap: [ProductComponent]
})
...

```

The function in this example is simple: it receives no arguments and just creates a new LogService object. The real flexibility of this provider comes when the deps property is used, which allows for dependencies to be created on other services. In Listing 18-17, I have defined a token that specifies a debugging level.

Listing 18-17. Defining a Logging-Level Service in the log.service.ts File in the src/app Folder

```

import { Injectable, InjectionToken } from "@angular/core";

export const LOG_SERVICE = new InjectionToken("logger");
export const LOG_LEVEL = new InjectionToken("log_level");

export enum LogLevel {
  DEBUG, INFO, ERROR
}

@Injectable()
export class LogService {
  minimumLevel: LogLevel = LogLevel.INFO;

  // ...methods omitted for brevity...
}

@Injectable()
export class SpecialLogService extends LogService {

  // ...methods omitted for brevity...
}

```

In Listing 18-18, I have defined a value provider that creates a service using the LOG_LEVEL token and used that service in the factory function that creates the LogService object.

Listing 18-18. Using Factory Dependencies in the app.module.ts File in the src/app Folder

```

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

```

```

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";
import { LogService, LOG_SERVICE, SpecialLogService,
LogLevel, LOG_LEVEL} from "./log.service";

let logger = new LogService();
logger.minimumLevel = LogLevel.DEBUG;

registerLocaleData(localeFr);

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
    PaDiscountPipe, PaDiscountAmountDirective],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [DiscountService, SimpleDataSource, Model,
    { provide: LOG_LEVEL, useValue: LogLevel.DEBUG },
    { provide: LogService,
      deps: [LOG_LEVEL],
      useFactory: (level: LogLevel) => {
        let logger = new LogService();

```

```

    logger.minimumLevel = level;
    return logger;
  }],
  bootstrap: [ProductComponent]
)
export class AppModule { }

```

The LOG_LEVEL token is used by a value provider to define a simple value as a service. The factory provider specifies this token in its deps array, which the dependency injection system resolves and provides as an argument to the factory function, which uses it to set the minimumLevel property of a new LogService object.

Using the Existing Service Provider

This provider is used to create aliases for services so they can be targeted using more than one token, using the properties described in Table 18-7.

Table 18-7. The Existing Provider Properties

Name	Description
provide	This property defines the service token, as described in the “Understanding the Token” section earlier in the chapter.
useExisting	This property is used to specify the token of another provider, whose service object will be used to resolve dependencies on this service.
multi	This property is used to allow multiple providers to be combined to provide an array of objects that will be used to resolve a dependency on the token. See the “Resolving a Dependency with Multiple Objects” section earlier in the chapter for an example.

This provider can be useful when you want to refactor the set of providers but don’t want to eliminate all the obsolete tokens to avoid refactoring the rest of the application. Listing 18-19 shows the use of this provider.

Listing 18-19. Creating a Service Alias in the app.module.ts File in the src/app Folder

```

...
@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
    PaDiscountPipe, PaDiscountAmountDirective],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],

```

```

providers: [DiscountService, SimpleDataSource, Model,
  { provide: LOG_LEVEL, useValue: LogLevel.DEBUG },
  { provide: "debugLevel", useExisting: LOG_LEVEL },
  { provide: LogService,
    deps: ["debugLevel"],
    useFactory: (level: LogLevel) => {
      let logger = new LogService();
      logger.minimumLevel = level;
      return logger;
    })
  ],
bootstrap: [ProductComponent]
})
...

```

The token for the new service is the string `debugLevel`, and it is aliased to the provider with the `LOG_LEVEL` token. Using either token will result in the dependency being resolved with the same value.

Using Local Providers

When Angular creates a new instance of a class, it resolves any dependencies using an *injector*. It is an injector that is responsible for inspecting the constructors of classes to determine what dependencies have been declared and resolving them using the available providers.

So far, all the dependency injection examples have relied on providers configured in the application's Angular module. But the Angular dependency injection system is more complex: there is a hierarchy of injectors corresponding to the application's tree of components and directives. Each component and directive can have its own injector, and each injector can be configured with its own set of providers, known as *local providers*.

When there is a dependency to resolve, Angular uses the injector for the nearest component or directive. The injector first tries to resolve the dependency using its own set of local providers. If no local providers have been set up or there are no providers that can be used to resolve this specific dependency, then the injector consults the parent component's injector. The process is repeated—the parent component's injector tries to resolve the dependency using its own set of local providers. If a suitable provider is available, then it is used to provide the service object required to resolve the dependency. If there is no suitable provider, then the request is passed up to the next level in the hierarchy to the grandparent of the original injector. At the top of the hierarchy is the root Angular module, whose providers are the last resort before reporting an error.

Defining providers in the Angular module means that all dependencies for a token within the application will be resolved using the same object. As I explain in the following sections, registering providers further down the injector hierarchy can change this behavior and alter the way that services are created and used.

Understanding the Limitations of Single Service Objects

Using a single service object can be a powerful tool, allowing building blocks in different parts of the application to share data and respond to user interactions. But some services don't lend themselves to being shared so widely. As a simple example, Listing 18-20 adds a dependency on `LogService` to one of the pipes created in Chapter 16.

Listing 18-20. Adding a Service Dependency in the discount.pipe.ts File in the src/app Folder

```
import { Pipe, Injectable } from "@angular/core";
import { DiscountService } from "./discount.service";
import { LogService } from "./log.service";

@Pipe({
  name: "discount",
  pure: false
})
export class PaDiscountPipe {

  constructor(private discount: DiscountService,
    private logger: LogService) { }

  transform(price: number): number {
    if (price > 100) {
      this.logger.logInfoMessage(`Large price discounted: ${price}`);
    }
    return this.discount.applyDiscount(price);
  }
}
```

The pipe's transform method uses the LogService object, which is received as a constructor argument, to generate logging messages when the price value it transforms is greater than 100.

The problem is that these log messages are drowned out by the messages generated by the DiscountService object, which creates a message every time a discount is applied. The obvious thing to do is to change the minimum level in the LogService object when it is created by the module provider's factory function, as shown in Listing 18-21.

Listing 18-21. Changing the Logging Level in the app.module.ts File in the src/app Folder

```
...
@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
    PaDiscountPipe, PaDiscountAmountDirective],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [DiscountService, SimpleDataSource, Model,
    { provide: LOG_LEVEL, useValue: LogLevel.ERROR },
    { provide: "debugLevel", useExisting: LOG_LEVEL },
    { provide: LogService,
      deps: ["debugLevel"],
      useFactory: (level: LogLevel) => {
        let logger = new LogService();
```

```

        logger.minimumLevel = level;
        return logger;
    }],
    bootstrap: [ProductComponent]
})
...

```

Of course, this doesn't have the desired effect because the same LogService object is used throughout the application and filtering the DiscountService messages means that the pipe messages are filtered too.

I could enhance the LogService class so there are different filters for each source of logging messages, but that quickly becomes complicated. Instead, I am going to solve the problem by creating a local provider so that there are multiple LogService objects in the application, each of which can then be configured separately.

Creating Local Providers in a Component

Components can define local providers, which allow separate servers to be created and used by part of the application. Components support two decorator properties for creating local providers, as described in Table 18-8.

Table 18-8. The Component Decorator Properties for Local Providers

Name	Description
providers	This property is used to create a provider used to resolve dependencies of view and content children.
viewProviders	This property is used to create a provider used to resolve dependencies of view children.

The simplest way to address my LogService issue is to use the providers property to set up a local provider, as shown in Listing 18-22.

Listing 18-22. Creating a Local Provider in the productTable.component.ts File in the src/app Folder

```

import { Component, Input, QueryList, ViewChildren } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
//import { Subject } from "rxjs";
import { DiscountService } from "./discount.service";
import { LogService } from "./log.service";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html",
  providers:[LogService]
})
export class ProductTableComponent {

  constructor(private dataModel: Model) { }

  // @Input("model")

```

```
// dataModel: Model | undefined;

getProduct(key: number): Product | undefined {
  return this.dataModel?.getProduct(key);
}

getProducts(): Product[] | undefined {
  return this.dataModel?.getProducts();
}

deleteProduct(key: number) {
  this.dataModel?.deleteProduct(key);
}

taxRate: number = 0;
categoryFilter: string | undefined;
itemCount: number = 3;
}
```

When Angular needs to create a new pipe object, it detects the dependency on `LogService` and starts working its way up the application hierarchy, examining each component it finds to determine whether they have a provider that can be used to resolve the dependency. The `ProductTableComponent` does have a `LogService` provider, which is used to create the service used to resolve the pipe's dependency. This means there are now two `LogService` objects in the application, each of which can be configured separately, as shown in Figure 18-2.

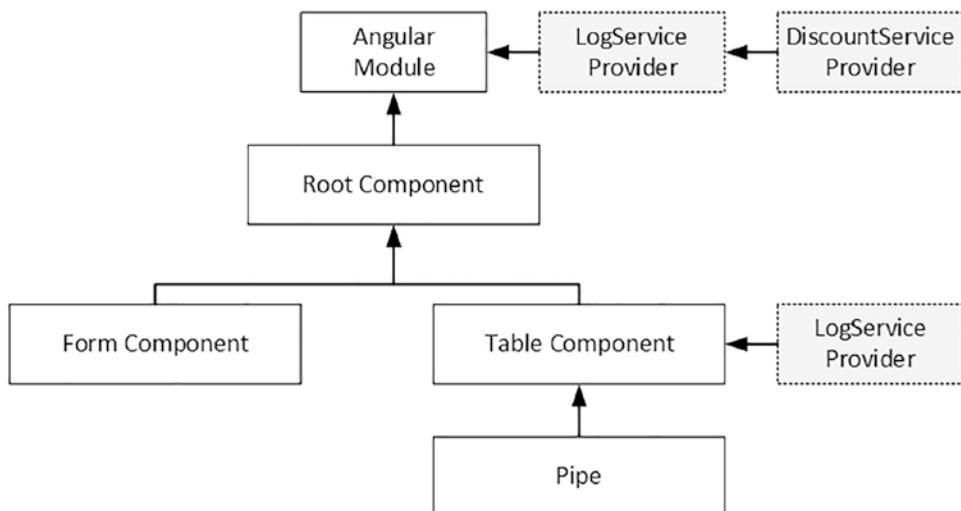


Figure 18-2. Creating a local provider

The `LogService` object created by the component's provider uses the default value for its `minimumLevel` property and will display `LogLevel.INFO` messages. The `LogService` object created by the module, which will be used to resolve all other dependencies in the application, including the one declared by the `DiscountService` class, is configured so that it will display only `LogLevel.ERROR` messages. When you

save the changes, you will see the logging messages from the pipe (which receives the service from the component) but not from `DiscountService` (which receives the service from the module).

Understanding the Provider Alternatives

As described in Table 18-8, there are two properties that can be used to create local providers. To demonstrate how these properties differ, I added a file called `valueDisplay.directive.ts` to the `src/app` folder and used it to define the directive shown in Listing 18-23.

Listing 18-23. The Contents of the `valueDisplay.directive.ts` File in the `src/app` Folder

```
import { Directive, InjectionToken, Inject, HostBinding} from "@angular/core";
export const VALUE_SERVICE = new InjectionToken("value_service");

@Directive({
  selector: "[paDisplayValue]"
})
export class PaDisplayValueDirective {

  constructor( @Inject(VALUE_SERVICE) serviceValue: string) {
    this.elementContent = serviceValue;
  }

  @HostBinding("textContent")
  elementContent: string;
}
```

The `VALUE_SERVICE` opaque token will be used to define a value-based service, on which the directive in this listing declares a dependency so that it can be displayed in the host element's content. Listing 18-24 shows the service being defined and the directive being registered in the Angular module. I have also simplified the `LogService` provider in the module for brevity.

Listing 18-24. Registering the Directive and Service in the `app.module.ts` File in the `src/app` Folder

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
```

```

import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";
import { LogService, LOG_SERVICE, SpecialLogService,
    LogLevel, LOG_LEVEL} from "./log.service";
import { VALUE_SERVICE, PaDisplayValueDirective} from "./valueDisplay.directive";

let logger = new LogService();
logger.minimumLevel = LogLevel.DEBUG;

registerLocaleData(localeFr);

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
    PaDiscountPipe, PaDiscountAmountDirective, PaDisplayValueDirective],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [DiscountService, SimpleDataSource, Model, LogService,
    { provide: VALUE_SERVICE, useValue: "Apples" }],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

The provider sets up a value of Apples for the VALUE_SERVICE service. The next step is to apply the new directive so there is an instance that is a view child of a component and another that is a content child. Listing 18-25 sets up the content child instance.

Listing 18-25. Applying a Content Child Directive in the template.html File in the src/app Folder

```

<div class="container-fluid angularApp">
  <div class="row p-2">
    <div class="col-4 p-2 text-dark">
```

```

<pa-productform>
    <span paDisplayValue></span>
</pa-productform>
</div>
<div class="col p-2">
    <paProductTable></paProductTable>
</div>
</div>
</div>

```

Listing 18-26 projects the host element's content and adds a view child instance of the new directive.

Listing 18-26. Adding Directives in the productForm.component.html File in the src/app Folder

```

<form #form="ngForm" (ngSubmit)="submitForm(form)">
    <div class="form-group">
        <label>Name</label>
        <input class="form-control"
            name="name" [(ngModel)]="newProduct.name" />
    </div>
    <div class="form-group">
        <label>Category</label>
        <input class="form-control"
            name="category" [(ngModel)]="newProduct.category" />
    </div>
    <div class="form-group">
        <label>Price</label>
        <input class="form-control"
            name="name" [(ngModel)]="newProduct.price" />
    </div>
    <button class="btn btn-primary mt-2" type="submit">
        Create
    </button>
</form>

<div class="bg-info text-white m-2 p-2">
    View Child Value: <span paDisplayValue></span>
</div>
<div class="bg-info text-white m-2 p-2">
    Content Child Value: <ng-content></ng-content>
</div>

```

When you save the changes, you will see the new elements, as shown in Figure 18-3, both of which show the same value because the only provider for VALUE_SERVICE is defined in the module.

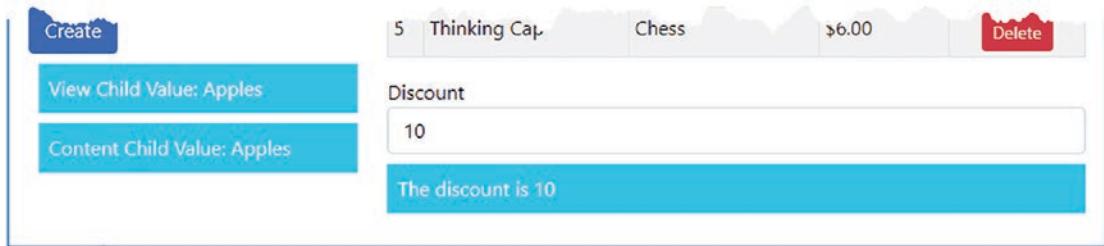


Figure 18-3. View and content child directives

Creating a Local Provider for All Children

The `@Component` decorator's `providers` property is used to define providers that will be used to resolve service dependencies for all children, regardless of whether they are defined in the template (view children) or projected from the host element (content children). Listing 18-27 defines a `VALUE_SERVICE` provider in the parent component for two new directive instances.

Listing 18-27. Defining a Provider in the `productForm.component.ts` File in the `src/app` Folder

```
import { Component, Output, EventEmitter, ViewEncapsulation } from "@angular/core";
import { Product } from "./product.model";
import { Model } from "./repository.model";
import { VALUE_SERVICE } from "./valueDisplay.directive";

@Component({
  selector: "pa-productform",
  templateUrl: "productForm.component.html",
  providers: [{ provide: VALUE_SERVICE, useValue: "Oranges" }]
})
export class ProductFormComponent {
  newProduct: Product = new Product();

  constructor(private model: Model) { }

  submitForm(form: any) {
    this.model.saveProduct(this.newProduct);
    this.newProduct = new Product();
    form.resetForm();
  }
}
```

The new provider changes the service value. When Angular comes to create the instances of the new directive, it begins its search for providers by working its way up the application hierarchy and finds the `VALUE_SERVICE` provider defined in Listing 18-27. The service value is used by both instances of the directive, as shown in Figure 18-4.

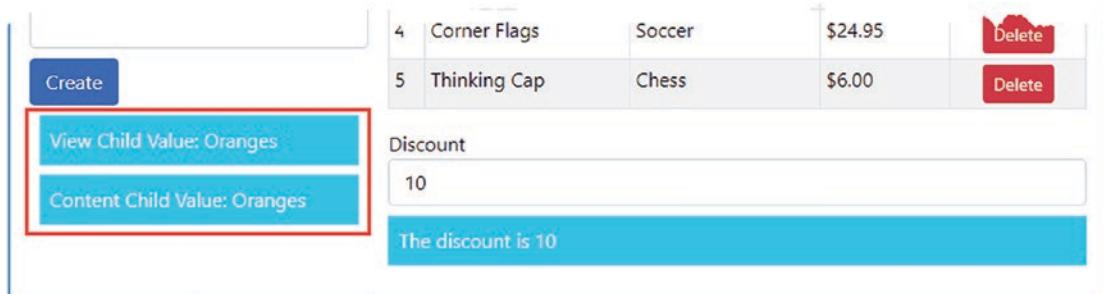


Figure 18-4. Defining a provider for all children in a component

Creating a Provider for View Children

The `viewProviders` property defines providers that are used to resolve dependencies for view children but not content children. Listing 18-28 uses the `viewProviders` property to define a provider for `VALUE_SERVICE`.

Listing 18-28. Defining a View Child Provider in the `productForm.component.ts` File in the `src/app` Folder

```
import { Component, Output, EventEmitter, ViewEncapsulation } from "@angular/core";
import { Product } from "./product.model";
import { Model } from "./repository.model";
import { VALUE_SERVICE } from "./valueDisplay.directive";

@Component({
  selector: "pa-productform",
  templateUrl: "productForm.component.html",
  viewProviders: [{ provide: VALUE_SERVICE, useValue: "Oranges" }]
})
export class ProductFormComponent {
  newProduct: Product = new Product();

  constructor(private model: Model) { }

  submitForm(form: any) {
    this.model.saveProduct(this.newProduct);
    this.newProduct = new Product();
    form.resetForm();
  }
}
```

```

    }
}

```

Angular uses the provider when resolving dependencies for view children but not for content children. This means dependencies for content children are referred up the application's hierarchy as though the component had not defined a provider. In the example, this means that the view child will receive the service created by the component's provider, and the content child will receive the service created by the module's provider, as shown in Figure 18-5.

Caution Defining providers for the same service using both the providers and viewProviders properties is not supported. If you do this, the view and content children both will receive the service created by the viewProviders provider.

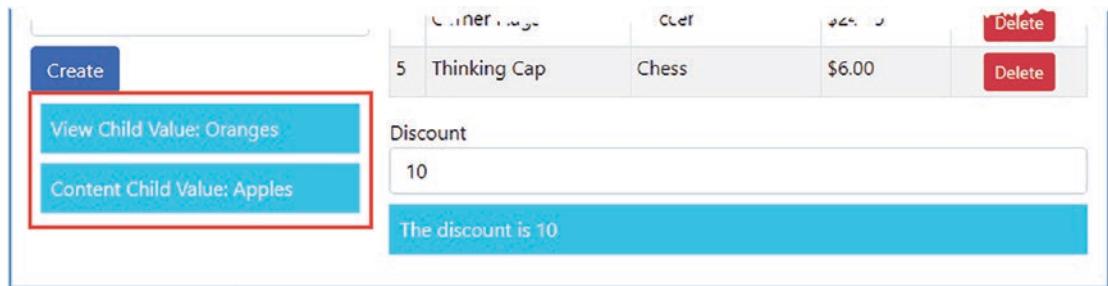


Figure 18-5. Defining a provider for view children

Controlling Dependency Resolution

Angular provides three decorators that can be used to provide instructions about how a dependency is resolved. These decorators are described in Table 18-9 and demonstrated in the following sections.

Table 18-9. The Dependency Resolution Decorators

Name	Description
@Host	This decorator restricts the search for a provider to the nearest component.
@Optional	This decorator stops Angular from reporting an error if the dependency cannot be resolved.
@SkipSelf	This decorator excludes the providers defined by the component/directive whose dependency is being resolved.

Restricting the Provider Search

The `@Host` decorator restricts the search for a suitable provider so that it stops once the closest component has been reached. The decorator is typically combined with `@Optional`, which prevents Angular from throwing an exception if a service dependency cannot be resolved. Listing 18-29 shows the addition of both decorators to the directive in the example.

Listing 18-29. Adding Dependency Decorators in the valueDisplay.directive.ts File in the src/app Folder

```
import { Directive, InjectionToken, Inject,
    HostBinding, Host, Optional } from "@angular/core";

export const VALUE_SERVICE = new InjectionToken("value_service");

@Directive({
    selector: "[paDisplayValue]"
})
export class PaDisplayValueDirective {

    constructor( @Inject(VALUE_SERVICE) @Host() @Optional() serviceValue: string ) {
        this.elementContent = serviceValue || "No Value";
    }

    @HostBinding("textContent")
    elementContent: string;
}
```

When using the `@Optional` decorator, you must ensure that the class is able to function if the service cannot be resolved, in which case the constructor argument for the service is `undefined`. The nearest component defines a service for its view children but not content children, which means that one instance of the directive will receive a service object and the other will not, as illustrated in Figure 18-6.

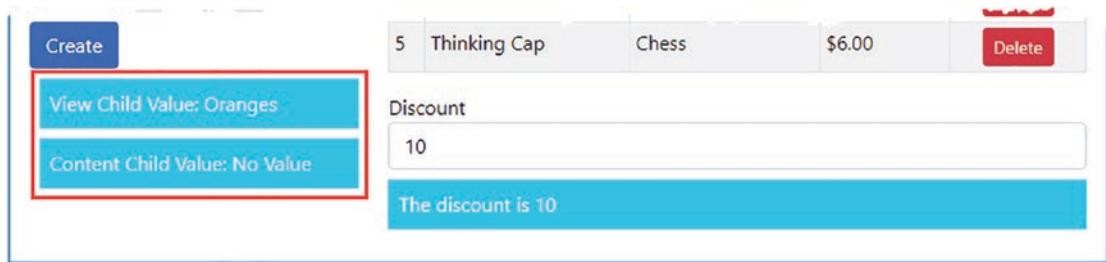


Figure 18-6. Controlling how a dependency is resolved

Skiping Self-Defined Providers

By default, the providers defined by a component are used to resolve its dependencies. The `@SkipSelf` decorator can be applied to constructor arguments to tell Angular to ignore the local providers and start the search at the next level in the application hierarchy, which means that the local providers will be used only

to resolve dependencies for children. In Listing 18-30, I have added a dependency on the VALUE_SERVICE provider that is decorated with @SkipSelf.

Listing 18-30. Skipping Local Providers in the productForm.component.ts File in the src/app Folder

```
import { Component, Output, EventEmitter, ViewEncapsulation,
  Inject, SkipSelf } from "@angular/core";
import { Product } from "./product.model";
import { Model } from "./repository.model";
import { VALUE_SERVICE } from "./valueDisplay.directive";

@Component({
  selector: "pa-productform",
  templateUrl: "productForm.component.html",
  viewProviders: [{ provide: VALUE_SERVICE, useValue: "Oranges" }]
})
export class ProductFormComponent {
  newProduct: Product = new Product();

  constructor(private model: Model,
    @Inject(VALUE_SERVICE) @SkipSelf() private serviceValue: string) {
    console.log("Service Value: " + serviceValue);
  }

  submitForm(form: any) {
    this.model.saveProduct(this.newProduct);
    this.newProduct = new Product();
    form.resetForm();
  }
}
```

When you save the changes and the browser reloads the page, you will see the following message in the browser's JavaScript console, showing that the service value defined locally (Oranges) has been skipped and allowing the dependency to be resolved by the Angular module:

Service Value: Apples

Summary

In this chapter, I explained the role that providers play in dependency injection and explained how they can be used to change how services are used to resolve dependencies. I described the different types of providers that can be used to create service objects and demonstrated how directives and components can define their own providers to resolve their own dependencies and those of their children. In the next chapter, I describe modules, which are the final building block for Angular applications.

CHAPTER 19



Using and Creating Modules

In this chapter, I describe the last of the Angular building blocks: modules. In the first part of the chapter, I describe the root module, which every Angular application uses to describe the configuration of the application to Angular. In the second part of the chapter, I describe feature modules, which are used to add structure to an application so that related features can be grouped as a single unit. Table 19-1 puts modules in context.

Table 19-1. Putting Modules in Context

Question	Answer
What are they?	Modules provide configuration information to Angular.
Why are they useful?	The root module describes the application to Angular, setting up essential features such as components and services. Feature modules are useful for adding structure to complex projects, which makes them easier to manage and maintain.
How are they used?	Modules are classes to which the <code>@NgModule</code> decorator has been applied. The properties used by the decorator have different meanings for root and feature modules.
Are there any pitfalls or limitations?	There is no module-wide scope for providers, which means that the providers defined by a feature module will be available as though they had been defined by the root module.
Are there any alternatives?	Every application must have a root module, but the use of feature modules is entirely optional. However, if you don't use feature modules, then the files in an application can become difficult to manage.

Table 19-2 summarizes the chapter.

Table 19-2. Chapter Summary

Problem	Solution	Listing
Describing an application and the building blocks it contains	Use the root module	1-7
Grouping related features together	Create a feature module	8-28

Preparing the Example Project

As with the other chapters in this part of the book, I am going to use the example project that was created in Chapter 9 and has been expanded and extended in each chapter since.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

To prepare for this chapter, I have removed some functionality from the component templates. Listing 19-1 shows the template for the product table, in which I have commented out the elements for the discount editor and display components.

Listing 19-1. The Contents of the productTable.component.html File in the src/app Folder

```
<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of getProducts(); let i = index">
      <td>{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td [pa-price]="item.price" #discount="discount">
        {{ discount.discountAmount | currency:"USD": "symbol" }}</td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm"
               (click)="deleteProduct(item.id)">
          Delete
        </button>
      </td>
    </tr>
  </tbody>
</table>

<!-- <paDiscountEditor></paDiscountEditor> -->
<!-- <paDiscountDisplay></paDiscountDisplay> -->
```

Listing 19-2 shows the template from the product form component, in which I have commented out the elements that I used to demonstrate the difference between providers for view children and content children in Chapter 18.

Listing 19-2. The Contents of the productForm.component.html File in the src/app Folder

```
<form #form="ngForm" (ngSubmit)="submitForm(form)">
  <div class="form-group">
    <label>Name</label>
    <input class="form-control"
```

```

        name="name" [(ngModel)]="newProduct.name" />
    </div>
    <div class="form-group">
        <label>Category</label>
        <input class="form-control"
            name="category" [(ngModel)]="newProduct.category" />
    </div>
    <div class="form-group">
        <label>Price</label>
        <input class="form-control"
            name="name" [(ngModel)]="newProduct.price" />
    </div>
    <button class="btn btn-primary mt-2" type="submit">
        Create
    </button>
</form>

<!-- <div class="bg-info text-white m-2 p-2">
    View Child Value: <span>paDisplayValue</span>
</div>
<div class="bg-info text-white m-2 p-2">
    Content Child Value: <ng-content></ng-content>
</div> -->

```

Run the following command in the example folder to start the Angular development tools:

```
ng serve
```

Open a new browser window and navigate to <http://localhost:4200> to see the content shown in Figure 19-1.

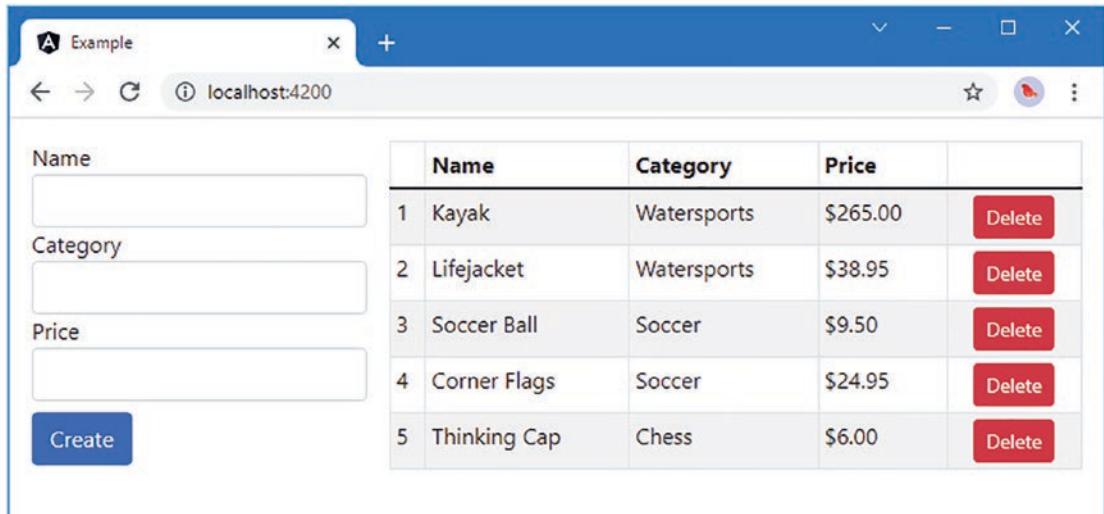


Figure 19-1. Running the example application

Understanding the Root Module

Every Angular has at least one module, known as the *root module*. The root module is conventionally defined in a file called `app.module.ts` in the `src/app` folder, and it contains a class to which the `@NgModule` decorator has been applied. Listing 19-3 shows the root module from the example application.

Listing 19-3. The Root Module in the `app.module.ts` File in the `src/app` Folder

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoWay.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";
import { LogService, LOG_SERVICE, SpecialLogService,
    LogLevel, LOG_LEVEL} from "./log.service";
import { VALUE_SERVICE, PaDisplayValueDirective} from "./valueDisplay.directive";

let logger = new LogService();
logger.minimumLevel = LogLevel.DEBUG;

registerLocaleData(localeFr);
```

```

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
    PaDiscountPipe, PaDiscountAmountDirective, PaDisplayValueDirective],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [DiscountService, SimpleDataSource, Model, LogService,
    { provide: VALUE_SERVICE, useValue: "Apples" }],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

There can be multiple modules in a project, but the root module is the one used in the bootstrap file, which is conventionally called `main.ts` and is defined in the `src` folder. Listing 19-4 shows the `main.ts` file for the example project.

Listing 19-4. The Angular Bootstrap in the `main.ts` File in the `src` Folder

```

import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));

```

Angular applications can be run in different environments, such as web browsers and native application containers. The job of the bootstrap file is to select the platform and identify the root module. The `platformBrowserDynamic` method creates the browser runtime, and the `bootstrapModule` method is used to specify the module, which is the `AppModule` class from Listing 19-3.

When defining the root module, the `@NgModule` decorator properties described in Table 19-3 are used. (There are additional decorator properties, which are described later in the chapter.)

Table 19-3. The `@NgModule` Decorator Root Module Properties

Name	Description
imports	This property specifies the Angular modules that are required to support the directives, components, and pipes in the application.
declarations	This property is used to specify the directives, components, and pipes that are used in the application.
providers	This property defines the service providers that will be used by the module's injector. These are the providers that will be available throughout the application and used when no local provider for a service is available, as described in Chapter 18.
bootstrap	This property specifies the root components for the application.

Understanding the imports Property

The `imports` property is used to list the other modules that the application requires. In the example application, these are all modules provided by the Angular framework.

```
...
imports: [
  BrowserModule,
  BrowserAnimationsModule,
  FormsModule, ReactiveFormsModule
],
...
```

The `BrowserModule` provides the functionality required to run Angular applications in web browsers. The `BrowserAnimationsModule` module was added to the project by the Angular Material package and enables the animation features described in Chapter 27. The other two modules provide support for working with HTML forms. There are other Angular modules, which are introduced in later chapters.

The `imports` property is also used to declare dependencies on custom modules, which are used to manage complex Angular applications and to create units of reusable functionality. I explain how custom modules are defined in the “Creating Feature Modules” section.

Understanding the declarations Property

The `declarations` property is used to provide Angular with a list of the directives, components, and pipes that the application requires, known collectively as the *declarable classes*. The `declarations` property in the example project root module contains a long list of classes, each of which is available for use elsewhere in the application only because it is listed here.

```
...
declarations: [ProductComponent, PaAttrDirective, PaModel,
  PaStructureDirective, PaIteratorDirective,
  PaCellColor, PaCellColorSwitcher, ProductTableComponent,
  ProductFormComponent, PaToggleView, PaAddTaxPipe,
  PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
  PaDiscountPipe, PaDiscountAmountDirective, PaDisplayValueDirective],
```

...

Notice that the built-in declarable classes, such as the directives described in Chapter 11 and the pipes described in Chapter 16, are not included in the declarations property for the root module. This is because they are part of the `BrowserModule` module, and when you add a module to the imports property, its declarable classes are automatically available for use in the application.

Understanding the providers Property

The providers property is used to define the service providers that will be used to resolve dependencies when there are no suitable local providers available. The use of providers for services is described in detail in Chapters 17 and 19.

Understanding the bootstrap Property

The bootstrap property specifies the root component or components for the application. When Angular processes the main HTML document, which is conventionally called `index.html`, it inspects the root components and applies them using the value of the selector property in the `@Component` decorators.

Tip The components listed in the bootstrap property must also be included in the declarations list.

Here is the bootstrap property from the example project's root module:

```
...
bootstrap: [ProductComponent]
...
```

The `ProductComponent` class provides the root component, and its selector property specifies the `app` element, as shown in Listing 19-5.

Listing 19-5. The Root Component in the `component.ts` File in the `src/app` Folder

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

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
```

When I started the example project in Chapter 9, the root component had a lot of functionality. But since the introduction of additional components, the role of this component has been reduced, and it is now essentially a placeholder that tells Angular to project the contents of the `app/template.html` file into the `app` element in the HTML document, which allows the components that do the real work in the application to be loaded.

There is nothing wrong with this approach, but it does mean the root component in the application doesn't have a great deal to do. If this kind of redundancy feels untidy, then you can specify multiple root components in the root module, and all of them will be used to target elements in the HTML document. To demonstrate, I have removed the existing root component from the root module's `bootstrap` property and replaced it with the component classes that are responsible for the product form and the product table, as shown in Listing 19-6.

Listing 19-6. Specifying Multiple Root Components in the app.module.ts File in the src/app Folder

```
...
@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
    PaDiscountPipe, PaDiscountAmountDirective, PaDisplayValueDirective],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [DiscountService, SimpleDataSource, Model, LogService,
    { provide: VALUE_SERVICE, useValue: "Apples" }],
  bootstrap: [ProductFormComponent, ProductTableComponent]
})
export class AppModule { }
```

...

Listing 19-7 reflects the change in the root components in the main HTML document. The inconsistent element names are from an earlier chapter, where I changed the selector for the form component to demonstrate the use of the shadow DOM feature.

Listing 19-7. Changing the Root Component Elements in the index.html File in the src Folder

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Example</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link rel="preconnect" href="https://fonts.gstatic.com">
  <link
    href="https://fonts.googleapis.com/css2?family=Roboto:wght@300;400;500&display=swap"
    rel="stylesheet">
  <link href="https://fonts.googleapis.com/icon?family=Material+Icons"
    rel="stylesheet">
</head>
<body class="container-fluid">
  <div class="row">
```

```

<div class="col-8 p-2">
  <paProductTable></paProductTable>
</div>
<div class="col-4 p-2">
  <pa-productform></pa-productform>
</div>
</div>
</body>
</html>

```

I have reversed the order in which these components appear compared to previous examples, just to create a detectable change in the application's layout. When all the changes are saved and the browser has reloaded the page, you will see the new root components displayed, as illustrated by Figure 19-2.

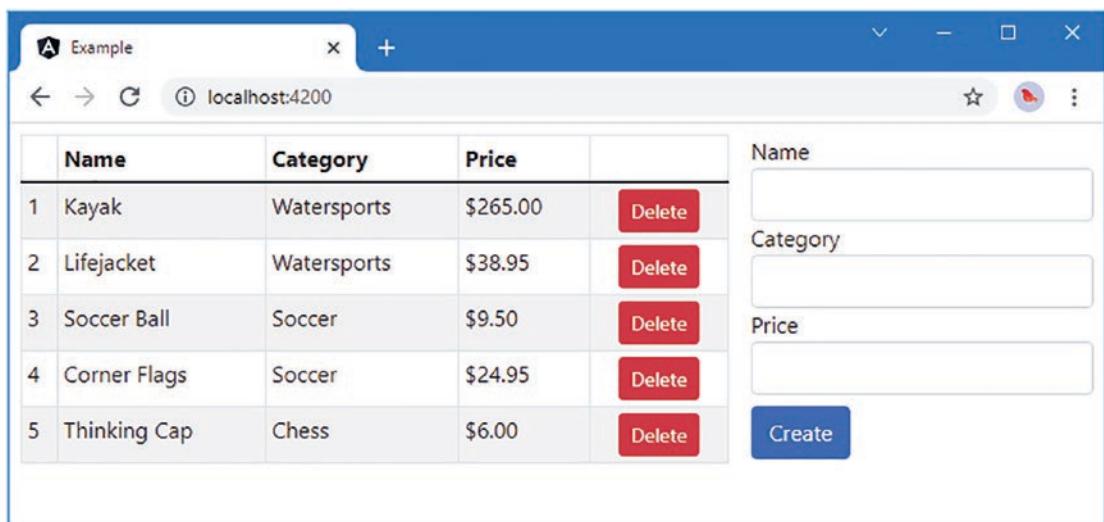


Figure 19-2. Using multiple root components

The module's service providers are used to resolve dependencies for all root components. In the case of the example application, this means there is a single Model service object that is shared throughout the application and that allows products created with the HTML form to be displayed automatically in the table, even though these components have been promoted to be root components.

Creating Feature Modules

The root module has become increasingly complex as I added features in earlier chapters, with a long list of import statements to load JavaScript modules and a set of classes in the declarations property of the @NgModule decorator that spans several lines, as shown in Listing 19-8.

Listing 19-8. The Contents of the app.module.ts File in the src/app Folder

```

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

```

```

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoWay.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";
import { LogService, LOG_SERVICE, SpecialLogService,
    LogLevel, LOG_LEVEL} from "./log.service";
import { VALUE_SERVICE, PaDisplayValueDirective} from "./valueDisplay.directive";

let logger = new LogService();
logger.minimumLevel = LogLevel.DEBUG;

registerLocaleData(localeFr);

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
    PaDiscountPipe, PaDiscountAmountDirective, PaDisplayValueDirective],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [DiscountService, SimpleDataSource, Model, LogService,
    { provide: VALUE_SERVICE, useValue: "Apples" }]
})

```

```

    bootstrap: [ProductFormComponent, ProductTableComponent]
})
export class AppModule { }

```

Feature modules are used to group related functionality so that it can be used as a single entity, just like the Angular modules such as `BrowserModule`. When I need to use the features for working with forms, for example, I don't have to add `import` statements and declarations entries for each individual directive, component, or pipe. Instead, I just add `BrowserModule` to the decorator's `imports` property, and all of the functionality it contains is available throughout the application.

When you create a feature module, you can choose to focus on an application function or elect to group a set of related building blocks that provide your application's infrastructure. I'll do both in the sections that follow because they work in slightly different ways and have different considerations. Feature modules use the same `@NgModule` decorator but with an overlapping set of configuration properties, some of which are new and some of which are used in common with the root module but have a different effect. I explain how these properties are used in the following sections, but Table 19-4 provides a summary for quick reference.

Table 19-4. The `@NgModule` Decorator Properties for Feature Modules

Name	Description
<code>imports</code>	This property is used to import the modules that are required by the classes in the modules.
<code>providers</code>	This property is used to define the module's providers. When the feature module is loaded, the set of providers is combined with those in the root module, which means that the feature module's services are available throughout the application (and not just within the module).
<code>declarations</code>	This property is used to specify the directives, components, and pipes in the module. This property must contain the classes that are used within the module and those that are exposed by the module to the rest of the application.
<code>exports</code>	This property is used to define the public exports from the module. It contains some or all of the directives, components, and pipes from the <code>declarations</code> property and some or all of the modules from the <code>imports</code> property.

Creating a Model Module

The term *model module* might be a tongue twister, but it is generally a good place to start when refactoring an application using feature modules because just about every other building block in the application depends on the model.

The first step is to create the folder that will contain the module. Module folders are defined within the `src/app` folder and are given a meaningful name. For this module, I created an `src/app/model` folder by running the following command in the example folder:

```
mkdir src/app/model
```

The naming conventions used for Angular files make it easy to move and delete multiple files. Run the following command in the example folder to move the files (they will work in Windows PowerShell, Linux, and macOS):

```
mv src/app/*.model.ts src/app/model/
```

The result is that the files listed in Table 19-5 are moved to the `model` folder.

Table 19-5. The File Moves Required for the Module

File	New Location
<code>src/app/datasource.model.ts</code>	<code>src/app/model/datasource.model.ts</code>
<code>src/app/product.model.ts</code>	<code>src/app/model/product.model.ts</code>
<code>src/app/repository.model.ts</code>	<code>src/app/model/repository.model.ts</code>

If you try to build the project once you have moved the files, the TypeScript compiler will list a series of compiler errors because some of the key declarable classes are unavailable. I'll deal with these problems shortly.

Creating the Module Definition

The next step is to define a module that brings together the functionality in the files that have been moved to the new folder. I added a file called `model.module.ts` in the `src/app/model` folder and defined the module shown in Listing 19-9.

Listing 19-9. The Contents of the `model.module.ts` File in the `src/app/model` Folder

```
import { NgModule } from "@angular/core";
import { SimpleDataSource } from "./datasource.model";
import { Model } from "./repository.model";

@NgModule({
    providers: [Model, SimpleDataSource]
})
export class ModelModule { }
```

The purpose of a feature module is to selectively expose the contents of the folder to the rest of the application. The `@NgModule` decorator for this module uses only the `providers` property to define class providers for the `Model` and `SimpleDataSource` services. When you use providers in a feature module, they are registered with the root module's injector, which means they are available throughout the application, which is exactly what is required for the data model in the example application.

Tip A common mistake is to assume that services defined in a module are accessible only to the classes within that module. There is no module scope in Angular. Providers defined by a feature module are used as though they were defined by the root module. Local providers defined by directives and components in the feature module are available to their view and content children even if they are defined in other modules.

Updating the Other Classes in the Application

Moving classes into the `model` folder has broken import statements in other parts of the application. The next step is to update those import statements to point to the new module. There are four affected

files: attr.directive.ts, categoryFilter.pipe.ts, productForm.component.ts, and productTable.component.ts. Listing 19-10 shows the changes required to the attr.directive.ts file.

Listing 19-10. Updating the Import Reference in the attr.directive.ts File in the src/app Folder

```
import { Directive, ElementRef, Input, SimpleChanges, Output,
    EventEmitter, HostListener, HostBinding } from "@angular/core";
import { Product } from "./model/product.model";

@Directive({
    selector: "[pa-attr]",
})
export class PaAttrDirective {

    @Input("pa-attr")
    @HostBinding("class")
    bgClass: string | null = "";

    @Input("pa-product")
    product: Product = new Product();

    @Output("pa-category")
    click = new EventEmitter<string>();

    @HostListener("click")
    triggerCustomEvent() {
        if (this.product != null) {
            this.click.emit(this.product.category);
        }
    }
}
```

The only change that is required is to update the path used in the `import` statement to reflect the new location of the code file. Listing 19-11 shows the same change applied to the categoryFilter.pipe.ts file.

Listing 19-11. Updating the Import Reference in the categoryFilter.pipe.ts File in the src/app Folder

```
import { Pipe } from "@angular/core";
import { Product } from "./model/product.model";

@Pipe({
    name: "filter",
    pure: false
})
export class PaCategoryFilterPipe {

    transform(products: Product[] | undefined, category: string | undefined): Product[] {
        if (products == undefined) {
            return [];
        }
        return category == undefined ?
            products : products.filter(p => p.category == category);
    }
}
```

```

    }
}
}
```

Listing 19-12 updates the import statements in the `productForm.component.ts` file.

Listing 19-12. Updating Import Paths in the `productForm.component.ts` File in the `src/app` Folder

```

import { Component, Output, EventEmitter, ViewEncapsulation,
    Inject, SkipSelf } from "@angular/core";
import { Product } from "./model/product.model";
import { Model } from "./model/repository.model";
import { VALUE_SERVICE } from "./valueDisplay.directive";

@Component({
    selector: "pa-productform",
    templateUrl: "productForm.component.html",
    viewProviders: [{ provide: VALUE_SERVICE, useValue: "Oranges" }]
})
export class ProductFormComponent {
    newProduct: Product = new Product();

    constructor(private model: Model,
        @Inject(VALUE_SERVICE) @SkipSelf() private serviceValue: string) {
        console.log("Service Value: " + serviceValue);
    }

    submitForm(form: any) {
        this.model.saveProduct(this.newProduct);
        this.newProduct = new Product();
        form.resetForm();
    }
}
```

Listing 19-13 updates the paths in the final file, `productTable.component.ts`.

Listing 19-13. Updating Import Paths in the `productTable.component.ts` File in the `src/app` Folder

```

import { Component, Input, QueryList, ViewChildren } from "@angular/core";
import { Model } from "./model/repository.model";
import { Product } from "./model/product.model";
import { DiscountService } from "./discount.service";
import { LogService } from "./log.service";

@Component({
    selector: "paProductTable",
    templateUrl: "productTable.component.html",
    providers:[LogService]
})
export class ProductTableComponent {

    // ...statements omitted for brevity...
}
```

USING A JAVASCRIPT MODULE WITH AN ANGULAR MODULE

Creating an Angular module allows related application features to be grouped together but still requires that each one is imported from its own file when it is needed elsewhere in the application, as you have seen in the listings in this section.

You can also define a JavaScript module that exports the public-facing features of the Angular module so they can be accessed with the same kind of `import` statement that is used for the `@angular/core` module, for example. To use a JavaScript module, add a file called `index.ts` alongside the TypeScript file that defines the Angular module, which is the `src/app/model` folder for the examples in this section. For each of the application features that you want to use outside of the application, add an `export...from` statement, like this:

```
...
export { ModelModule } from "./model.module";
export { Product } from "./product.model";
export { SimpleDataSource } from "./datasource.model";
export { Model } from "./repository.model";
...
```

These statements export the contents of the individual TypeScript files. You can then import the features you require without having to specify individual files, like this:

```
...
import { Component, Output, EventEmitter, ViewEncapsulation,
  Inject, SkipSelf } from "@angular/core";
import { Product, Model } from "./model";
import { VALUE_SERVICE } from "./valueDisplay.directive";
...
```

Using the filename `index.ts` means that you only have to specify the name of the folder in the `import` statement, producing a result that is neater and more consistent with the Angular core packages.

That said, I don't use this technique in my own projects. Using an `index.ts` file means you have to remember to add every feature to both the Angular and JavaScript modules, which is an extra step that I often forget to do. Instead, I use the approach shown in this chapter and import directly from the files that contain the application's features.

Updating the Root Module

The final step is to update the root module so that the services defined in the feature module are made available throughout the application. Listing 19-14 shows the required changes.

Listing 19-14. Updating the Root Module in the app.module.ts File in the src/app Folder

```

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { DiscountService } from "./discount.service";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
// import { SimpleDataSource } from "./datasource.model";
// import { Model } from "./repository.model";
import { LogService, LOG_SERVICE, SpecialLogService,
    LogLevel, LOG_LEVEL} from "./log.service";
import { VALUE_SERVICE, PaDisplayValueDirective} from "./valueDisplay.directive";
import { ModelModule } from "./model/model.module";

let logger = new LogService();
logger.minimumLevel = LogLevel.DEBUG;

registerLocaleData(localeFr);

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe, PaDiscountDisplayComponent, PaDiscountEditorComponent,
    PaDiscountPipe, PaDiscountAmountDirective, PaDisplayValueDirective],

```

```

imports: [
  BrowserModule,
  BrowserAnimationsModule,
  FormsModule, ReactiveFormsModule, ModelModule
],
providers: [DiscountService, LogService,
  { provide: VALUE_SERVICE, useValue: "Apples"  }],
bootstrap: [ProductFormComponent, ProductTableComponent]
})
export class AppModule { }

```

I imported the feature module and added it to the root module's imports list. Since the feature module defines providers for Model and SimpleDataSource, I removed the entries from the root module's providers list and removed the associated import statements.

Once you have saved the changes, you can run `ng serve` to start the Angular development tools. The application will compile, and the revised root module will provide access to the model service. There are no visible changes to the content displayed in the browser, and the changes are limited to the structure of the project. (You may need to restart the Angular development tools and reload the browser to see the changes.)

Creating a Utility Feature Module

A model module is a good place to start because it demonstrates the basic structure of a feature module and how it relates to the root module. The impact on the application was slight, however, and not a great deal of simplification was achieved.

The next step up in complexity is a utility feature module, which groups together all of the common functionality in the application, such as pipes and directives. In a real project, you might be more selective about how you group these types of building blocks together so that there are several modules, each containing similar functionality. For the example application, I am going to move all of the pipes, directives, and services into a single module.

Creating the Module Folder and Moving the Files

As with the previous module, the first step is to create the folder. For this module, I created a folder called `src/app/common` and moved code files for the pipes and directives by running the following commands in the example folder:

```

mkdir src/app/common
mv src/app/*.pipe.ts src/app/common/
mv src/app/*.directive.ts src/app/common/

```

These commands should work in Windows PowerShell, Linux, and macOS. Some of the directives and pipes in the application rely on the `DiscountService` and `LogServices` classes, which are provided to them through dependency injection. Run the following command in the example folder to move the TypeScript file for the service into the module folder:

```
mv src/app/*.service.ts src/app/common/
```

The result is that the files listed in Table 19-6 are moved to the `common` module folder.

Table 19-6. The File Moves Required for the Module

File	New Location
app/addTax.pipe.ts	app/common/addTax.pipe.ts
app/attr.directive.ts	app/common/attr.directive.ts
app/categoryFilter.pipe.ts	app/common/categoryFilter.pipe.ts
app/cellColor.directive.ts	app/common/cellColor.directive.ts
app/cellColorSwitcher.directive.ts	app/common/cellColorSwitcher.directive.ts
app/discount.pipe.ts	app/common/discount.pipe.ts
app/discountAmount.directive.ts	app/common/discountAmount.directive.ts
app/iterator.directive.ts	app/common/iterator.directive.ts
app/structure.directive.ts	app/common/structure.directive.ts
app/twoWay.directive.ts	app/common/twoWay.directive.ts
app/valueDisplay.directive.ts	app/common/valueDisplay.directive.ts
app/discount.service.ts	app/common/discount.service.ts
app/log.service.ts	app/common/log.service.ts

Updating the Classes in the New Module

Some of the classes that have been moved into the new folder have `import` statements that have to be updated to reflect the new path to the model module. Listing 19-15 shows the change required to the `attr.directive.ts` file.

Listing 19-15. Updating the Imports in the attr.directive.ts File in the src/app/common Folder

```
import { Directive, ElementRef, Input, SimpleChanges, Output,
    EventEmitter, HostListener, HostBinding } from "@angular/core";
import { Product } from "../model/product.model";

@Directive({
    selector: "[pa-attr]",
})
export class PaAttrDirective {

    // ...statements omitted for brevity...
}
```

Listing 19-16 shows the corresponding change to the `categoryFilter.pipe.ts` file.

Listing 19-16. Updating the Imports in the categoryFilter.pipe.ts File in the src/app/common Folder

```
import { Pipe } from "@angular/core";
import { Product } from "../model/product.model";

@Pipe({
  name: "filter",
  pure: false
})
export class PaCategoryFilterPipe {

  transform(products: Product[] | undefined, category: string | undefined):
    Product[] {
    if (products == undefined) {
      return [];
    }
    return category == undefined ?
      products : products.filter(p => p.category == category);
  }
}
```

Creating the Module Definition

The next step is to define a module that brings together the functionality in the files that have been moved to the new folder. I added a file called common.module.ts in the src/app/common folder and defined the module shown in Listing 19-17.

Listing 19-17. The Contents of the common.module.ts File in the src/app/common Folder

```
import { NgModule } from "@angular/core";
import { PaAddTaxPipe } from "./addTax.pipe";
import { PaAttrDirective } from "./attr.directive";
import { PaCategoryFilterPipe } from "./categoryFilter.pipe";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { PaDiscountPipe } from "./discount.pipe";
import { PaDiscountAmountDirective } from "./discountAmount.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaModel } from "./twoWay.directive";
import { VALUE_SERVICE, PaDisplayValueDirective } from "./valueDisplay.directive";
import { DiscountService } from "./discount.service";
import { LogService } from "./log.service";
import { ModelModule } from "../model/model.module";

@NgModule({
  imports: [ModelModule],
  providers: [LogService, DiscountService,
    { provide: VALUE_SERVICE, useValue: "Apples" }],
  declarations: [PaAddTaxPipe, PaAttrDirective, PaCategoryFilterPipe,
```

```

    PaCellColor, PaCellColorSwitcher, PaDiscountPipe,
    PaDiscountAmountDirective, PaIteratorDirective, PaStructureDirective,
    PaModel, PaDisplayValueDirective],
exports: [PaAddTaxPipe, PaAttrDirective, PaCategoryFilterPipe,
    PaCellColor, PaCellColorSwitcher, PaDiscountPipe,
    PaDiscountAmountDirective, PaIteratorDirective, PaStructureDirective,
    PaModel, PaDisplayValueDirective]
})
export class CommonModule { }

```

This is a more complex module than the one required for the data model. In the sections that follow, I describe the values that are used for each of the decorator's properties.

Understanding the Imports

Some of the directives and pipes in the module depend on the services defined in the `model` module, created earlier in this chapter. To ensure that the features in that module are available, I have added to the common module's `imports` property.

Understanding the Providers

The `providers` property ensures that the services, directives, and pipes in the feature module have access to the services they require. This means adding class providers to create `LogService` and `DiscountService` services, which will be added to the root module's providers when the module is loaded. Not only will the services be available to the directives and pipes in the common module; they will also be available throughout the application.

Understanding the Declarations

The `declarations` property is used to provide Angular with a list of the directives and pipes (and components, if there are any) in the module. In a feature module, this property has two purposes: it enables the declarable classes for use in any templates contained within the module, and it allows a module to make those declarable classes available outside of the module. I create a module that contains template content later in this chapter, but for this module, the value of the `declarations` property is that it must be used to prepare for the `exports` property, described in the next section.

Understanding the Exports

For a module that contains directives and pipes intended for use elsewhere in the application, the `exports` property is the most important in the `@NgModule` decorator because it defines the set of directives, components, and pipes that the module provides for use when it is imported elsewhere in the application. The `exports` property can contain individual classes and module types, although both must already be listed in the `declarations` or `imports` property. When the module is imported, the types listed behave as though they had been added to the importing module's `declarations` property.

Updating the Other Classes in the Application

Now that the module has been defined, I can update the other files in the application that contain `import` statements for the types that are now part of the common module. Listing 19-18 shows the changes required to the `discountDisplay.component.ts` file.

Listing 19-18. Updating the Import in the `discountDisplay.component.ts` File in the `src/app` Folder

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "./common/discount.service";

@Component({
  selector: "paDiscountDisplay",
  template: `<div class="bg-info text-white p-2 my-2">
    The discount is {{discouter?.discount }}
  </div>`
})
export class PaDiscountDisplayComponent {

  constructor(public discouter: DiscountService) { }
}
```

Listing 19-19 shows the changes to the `discountEditor.component.ts` file.

Listing 19-19. Updating the Import Reference in the `discountEditor.component.ts` File in the `src/app` Folder

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "./common/discount.service";

@Component({
  selector: "paDiscountEditor",
  template: `<div class="form-group">
    <label>Discount</label>
    <input [(ngModel)]="discouter.discount"
      class="form-control" type="number" />
  </div>`
})
export class PaDiscountEditorComponent {

  constructor(public discouter: DiscountService) { }
}
```

Listing 19-20 shows the changes to the `productForm.component.ts` file.

Listing 19-20. Updating the Import Reference in the `productForm.component.ts` File in the `src/app` Folder

```
import { Component, Output, EventEmitter, ViewEncapsulation,
  Inject, SkipSelf } from "@angular/core";
import { Product } from "./model/product.model";
import { Model } from "./model/repository.model";
import { VALUE_SERVICE } from "./common/valueDisplay.directive";
```

```

@Component({
  selector: "pa-productform",
  templateUrl: "productForm.component.html",
  viewProviders: [{ provide: VALUE_SERVICE, useValue: "Oranges" }]
})
export class ProductFormComponent {
  newProduct: Product = new Product();

  constructor(private model: Model,
    @Inject(VALUE_SERVICE) @SkipSelf() private serviceValue: string) {
    console.log("Service Value: " + serviceValue);
  }

  submitForm(form: any) {
    this.model.saveProduct(this.newProduct);
    this.newProduct = new Product();
    form.resetForm();
  }
}

```

The final change is to the `productTable.component.ts` file, as shown in Listing 19-21.

Listing 19-21. Updating the Import Reference in the `productTable.component.ts` File in the `src/app` Folder

```

import { Component, Input, QueryList, ViewChildren } from "@angular/core";
import { Model } from "./model/repository.model";
import { Product } from "./model/product.model";
import { DiscountService } from "./common/discount.service";
import { LogService } from "./common/log.service";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html",
  providers:[LogService]
})
export class ProductTableComponent {

  constructor(private dataModel: Model) { }

  getProduct(key: number): Product | undefined {
    return this.dataModel?.getProduct(key);
  }

  getProducts(): Product[] | undefined {
    return this.dataModel?.getProducts();
  }

  deleteProduct(key: number) {
    this.dataModel?.deleteProduct(key);
  }
}

```

```

    taxRate: number = 0;
    categoryFilter: string | undefined;
    itemCount: number = 3;
}

```

Updating the Root Module

The final step is to update the root module so that it loads the common module to provide access to the directives and pipes it contains, as shown in Listing 19-22.

Listing 19-22. Importing a Feature Module in the app.module.ts File in the src/app Folder

```

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
// import { PaAttrDirective } from "./attr.directive";
// import { PaModel } from "./twoWay.directive";
// import { PaStructureDirective } from "./structure.directive";
// import { PaIteratorDirective } from "./iterator.directive";
// import { PaCellColor } from "./cellColor.directive";
// import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";
import { ProductTableComponent } from "./productTable.component";
import { ProductFormComponent } from "./productForm.component";
import { PaToggleView } from "./toggleView.component";
// import { PaAddTaxPipe } from "./addTax.pipe";
// import { PaCategoryFilterPipe } from "./categoryFilter.pipe";

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
// import { DiscountService } from "./discount.service";
// import { PaDiscountPipe } from "./discount.pipe";
// import { PaDiscountAmountDirective } from "./discountAmount.directive";

// import { LogService, LOG_SERVICE, SpecialLogService,
//         LogLevel, LOG_LEVEL} from "./log.service";
// import { VALUE_SERVICE, PaDisplayValueDirective} from "./valueDisplay.directive";
import { ModelModule } from "./model/model.module";
import { CommonModule } from "./common/common.module";

// let logger = new LogService();
// logger.minimumLevel = LogLevel.DEBUG;

```

```

registerLocaleData(localeFr);

@NgModule({
  declarations: [ProductComponent, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaDiscountDisplayComponent,
    PaDiscountEditorComponent],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule, ModelModule, CommonModule
  ],
  // providers: [DiscountService, LogService,
  //   { provide: VALUE_SERVICE, useValue: "Apples" }],
  bootstrap: [ProductFormComponent, ProductTableComponent]
})
export class AppModule { }

```

The root module has been substantially simplified with the creation of the `common` module, which has been added to the `imports` list. All of the individual classes for directives and pipes have been removed from the `declarations` list, and their associated `import` statements have been removed from the file. When the `common` module is imported, all of the types listed in its `exports` property will be added to the root module's `declarations` property.

Once you have saved the changes in this section, you can run the `ng serve` command to start the Angular development tools. Once again, there is no visible change in the content presented to the user, and the differences are all in the structure of the application.

Creating a Feature Module with Components

The final module that I am going to create will contain the application's components. The process for creating the module is the same as in the previous examples, as described in the sections that follow.

Creating the Module Folder and Moving the Files

The module will be called `components`, and I created the folder `src/app/components` to contain the files. Run the following commands in the example folder to create the folder; move the directive TypeScript, HTML, and CSS files into the new folder; and delete the corresponding JavaScript files:

```

mkdir src/app/components
mv src/app/*.component.ts src/app/components/
mv src/app/*.component.html src/app/components/
mv src/app/*.component.css src/app/components/

```

The result of these commands is that the component code files, templates, and style sheets are moved into the new folder, as listed in Table 19-7.

Table 19-7. The File Moves Required for the Component Module

File	New Location
src/app/app.component.ts	src/app/components/app.component.ts
src/app/app.component.html	src/app/components/app.component.html
src/app/app.component.css	src/app/components/app.component.css
src/app /discountDisplay.component.ts	src/app/components /discountDisplay.component.ts
src/app/discountEditor.component.ts	src/app/components/discountEditor.component.ts
src/app/productForm.component.ts	src/app/components/productForm.component.ts
src/app/productForm.component.html	src/app/components/productForm.component.html
src/app/productForm.component.css	src/app/components/productForm.component.css
src/app/productTable.component.ts	src/app/components/productTable.component.ts
src/app/productTable.component.html	src/app/components/productTable.component.html
src/app/productTable.component.css	src/app/components/productTable.component.css
src/app/toggleView.component.ts	src/app/components/toggleView.component.ts
src/app/toggleView.component.html	src/app/components/toggleView.component.ts

Creating the Module Definition

To create the module, I added a file called `components.module.ts` to the `src/app/components` folder and added the statements shown in Listing 19-23.

Listing 19-23. The Contents of the `components.module.ts` File in the `src/app/components` Folder

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { CommonModule } from "../common/common.module";
import { FormsModule, ReactiveFormsModule } from "@angular/forms"
import { PaDiscountDisplayComponent } from "./discountDisplay.component";
import { PaDiscountEditorComponent } from "./discountEditor.component";
import { ProductFormComponent } from "./productForm.component";
import { ProductTableComponent } from "./productTable.component";

@NgModule({
  imports: [BrowserModule, FormsModule, ReactiveFormsModule, CommonModule],
  declarations: [PaDiscountDisplayComponent, PaDiscountEditorComponent,
    ProductFormComponent, ProductTableComponent],
  exports: [ProductFormComponent, ProductTableComponent]
})
export class ComponentsModule { }
```

This module imports `BrowserModule` and `CommonModule` to ensure that the directives have access to the services and the declarable classes they require. It exports the `ProductFormComponent` and `ProductTableComponent` components, which are the two components used in the root component's `bootstrap` property. The other components are private to the module.

Updating the Other Classes

Moving the TypeScript files into the `components` folder requires some changes to the paths in the `import` statements. Listing 19-24 shows the change required for the `discountDisplay.component.ts` file.

Listing 19-24. Updating a Path in the `discountDisplay.component.ts` File in the `src/app/component` Folder

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "../common/discount.service";

@Component({
  selector: "paDiscountDisplay",
  template: `<div class="bg-info text-white p-2 my-2">
    The discount is {{discountr?.discount }}
  </div>`
})
export class PaDiscountDisplayComponent {

  constructor(public discountr: DiscountService) { }
}
```

Listing 19-25 shows the change required to the `discountEditor.component.ts` file.

Listing 19-25. Updating a Path in the `discountEditor.component.ts` File in the `src/app/component` Folder

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "../common/discount.service";

@Component({
  selector: "paDiscountEditor",
  template: `<div class="form-group">
    <label>Discount</label>
    <input [(ngModel)]="discountr.discount"
      class="form-control" type="number" />
  </div>`
})
export class PaDiscountEditorComponent {

  constructor(public discountr: DiscountService) { }
}
```

Listing 19-26 shows the changes required for the `productForm.component.ts` file.

Listing 19-26. Updating a Path in the productForm.component.ts File in the src/app/component Folder

```

import { Component, Output, EventEmitter, ViewEncapsulation,
    Inject, SkipSelf } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { VALUE_SERVICE } from "../common/valueDisplay.directive";

@Component({
    selector: "pa-productform",
    templateUrl: "productForm.component.html",
    viewProviders: [{ provide: VALUE_SERVICE, useValue: "Oranges" }]
})
export class ProductFormComponent {
    newProduct: Product = new Product();

    constructor(private model: Model,
        @Inject(VALUE_SERVICE) @SkipSelf() private serviceValue: string) {
        console.log("Service Value: " + serviceValue);
    }

    submitForm(form: any) {
        this.model.saveProduct(this.newProduct);
        this.newProduct = new Product();
        form.resetForm();
    }
}

```

Listing 19-27 shows the changes required to the productTable.component.ts file.

Listing 19-27. Updating a Path in the productTable.component.ts File in the src/app/component Folder

```

import { Component, Input, QueryList, ViewChildren } from "@angular/core";
import { Model } from "../model/repository.model";
import { Product } from "../model/product.model";
import { DiscountService } from "../common/discount.service";
import { LogService } from "../common/log.service";

@Component({
    selector: "paProductTable",
    templateUrl: "productTable.component.html",
    providers:[LogService]
})
export class ProductTableComponent {

    constructor(private dataModel: Model) { }

    getProduct(key: number): Product | undefined {
        return this.dataModel?.getProduct(key);
    }
}

```

```

getProducts(): Product[] | undefined {
    return this.dataModel?.getProducts();
}

deleteProduct(key: number) {
    this.dataModel?.deleteProduct(key);
}

taxRate: number = 0;
categoryFilter: string | undefined;
itemCount: number = 3;
}

```

Updating the Root Module

The final step is to update the root module to remove the outdated references to the individual files and to import the new module, as shown in Listing 19-28.

Listing 19-28. Importing a Feature Module in the app.module.ts File in the src/app Folder

```

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

//import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";

import { ProductTableComponent } from "./components/productTable.component";
import { ProductFormComponent } from "./components/productForm.component";
// import { PaToggleView } from "./toggleView.component";

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

// import { PaDiscountDisplayComponent } from "./discountDisplay.component";
// import { PaDiscountEditorComponent } from "./discountEditor.component";

import { ModelModule } from "./model/model.module";
import { CommonModule } from "./common/common.module";
import { ComponentsModule } from "./components/components.module";

registerLocaleData(localeFr);

@NgModule({
    declarations: [ProductComponent],
    imports: [
        BrowserModule,
        BrowserAnimationsModule,
        FormsModule,
        ReactiveFormsModule,
        ComponentsModule
    ]
})

```

```
FormsModule, ReactiveFormsModule, ModelModule, CommonModule,  
ComponentsModule  
],  
bootstrap: [ProductFormComponent, ProductTableComponent]  
}  
export class AppModule { }
```

Restart the Angular development tools to build and display the application. Adding modules to the application has radically simplified the root module and allows related features to be defined in self-contained blocks, which can be extended or modified in relative isolation from the rest of the application.

Summary

In this chapter, I described the last of the Angular building blocks: modules. I explained the role of the root module and demonstrated how to create feature modules to add structure to an application. In the next part of the book, I describe the features that Angular provides to shape the building blocks into complex and responsive applications.

CHAPTER 20



Creating the Example Project

Throughout the chapters in the previous part of the book, I added classes and content to the example project to demonstrate different Angular features and then, in Chapter 19, introduced feature modules to add some structure to the project. The result is a project with a lot of redundant and unused functionality, and for this part of the book, I am going to start a new project that takes some of the core features from earlier chapters and provides a clean foundation on which to build in the chapters that follow.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Starting the Example Project

To create the project and populate it with tools and placeholder content, open a new command prompt, navigate to a convenient location, and run the command shown in Listing 20-1.

Listing 20-1. Creating the Example Project

```
ng new exampleApp --routing false --style css --skip-git --skip-tests
```

To distinguish the project used in this part of the book from earlier examples, I created a project called exampleApp. The project initialization process will take a while to complete as all the required packages are downloaded.

Adding and Configuring the Bootstrap CSS Package

I continue to use the Bootstrap CSS framework to style the HTML elements in this chapter and the rest of the book. Run the command shown in Listing 20-2 in the exampleApp folder to add the Bootstrap package to the project.

Listing 20-2. Adding a Package to the Project

```
npm install bootstrap@5.1.3
```

The Bootstrap package isn't specific to Angular development and doesn't use the schematics API, which means that a manual change must be made to the `angular.config` file to include the Bootstrap CSS stylesheet in the styles bundle. Run the command shown in Listing 20-3 in the `exampleApp` folder. Take care to enter the command exactly as shown and do not introduce additional spaces or quotes.

Listing 20-3. Changing the Application Configuration

```
ng config projects.exampleApp.architect.build.options.styles \
'["src/styles.css", \
'"node_modules/bootstrap/dist/css/bootstrap.min.css"]'
```

If you are using Windows, then use a PowerShell prompt to run the command shown in Listing 20-4 in the `exampleApp` folder.

Listing 20-4. Changing the Application Configuration Using PowerShell

```
ng config projects.exampleApp.architect.build.options.styles ` 
'[""src/styles.css"", ""node_modules/bootstrap/dist/css/bootstrap.min.css"""]'
```

Creating the Project Structure

Create the folders shown in Table 20-1 in preparation for the feature modules that the example project will contain.

Table 20-1. The Folders Created for the Example Application

Name	Description
<code>src/app/model</code>	This folder will contain a feature module containing the data model.
<code>src/app/core</code>	This folder will contain a feature module containing components that provide the core features of the application.
<code>src/app/messages</code>	This folder will contain a feature module that is used to display messages and errors to the user.

Creating the Model Module

The first feature module will contain the project's data model, which is similar to the one used in Part 2.

Creating the Product Data Type

To define the basic data type around which the application is based, I added a file called `product.model.ts` to the `src/app/model` folder and defined the class shown in Listing 20-5.

Listing 20-5. The Contents of the product.model.ts File in the src/app/model Folder

```
export class Product {

    constructor(public id?: number,
        public name?: string,
        public category?: string,
        public price?: number) { }

}
```

Creating the Data Source and Repository

To provide the application with some initial data, I created a file called static.datasource.ts in the src/app/model folder and defined the service shown in Listing 20-6. This class will be used as the data source until Chapter 23, where I explain how to use asynchronous HTTP requests to request data from web services.

Tip I am more relaxed about following the name conventions for Angular files when creating files within a feature module, especially if the purpose of the module is obvious from its name.

Listing 20-6. The Contents of the static.datasource.ts File in the src/app/model Folder

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";

@Injectable()
export class StaticDataSource {
    private data: Product[];

    constructor() {
        this.data = new Array<Product>(
            new Product(1, "Kayak", "Watersports", 275),
            new Product(2, "Lifejacket", "Watersports", 48.95),
            new Product(3, "Soccer Ball", "Soccer", 19.50),
            new Product(4, "Corner Flags", "Soccer", 34.95),
            new Product(5, "Thinking Cap", "Chess", 16));
    }

    getData(): Product[] {
        return this.data;
    }
}
```

The next step is to define the repository, through which the rest of the application will access the model data. I created a file called repository.model.ts in the src/app/model folder and used it to define the class shown in Listing 20-7.

Listing 20-7. The Contents of the repository.model.ts File in the src/app/model Folder

```

import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { StaticDataSource } from "./static.datasource";

@Injectable()
export class Model {
    private products: Product[];
    private locator = (p: Product, id?: number) => p.id == id;

    constructor(private dataSource: StaticDataSource) {
        this.products = new Array<Product>();
        this.dataSource.getData().forEach(p => this.products.push(p));
    }

    getProducts(): Product[] {
        return this.products;
    }

    getProduct(id: number): Product | undefined {
        return this.products.find(p => this.locator(p, id));
    }

    saveProduct(product: Product) {
        if (product.id == 0 || product.id == null) {
            product.id = this.generateID();
            this.products.push(product);
        } else {
            let index = this.products
                .findIndex(p => this.locator(p, product.id));
            this.products.splice(index, 1, product);
        }
    }

    deleteProduct(id: number) {
        let index = this.products.findIndex(p => this.locator(p, id));
        if (index > -1) {
            this.products.splice(index, 1);
        }
    }

    private generateID(): number {
        let candidate = 100;
        while (this.getProduct(candidate) != null) {
            candidate++;
        }
        return candidate;
    }
}

```

Completing the Model Module

To complete the data model, I need to define the module. I created a file called `model.module.ts` in the `src/app/model` folder and used it to define the Angular module shown in Listing 20-8.

Listing 20-8. The Contents of the `model.module.ts` File in the `src/app/model` Folder

```
import { NgModule } from "@angular/core";
import { StaticDataSource } from "./static.datasource";
import { Model } from "./repository.model";

@NgModule({
    providers: [Model, StaticDataSource]
})
export class ModelModule { }
```

Creating the Messages Module

The messages module will contain a service that is used to report messages or errors that should be displayed to the user and a component that presents them.

Creating the Message Model and Service

To represent messages that should be displayed to the user, I added a file called `message.model.ts` to the `src/app/messages` folder and added the code shown in Listing 20-9.

Listing 20-9. The Contents of the `message.model.ts` File in the `src/app/messages` Folder

```
export class Message {

    constructor(public text: string,
               public error: boolean = false) { }

}
```

The `Message` class defines properties that present the text that will be displayed to the user and whether the message represents an error. Next, I created a file called `message.service.ts` in the `src/app/messages` folder and used it to define the service shown in Listing 20-10, which will be used to register messages that should be displayed to the user.

Listing 20-10. The Contents of the `message.service.ts` File in the `src/app/messages` Folder

```
import { Injectable } from "@angular/core";
import { Message } from "./message.model";
import { Observable, ReplaySubject, Subject } from "rxjs";

@Injectable()
export class MessageService {

    messages: Observable<Message> = new ReplaySubject<Message>(1);
```

```

    reportMessage(msg: Message) {
        (this.messages as Subject<Message>).next(msg);
    }
}

```

Angular services don't support output properties but can still use the features provided by the RxJS package to send events. This service defines a `reportMessage` method that sends a new event through a `ReplaySubject<Message>` that is presented to the rest of the application as an `Observable<Message>`. I used a `ReplaySubject` so that new subscribers will immediately receive the most recent message.

This service is essentially used to provide access to the observable/subject, and I could have made this object available directly as a service using the service provider features described in Chapter 18. However, I prefer to introduce a lightweight service as a wrapper around the observable so that I can easily alter the way that events are created by modifying the `reportMessage` method.

Creating the Component and Template

Now that I have a source of messages, I can create a component that will display them to the user. I added a file called `message.component.ts` to the `src/app/messages` folder and defined the component shown in Listing 20-11.

Listing 20-11. The Contents of the `message.component.ts` File in the `src/app/messages` Folder

```

import { Component } from "@angular/core";
import { MessageService } from "./message.service";
import { Message } from "./message.model";

@Component({
    selector: "paMessages",
    templateUrl: "message.component.html",
})
export class MessageComponent {
    lastMessage?: Message;

    constructor(messageService: MessageService) {
        messageService.messages.subscribe(msg => this.lastMessage = msg);
    }
}

```

The component receives a `MessageService` object as its constructor argument and subscribes to the events it emits in order to receive messages, the most recent of which is assigned to a property called `lastMessage`. To provide a template for the component, I created a file called `message.component.html` in the `src/app/messages` folder and added the markup shown in Listing 20-12, which displays the message to the user.

Listing 20-12. The Contents of the `message.component.html` File in the `src/app/messages` Folder

```

<div *ngIf="lastMessage"
    class="bg-primary text-white p-2 text-center"
    [class.bg-danger]="lastMessage.error">
    <h4>{{lastMessage.text}}</h4>
</div>

```

Completing the Message Module

I added a file called `message.module.ts` in the `src/app/messages` folder and defined the module shown in Listing 20-13.

Listing 20-13. The Contents of the `message.module.ts` File in the `src/app/messages` Folder

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { MessageComponent } from "./message.component";
import { MessageService } from "./message.service";

@NgModule({
  imports: [BrowserModule],
  declarations: [MessageComponent],
  exports: [MessageComponent],
  providers: [MessageService]
})
export class MessageModule { }
```

Creating the Core Module

The core module will contain the central functionality of the application, built on features that were described in Part 2, that presents the user with a list of the products in the model and the ability to create and edit them.

Creating the Shared State Service

To help the components in this module to collaborate, I am going to add a service that records the current mode, noting whether the user is editing or creating a product. I added a file called `sharedState.service.ts` to the `src/app/core` folder and defined the enum and class shown in Listing 20-14.

Listing 20-14. The Contents of the `sharedState.service.ts` File in the `src/app/core` Folder

```
import { Injectable } from "@angular/core";
import { Observable, Subject } from "rxjs"

export enum MODES {
  CREATE, EDIT
}

export interface StateUpdate {
  mode: MODES
  id?: number
}

@Injectable()
export class SharedState {
  private modeValue: MODES = MODES.EDIT;
  private idValue?: number;
```

```

constructor() {
    this.changes = new Subject<StateUpdate>();
}
get id(): number | undefined { return this.idValue; }
get mode(): MODES { return this.modeValue; }
changes: Observable<StateUpdate>
update(mode: MODES, id?: number) {
    this.modeValue = mode;
    this.idValue = id;
    (this.changes as Subject<StateUpdate>).next({
        mode: this.modeValue, id: this.idValue
    });
}
}

```

The SharedState class contains two get-only properties that reflect the current mode and the ID of the data model object that is being operated on. These properties return the values of private fields. The state is changed using the update method, which sends out events through an RXJS Subject, which is publicly presented as an Observable to present other components from creating their own events.

Creating the Table Component

This component will present the user with the table that lists all the products in the application and that will be the main focal point in the application, providing access to other areas of functionality through buttons that allow objects to be created, edited, or deleted. Listing 20-15 shows the contents of the table.component.ts file, which I created in the src/app/core folder.

Listing 20-15. The Contents of the table.component.ts File in the src/app/core Folder

```

import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { MODES, SharedState } from "./sharedState.service";

@Component({
    selector: "paTable",
    templateUrl: "table.component.html"
})
export class TableComponent {

    constructor(private model: Model, private state: SharedState) { }

    getProduct(key: number): Product | undefined {
        return this.model.getProduct(key);
    }
}

```

```

getProducts(): Product[] {
    return this.model.getProducts();
}

deleteProduct(key?: number) {
    if (key != undefined) {
        this.model.deleteProduct(key);
    }
}

editProduct(key?: number) {
    this.state.update(MODES.EDIT, key)
}

createProduct() {
    this.state.update(MODES.CREATE);
}
}

```

This component provides the same basic functionality used in Part 2, with the addition of the `editProduct` and `createProduct` methods. These methods update the shared state service when the user wants to edit or create a product.

Creating the Table Component Template

To provide the table component with a template, I added an HTML file called `table.component.html` to the `src/app/core` folder and added the markup shown in Listing 20-16.

Listing 20-16. The Contents of the `table.component.html` File in the `src/app/core` Folder

```

<table class="table table-sm table-bordered table-striped">
    <thead>
        <tr>
            <th>ID</th><th>Name</th><th>Category</th><th>Price</th><th></th>
        </tr>
    </thead>
    <tbody>
        <tr *ngFor="let item of getProducts()">
            <td>{{item.id}}</td>
            <td>{{item.name}}</td>
            <td>{{item.category}}</td>
            <td>{{item.price | currency:"USD" }}</td>
            <td class="text-center">
                <button class="btn btn-danger btn-sm m-1"
                    (click)="deleteProduct(item.id)">
                    Delete
                </button>
                <button class="btn btn-warning btn-sm" (click)="editProduct(item.id)">
                    Edit
                </button>
            </td>
        </tr>
    </tbody>
</table>

```

```

</tr>
</tbody>
</table>
<button class="btn btn-primary mt-1" (click)="createProduct()">
  Create New Product
</button>

```

This template uses the `ngFor` directive to create rows in a table for each product in the data model, including buttons that call the `deleteProduct` and `editProduct` methods. There is also a button element outside of the table that calls the component's `createProduct` method when it is clicked.

Creating the Form Component

For this project, I am going to create a form component that will manage an HTML form that will allow new products to be created and allow existing products to be modified. To define the component, I added a file called `form.component.ts` to the `src/app/core` folder and added the code shown in Listing 20-17.

Listing 20-17. The Contents of the `form.component.ts` File in the `src/app/core` Folder

```

import { Component } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;

  constructor(private model: Model, private state: SharedState,
    private messageService: MessageService) {
    this.state.changes.subscribe((upd) => this.handleStateChange(upd))
    this.messageService.reportMessage(new Message("Creating New Product"));
  }

  handleStateChange(newState: StateUpdate) {
    this.editing = newState.mode == MODES.EDIT;
    if (this.editing && newState.id) {
      Object.assign(this.product, this.model.getProduct(newState.id)
        ?? new Product());
      this.messageService.reportMessage(
        new Message(`Editing ${this.product.name}`));
    } else {
  }
}

```

```

        this.product = new Product();
        this.messageService.reportMessage(new Message("Creating New Product"));
    }
}

submitForm(form: NgForm) {
    if (form.valid) {
        this.model.saveProduct(this.product);
        this.product = new Product();
        form.resetForm();
    }
}
}

```

A single component will present a form used to both create new products and edit existing ones. The component depends on two services. The SharedState service provides events that are used to change the details of the form shown to the user, based on whether a product is being created or edited. The MessageService service is used to send messages that indicate the create/edit mode so they can be displayed to the user.

Notice that an initial message is sent via the `MessageService` object using the JavaScript `setTimeout` method. This prevents the message from being lost.

Creating the Form Component Template

To provide the component with a template, I added an HTML file called `form.component.html` to the `src/app/core` folder and added the markup shown in Listing 20-18.

Listing 20-18. The Contents of the `form.component.html` File in the `src/app/core` Folder

```

<form #form="ngForm" (ngSubmit)="submitForm(form)" (reset)="form.resetForm()" >
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" name="name"
               [(ngModel)]="product.name" required />
    </div>

    <div class="form-group">
        <label>Category</label>
        <input class="form-control" name="category"
               [(ngModel)]="product.category" required />
    </div>

    <div class="form-group">
        <label>Price</label>
        <input class="form-control" name="price"
               [(ngModel)]="product.price"
               required pattern="^[\d\.\d]+\$" />
    </div>

```

```

<div class="mt-2">
  <button type="submit" class="btn btn-primary"
    [class.btn-warning]="editing" [disabled]="form.invalid">
    {{editing ? "Save" : "Create"}}
  </button>
  <button type="reset" class="btn btn-secondary m-1">Cancel</button>
</div>
</form>

```

The most important part of this template is the `form` element, which contains input elements for the name, category, and price properties required to create or edit a product. The header at the top of the template and the submit button for the form change their content and appearance based on the editing mode to distinguish between different operations.

Creating the Form Component Styles

To keep the example simple, I have used the basic form validation without any error messages. Instead, I rely on CSS styles that are applied using Angular validation classes. I added a file called `form.component.css` to the `src/app/core` folder and defined the styles shown in Listing 20-19.

Listing 20-19. The Contents of the `form.component.css` File in the `src/app/core` Folder

```

input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
input.ng-dirty.ng-valid { border: 2px solid #6bc502 }

```

Completing the Core Module

To define the module that contains the components, I added a file called `core.module.ts` to the `src/app/core` folder and created the Angular module shown in Listing 20-20.

Listing 20-20. The Contents of the `core.module.ts` File in the `src/app/core` Folder

```

import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "./table.component";
import { FormComponent } from "./form.component";
import { SharedState } from "./sharedState.service";

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule],
  declarations: [TableComponent, FormComponent],
  exports: [ModelModule, TableComponent, FormComponent],
  providers: [SharedState]
})
export class CoreModule { }

```

This module imports the core Angular functionality, the Angular form features, and the application's data model, created earlier in the chapter. It also sets up a provider for the `SharedState` service.

Completing the Project

To bring all the different modules together, I made the changes shown in Listing 20-21 to the root module.

Listing 20-21. Configuring the Application in the app.module.ts File in the src/app Folder

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

//import { AppComponent } from './app.component';

import { ModelModule } from './model/model.module';
import { CoreModule } from './core/core.module';
import { TableComponent } from './core/table.component';
import { FormComponent } from './core/form.component';
import { MessageModule } from './messages/message.module';
import { MessageComponent } from './messages/message.component';

@NgModule({
  declarations: [],
  imports: [BrowserModule, ModelModule, CoreModule, MessageModule],
  providers: [],
  bootstrap: [TableComponent, FormComponent, MessageComponent]
})
export class AppModule { }
```

The module imports the feature modules created in this chapter and specifies three bootstrap components, two of which were defined in CoreModule and one from MessageModule. These will display the product table and form and any messages or errors.

The final step is to update the HTML file so that it contains elements that will be matched by the selector properties of the bootstrap components, as shown in Listing 20-22.

Listing 20-22. Adding Custom Elements in the index.html File in the src Folder

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>ExampleApp</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <paMessages></paMessages>
  <div class="row m-2">
    <div class="col-8 p-2">
      <paTable></paTable>
    </div>
    <div class="col-4 p-2">
      <paForm></paForm>
```

```

</div>
</div>
</body>
</html>

```

Run the following command in the exampleApp folder to start the Angular development tools and build the project:

```
ng serve
```

Once the initial build process has completed, open a new browser window and navigate to <http://localhost:4200> to see the content shown in Figure 20-1.

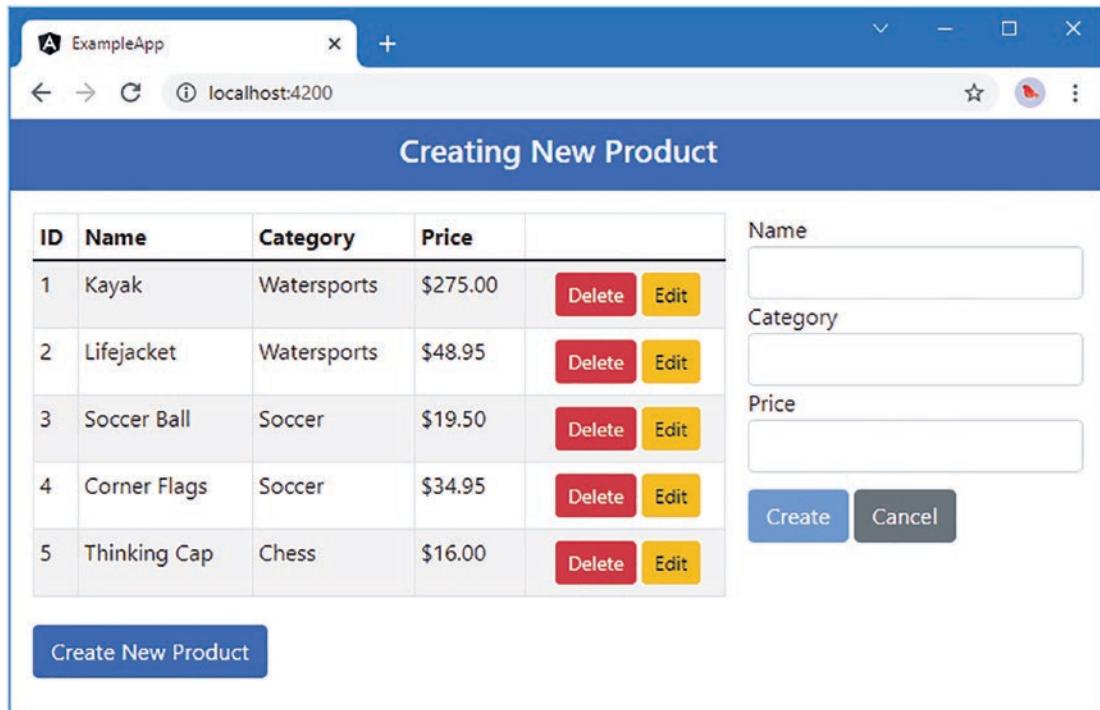


Figure 20-1. Running the example application

Fill out the form and click the Create button to add a product to the repository. You can also click the Edit button to select a product for editing and click the Delete button to remove a button.

Summary

In this chapter, I created the example project that I will use in this part of the book. The basic structure is the same as the example used in earlier chapters but without the redundant code and markup that I used to demonstrate earlier features. In the next chapter, I describe the advanced features Angular provides for working with forms.

CHAPTER 21



Using the Forms API, Part 1

In this chapter, I describe the Angular forms API, which provides an alternative to the template-based approach to forms introduced in Chapter 12. The forms API is a more complicated way of creating forms, but it allows fine-grained control over how forms behave, how they respond to user interaction, and how they are validated. Table 21-1 puts the forms API in context.

Table 21-1. Putting the Forms API in Context

Question	Answer
What is it?	The forms API allows for the creation of reactive forms, which are managed using the code in a component class.
Why is it useful?	The forms API provides a component with more control over the elements in forms and allows their behavior to be customized.
How is it used?	<code>FormControl</code> and <code>FormGroup</code> objects are created by the component class and associated with elements in the template using directives.
Are there any pitfalls or limitations?	The forms API is complex, and additional work is required to ensure that features such as validation behave consistently.
Are there any alternatives?	The forms API is optional. Forms can be defined using the basic features described in Chapter 12.

Table 21-2 summarizes the chapter.

Table 21-2. Chapter Summary

Problem	Solution	Listing
Creating a reactive form	Create a <code>FormControl</code> object in the component class and associate it with a form element in the template using the <code>formControl</code> directive	1-3
Responding to element value changes	Use the observable <code>valueChanges</code> property defined by the <code>FormControl</code> class	4, 5
Managing element state	Use the properties defined by the <code>FormControl</code> class	6
Responding to element validation changes	Use the observable <code>statusChanges</code> property defined by the <code>FormControl</code> class	7-13
Defining multiple related form elements	Use a <code>FormGroup</code> object	14-18, 22-26
Displaying validation messages for controls in a group	Obtain a <code>FormControl</code> object through the enclosing <code>FormGroup</code> object	19-20, 27, 28

Preparing for This Chapter

For this chapter, I will continue using the `exampleApp` project that I created in Chapter 20. No changes are required for this chapter.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

To start the development server, open a command prompt, navigate to the `exampleApp` folder, and run the following command:

```
ng serve
```

Open a new browser window and navigate to `http://localhost:4200` to see the content shown in Figure 21-1.

ID	Name	Category	Price	
1	Kayak	Watersports	\$275.00	<button>Delete</button> <button>Edit</button>
2	Lifejacket	Watersports	\$48.95	<button>Delete</button> <button>Edit</button>
3	Soccer Ball	Soccer	\$19.50	<button>Delete</button> <button>Edit</button>
4	Corner Flags	Soccer	\$34.95	<button>Delete</button> <button>Edit</button>
5	Thinking Cap	Chess	\$16.00	<button>Delete</button> <button>Edit</button>

Name

 Category

 Price

Create Cancel

Create New Product

Figure 21-1. Running the example application

Understanding the Reactive Forms API

The simplest way to use HTML forms is to use Angular two-way bindings to connect input elements to component properties, which is the approach I demonstrated in Chapter 12 and which I used to create the form in the example project in Chapter 20. These are known as *template-driven forms*.

For more complex forms, Angular provides a complete API that exposes the state of HTML forms and allows their data and structure to be managed, known as *reactive forms*. You can get a glimpse of the API in the way that the form element is defined in the `form.component.html` file in the `src/app` folder:

```
...
<form #form="ngForm" (ngSubmit)="submitForm(form)" (reset)="form.resetForm()">
...

```

The template variable named `form` is assigned the value `ngForm`, which is then used in the event bindings, as a method argument in the `ngSubmit` event or to invoke a method in the `reset` event. The `ngForm` value and the events themselves are defined as part of the Angular form API.

One of the themes of this book has been that nothing in Angular is magic. Every feature is implanted using the capabilities of the browser or builds on other Angular features. This includes `ngForm`, which is a directive that acts as a wrapper around a `FormGroup` object, exposing its capabilities using the directive features described in Chapter 12. The `FormGroup` class, which is defined in the `@angular/forms` package, provides an API for working with a form and can be used directly in a component class, allowing forms

to be manipulated in code and not just through HTML elements in a template. In turn, the `FormGroup` is a container for `FormControl` objects, each of which represents an element in the form. As you will learn, the `FormGroup` and `FormControl` classes are the building blocks of the reactive forms API, and the `ngForm` directive, with which you are already familiar, simply presents this API so it can be used easily in templates.

Rebuilding the Form Using the API

The simplest way to get started is with a single form element so you can understand the basic building blocks of the API. The reactive form features require a new module, `ReactiveFormsModule`, as shown in Listing 21-1.

Listing 21-1. Importing a Module in the core.module.ts File in the src/app/core Folder

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "./table.component";
import { FormComponent } from "./form.component";
import { SharedState } from "./sharedState.service";

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule, ReactiveFormsModule],
  declarations: [TableComponent, FormComponent],
  exports: [ModelModule, TableComponent, FormComponent],
  providers: [SharedState]
})
export class CoreModule { }
```

Listing 21-2 simplifies the form template so that it contains only a single `input` element, along with a label and a `div` element.

Listing 21-2. Simplifying the HTML Template in the form.component.html File in the src/app/core Folder

```
<div class="form-group">
  <label>Name</label>
  <input class="form-control" name="name" [formControl]="nameField" />
</div>
```

In addition to simplifying the template, Listing 21-1 makes an important change in the way that the `input` element is configured. The `ngModel` directive isn't used with the forms API, and a different directive is applied:

```
...
<input class="form-control" name="name" [formControl]="nameField" />
...
```

The `formControl` directive creates the relationship between the HTML element in the template and a `FormControl` property in the component class, through which the element will be managed. Listing 21-3 simplifies the component, adds a property for the `formControl` component to use, and takes advantage of one of the features provided by the `FormControl` class.

Listing 21-3. Using the Forms API in the form.components.ts File in the src/app/core Folder

```

import { Component } from "@angular/core";
import { FormControl, NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;

  nameField: FormControl = new FormControl("Initial Value");

  constructor(private model: Model, private state: SharedState,
    private messageService: MessageService) {
    this.state.changes.subscribe((upd) => this.handleStateChange(upd))
    this.messageService.reportMessage(new Message("Creating New Product"));
  }

  handleStateChange(newState: StateUpdate) {
    this.editing = newState.mode == MODES.EDIT;
    if (this.editing && newState.id) {
      Object.assign(this.product, this.model.getProduct(newState.id)
        ?? new Product());
      this.messageService.reportMessage(
        new Message(`Editing ${this.product.name}`));
      this.nameField.setValue(this.product.name);
    } else {
      this.product = new Product();
      this.messageService.reportMessage(new Message("Creating New Product"));
      this.nameField.setValue("");
    }
  }

  // submitForm(form: NgForm) {
  //   if (form.valid) {
  //     this.model.saveProduct(this.product);
  //     this.product = new Product();
  //     form.resetForm();
  //   }
  // }
}

```

Individual form controls are represented by `FormControl` properties, and Listing 21-3 defines such a property with the name specified by the `formControl` directive in Listing 21-2, creating the relationship between the HTML element and the component class. The constructor accepts an optional argument that sets the initial value for the element:

```
...
nameField: FormControl = new FormControl("Initial Value");
...
```

When the `input` element is presented to the user, it will contain the string `Initial Value`.

The reactive forms API provides direct access to the features that were previously managed through the `ngForm` and `ngModel` directives. In this case, I have used the `setValue` method to set the contents of the `input` element when the user clicks an `Edit` button:

```
...
this.nameField.setValue(this.product.name);
...
```

As its name suggests, the `setValue` method sets the value of the form control, which was previously done by the `ngModel` directive. The `setValue` method is one of the basic features provided by the `FormControl` class, the most useful of which are described in Table 21-3 and which I describe in the following sections.

Note Some of the methods described in Table 21-3 and later tables take an optional argument that manages the effect of changes. I don't describe these options because they are not typically required, but see the Angular API description for the `FormControl` class (<https://angular.io/api/forms/FormControl>) for details.

Table 21-3. Useful Basic `FormControl` Members

Name	Description
<code>value</code>	This property returns the current value of the form control, defined using the <code>any</code> type.
<code>setValue(value)</code>	This method sets the value of the form control.
<code>valueChanges</code>	This property returns an <code>Observable<any></code> , through which changes can be observed.
<code>enabled</code>	This property returns <code>true</code> if the form control is enabled.
<code>disabled</code>	This property returns <code>true</code> if the form control is disabled.
<code>enable()</code>	This method enables the form control.
<code>disable()</code>	This method disables the form control.
<code>reset(value)</code>	This method resets the form control, with an optional value. The form control will be reset to its default state if the value argument is omitted.

The overall effect is to transfer control of the `input` element from the template to the component, through the `FormControl` property. When the form component is displayed, the `input` element is populated with an initial value, which is replaced when the user clicks one of the `Edit` buttons, as shown in Figure 21-2.

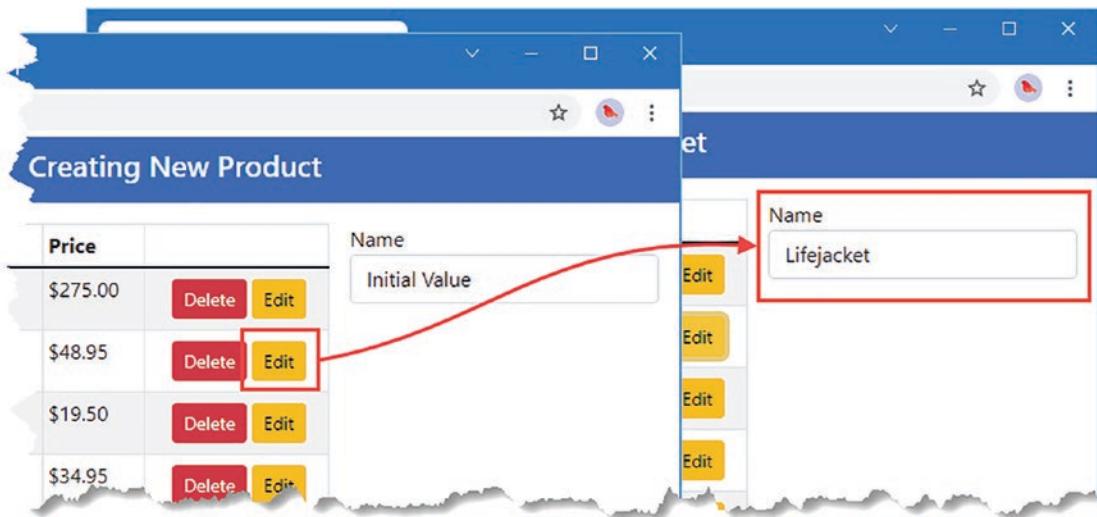


Figure 21-2. Using a `FormControl`

Responding to Form Control Changes

The `valueChanges` property returns an observable that emits new values from the form control. Components can observe these changes to respond to user interaction, as shown in Listing 21-4.

Listing 21-4. Observing Changes in the `form.component.ts` File in the `src/app/core` Folder

```
import { Component } from "@angular/core";
import { FormControl, NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;

  nameField: FormControl = new FormControl("Initial Value");
```

```

constructor(private model: Model, private state: SharedState,
    private messageService: MessageService) {
    this.state.changes.subscribe((upd) => this.handleStateChange(upd))
    this.messageService.reportMessage(new Message("Creating New Product"));
}

ngOnInit() {
    this.nameField.valueChanges.subscribe(newValue => {
        this.messageService.reportMessage(new Message(newValue || "(Empty)"));
    });
}

handleStateChange(newState: StateUpdate) {
    this.editing = newState.mode == MODES.EDIT;
    if (this.editing && newState.id) {
        Object.assign(this.product, this.model.getProduct(newState.id)
            ?? new Product());
        this.messageService.reportMessage(
            new Message(`Editing ${this.product.name}`));
        this.nameField.setValue(this.product.name);
    } else {
        this.product = new Product();
        this.messageService.reportMessage(new Message("Creating New Product"));
        this.nameField.setValue("");
    }
}
}

```

In the ngOnInit method, the component subscribes to the Observable<any> returned by the valueChanges property and passes on the values it receives to the message service. As the user types into the input element, the changed value is displayed at the top of the layout, as shown in Figure 21-3.

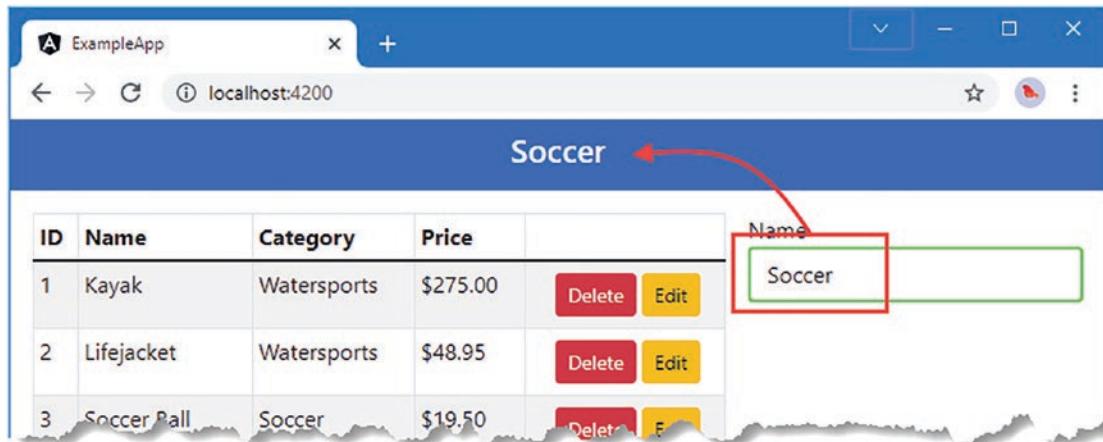


Figure 21-3. Responding to form control changes

By default, the observable will emit a new event in response to the HTML element's change event, but this can be altered by configuring the `FormControl` with a constructor argument, as shown in Listing 21-5.

Listing 21-5. Configuring a `FormControl` in the `form.component.ts` File in the `src/app/core` Folder

```
import { Component, Input } from "@angular/core";
import { NgForm, FormControl } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;

  nameField: FormControl = new FormControl("Initial Value", {
    updateOn: "blur"
  });

  // ...methods omitted for brevity...
}
```

The new argument to the `FormControl` constructor implements the `AbstractControlOptions` interface, which defines the properties described in Table 21-4. Those properties are all optional, which means you can omit those you do not need to change.

Table 21-4. The Properties Defined by the `AbstractControlOptions` Interface

Name	Description
validators	This property is used to configure the validation for the form control, as described in the “Managing Control Validation” section.
asyncValidators	This property is used to configure the async validation for the form control, as described in Chapter 22.
updateOn	This property is used to configure when the <code>valueChanges</code> observable will emit a new value. It can be set to <code>change</code> , the default; <code>blur</code> ; or <code>submit</code> . The <code>submit</code> value is used with form elements.

In Listing 21-5, I used the `blur` value for the `updateOn` property, which means that the observable will emit new values only when the `input` element loses focus, such as when the user tabs to another element, as shown in Figure 21-4.



Figure 21-4. Changing the update setting

Managing Control State

In Chapter 12, I showed you how form elements are added to classes to denote their state. The `FormControl` class defines properties that indicate the state of the HTML element and methods for manually changing the state, as described in Table 21-5.

Table 21-5. The `FormControl` Members for Element State

Name	Description
<code>untouched</code>	This property returns <code>true</code> if the HTML element is untouched, meaning that the element has not been selected.
<code>touched</code>	This property returns <code>true</code> if the HTML element has been touched, meaning that the element has been selected.
<code>markAsTouched()</code>	Calling method marks the element as touched.
<code>markAsUntouched()</code>	Calling this method marks the element as untouched.
<code>pristine</code>	This property returns <code>true</code> if the element contents have not been edited by the user.
<code>dirty</code>	This property returns <code>true</code> if the element contents have been edited by the user.
<code>markAsPristine()</code>	Calling this method marks the element as pristine.
<code>markAsDirty()</code>	Calling this method marks the element as dirty.

One benefit of using the reactive forms API is that you can control the way that the form features are applied, tailoring the behavior to the needs of your project. As a simple demonstration, Listing 21-6 changes the state of the element based on the number of characters in the value.

Listing 21-6. Changing Element State in the form.component.ts File in the src/app/core Folder

```

import { Component } from "@angular/core";
import { FormControl, NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";

@Component({
    selector: "paForm",
    templateUrl: "form.component.html",
    styleUrls: ["form.component.css"]
})
export class FormComponent {
    product: Product = new Product();
    editing: boolean = false;

    nameField: FormControl = new FormControl("Initial Value", {
        updateOn: "change"
    });

    constructor(private model: Model, private state: SharedState,
        private messageService: MessageService) {
        this.state.changes.subscribe((upd) => this.handleStateChange(upd))
        this.messageService.reportMessage(new Message("Creating New Product"));
    }

    ngOnInit() {
        this.nameField.valueChanges.subscribe(newValue => {
            this.messageService.reportMessage(new Message(newValue || "(Empty)"));
            if (typeof(newValue) == "string" && newValue.length % 2 == 0) {
                this.nameField.markAsPristine();
            }
        });
    }

    handleStateChange(newState: StateUpdate) {
        this.editing = newState.mode == MODES.EDIT;
        if (this.editing && newState.id) {
            Object.assign(this.product, this.model.getProduct(newState.id)
                ?? new Product());
            this.messageService.reportMessage(
                new Message(`Editing ${this.product.name}`));
            this.nameField.setValue(this.product.name);
        } else {
            this.product = new Product();
            this.messageService.reportMessage(new Message("Creating New Product"));
            this.nameField.setValue("");
        }
    }
}

```

I have changed the `updateOn` property so that a new value is emitted via the observable after every change, and I added an `if` expression to the subscriber function that calls the `FormControl.markAsPristine` method if the length of the character is an even number. The effect is that the border of the input element toggles on and off as the user types, as shown in Figure 21-5.

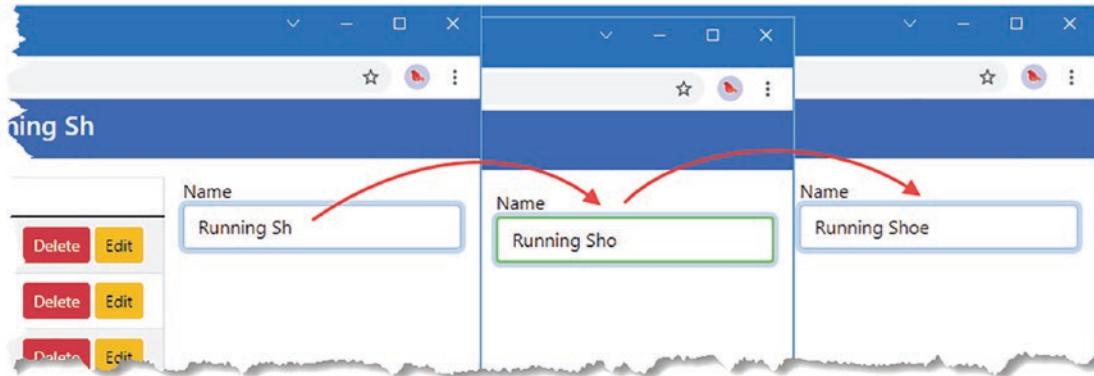


Figure 21-5. Changing element state

The reason the border color changes is that Angular marks form elements as valid, even if no validation requirements have been applied. This means that for an odd number of characters, the input element is added to the `ng-valid`, `ng-touched`, and `ng-dirty` classes, like this:

```
...
<input name="name" class="form-control ng-valid ng-touched ng-dirty">
...
```

This combination is matched by the selector for one of the styles defined in the `form.component.css` file, which I added to the project in Chapter 20:

```
...
input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
```

When there is an even number of characters, the call to the `markAsPristine` method creates a different combination of element classes:

```
...
<input name="name" class="form-control ng-valid ng-touched ng-pristine">
...
```

This doesn't match the CSS selector, and no border is displayed. This example demonstrates that you can customize the behavior of form elements by using the reactive forms API, even if this particular behavior is unlikely to be required in many projects.

Managing Control Validation

Form elements can be subject to validation, even when using the reactive forms API. Validation constraints can be added to the template, as described in Chapter 12, or applied through the `FormControl` constructor, using the `validators` and `asyncValidators` properties of the `AbstractControlOptions` interface.

Listing 21-7 uses this feature to apply validation to the example form element, using the `validators` property. (I explain the use of the `asyncValidators` property in Chapter 22.)

Listing 21-7. Applying Validation in the `form.component.ts` File in the `src/app/core` Folder

```
import { Component, Input } from "@angular/core";
import { NgForm, FormControl, Validators } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;

  nameField: FormControl = new FormControl("", {
    validators: [
      Validators.required,
      Validators.minLength(3),
      Validators.pattern("^[A-Za-z ]+$")
    ],
    updateOn: "change"
  });

  // ...constructor and methods omitted for brevity...
}
```

The built-in validators are defined as static properties of the `Validators` class, where each property name corresponds to the validation attributes described in Chapter 12. Validation rules are specified using the `validators` property of the `AbstractControlOptions` constructor argument, which is assigned an array of validators. In Listing 21-7, I used the `required`, `minLength`, and `pattern` properties to set the validation policy for the input element. I have also changed the first constructor argument so that the `input` element is empty when it is first presented to the user.

In addition to the constructor, the validators applied to a form control can be managed through the `FormControl` properties and methods described in Table 21-6. For each entry in the table, there is a corresponding member for managing asynchronous validators, which I describe in Chapter 22.

Table 21-6. The FormControl Members for Managing Validators

Name	Description
validator	This property returns a function that combines all of the configured validators so that the form control can be validated with a single function call.
hasValidator(v)	This function returns true if the form control has been configured with the specified validator.
setValidators(vals)	This method sets the form control validators. The argument can be a single validator or an array of validators.
addValidators(v)	This method adds one or more validators to the form control.
removeValidators(v)	This method removes one or more validators from the control.
clearValidators()	This method removes all of the validators from the form control.

The validation state of a FormControl is determined and managed through the members described in Table 21-7.

Table 21-7. The FormControl Members for Validation State

Name	Description
status	This property returns the validation state of the form control, expressed using a FormControlStatus value, which will be VALID, INVALID, PENDING, or DISABLED.
statusChanges	This property returns an Observable<FormControlStatus>, which will emit a FormControlStatus value when the state of the form control changes.
valid	This property returns true if the form control's value passes validation.
invalid	This property returns true if the form control's value fails validation.
pending	This property returns true if the form control's value is being validated asynchronously, as described in Chapter 22.
errors	This property returns a ValidationErrors object that contains the errors generated by the form control's validators, or null if there are no errors.
getError(v)	This method returns the error message, if there is one, for the specified validator. This method accepts an optional path for use with nested form controls, as described in the “Working with Multiple Form Controls” section.
hasError(v)	This method returns true if the specified validator has generated an error message. This method accepts an optional path for use with nested form controls, as described in the “Working with Multiple Form Controls” section.
setErrors(errs)	This method is used to add errors to the form control's validation status, which is useful when performing manual validation in the component. This method accepts an optional path for use with nested form controls, as described in the “Working with Multiple Form Controls” section.

The effect of the validators configured in Listing 21-7 can be determined through the features described in Table 21-7. Listing 21-8 uses a subscription to the statusChanges observable to generate messages summarizing the validation state of the form control.

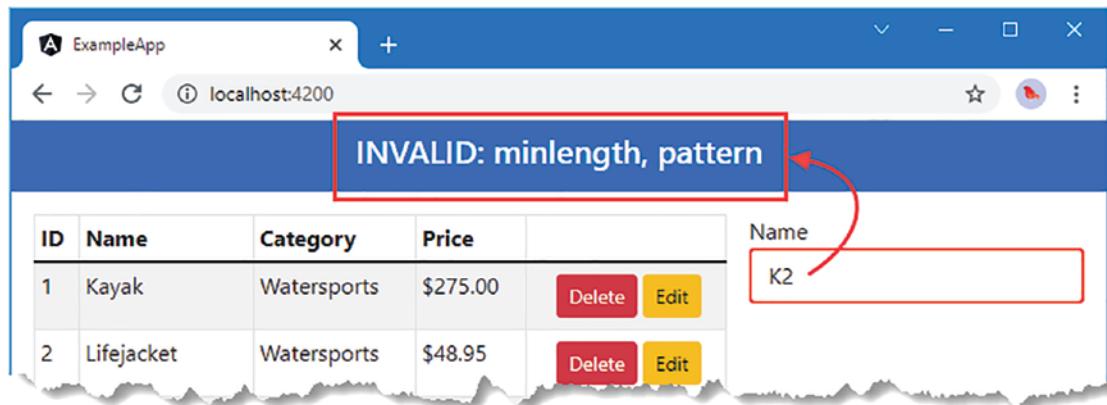
Listing 21-8. Generating Validation Messages in the form.component.ts File in the src/app/core Folder

```

...
ngOnInit() {
  this.nameField.statusChanges.subscribe(newStatus => {
    if (newStatus == "INVALID" && this.nameField.errors != null) {
      let errs = Object.keys(this.nameField.errors).join(", ");
      this.messageService.reportMessage(new Message(`INVALID: ${errs}`));
    } else {
      this.messageService.reportMessage(new Message(newStatus));
    }
  })
  // this.nameField.valueChanges.subscribe(newValue => {
  //   this.messageService.reportMessage(new Message(newValue || "(Empty)"));
  //   if (typeof(newValue) == "string" && newValue.length % 2 == 0) {
  //     this.nameField.markAsPristine();
  //   }
  // });
}
...

```

In response to status changes, a message is sent that details the status and, if it is INVALID, includes a list of the validators that have reported an error, as shown in Figure 21-6. (See Chapter 12 for details of how to process a ValidationErrors object to display validation messages that are more usefully presented to the user.)

**Figure 21-6.** Responding to status changes

The message shown in Figure 21-6 isn't especially useful to the user, but the `FormControl` directive uses the `exportAs` property to provide an identifier named `ngForm` for use in template variables, and this can be used to generate more helpful validation messages for a control. To prepare, add a file named `validation_helper.ts` to the `src/app/core` folder with the content shown in Listing 21-9, which creates a pipe to format validation messages.

Listing 21-9. The Contents of the validation_helper.ts File in the src/app/core Folder

```

import { Pipe } from "@angular/core";
import { FormControl, ValidationErrors } from "@angular/forms";

@Pipe({
  name: "validationFormat"
})
export class ValidationHelper {

  transform(source: any, name: any) : string[] {
    if (source instanceof FormControl) {
      return this.formatMessages((source as FormControl).errors, name)
    }
    return this.formatMessages(source as ValidationErrors, name)
  }

  formatMessages(errors: ValidationErrors | null, name: string): string[] {
    let messages: string[] = [];
    for (let errorName in errors) {
      switch (errorName) {
        case "required":
          messages.push(`You must enter a ${name}`);
          break;
        case "minlength":
          messages.push(`A ${name} must be at least
            ${errors['minlength'].requiredLength}
            characters`);
          break;
        case "pattern":
          messages.push(`The ${name} contains
            illegal characters`);
          break;
      }
    }
    return messages;
  }
}

```

This code is based on the approach I took in Chapter 12 to generate user-friendly messages for template-driven forms. Listing 21-10 registers the pipe so that it will be available in the rest of the module.

Listing 21-10. Registering a Pipe in the core.modules.ts File in the src/app/core Folder

```

import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "./table.component";
import { FormComponent } from "./form.component";
import { SharedState } from "./sharedState.service";
import { ValidationHelper } from "./validation_helper";

```

```

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule, ReactiveFormsModule],
  declarations: [TableComponent, FormComponent, ValidationHelper],
  exports: [ModelModule, TableComponent, FormComponent],
  providers: [SharedState]
})
export class CoreModule { }

```

Listing 21-11 uses a template variable to obtain a reference for the FormControl object, which is used to get validation messages that can be displayed to the user.

Listing 21-11. Generating Error Messages in the form.component.html File in the src/app/core Folder

```

<div class="form-group">
  <label>Name</label>
  <input class="form-control" name="name" [FormControl]="nameField"
    #name="ngForm" />
  <ul class="text-danger list-unstyled mt-1" *ngIf="name.dirty && name.invalid">
    <li *ngFor="let err of name.errors | validationFormat:'name'">
      {{ err }}
    </li>
  </ul>
</div>

```

The FormControl directive defines properties that correspond to those described in Table 21-5 and Table 21-7, which provides access to the validation state and errors, which are formatted using the new pipe. Validation messages are displayed to the user, as shown in Figure 21-7, achieving the same result as with the template-based form from earlier chapters.

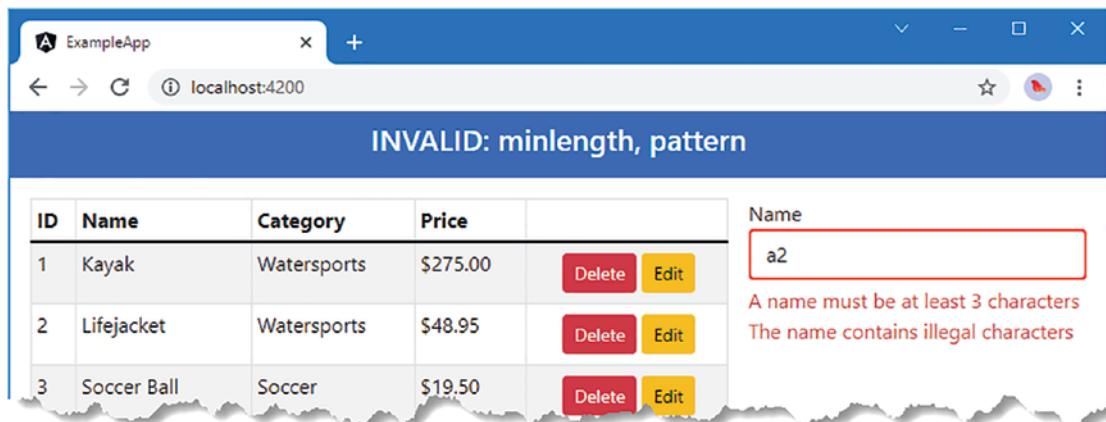


Figure 21-7. Displaying validation messages

Adding Additional Controls

The advantage of using the forms API is that you can customize the way that your forms work, and this extends to using the state of one control to determine the state of another. To prepare, Listing 21-12 introduces another input element to the template.

Listing 21-12. Adding an Element in the form.component.html File in the src/app/core Folder

```
<div class="form-group">
  <label>Name</label>
  <input class="form-control" name="name" [FormControl]="nameField"
    #name="ngForm" />
  <ul class="text-danger list-unstyled mt-1" *ngIf="name.dirty && name.invalid">
    <li *ngFor="let err of name.errors | validationFormat:'name'">
      {{ err }}
    </li>
  </ul>
</div>

<div class="form-group">
  <label>Category</label>
  <input class="form-control" name="category" [FormControl]="categoryField" />
</div>
```

The new input element is configured with a `FormControl` binding that specifies a property named `categoryField`. Listing 21-13 defines this property and uses the features provided by the `FormControl` class to change the element's state based on the `name` field.

Listing 21-13. Adding a FormControl in the form.component.ts File in the src/app/core Folder

```
import { Component } from "@angular/core";
import { FormControl, NgForm, Validators } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;

  nameField: FormControl = new FormControl("", {
    validators: [
      Validators.required,
      Validators.minLength(3),
    ]
  })
}
```

```

        Validators.pattern("^[A-Za-z ]+$")
    ],
    updateOn: "change"
});
categoryField: FormControl = new FormControl();

constructor(private model: Model, private state: SharedState,
    private messageService: MessageService) {
    this.state.changes.subscribe((upd) => this.handleStateChange(upd))
    this.messageService.reportMessage(new Message("Creating New Product"));
}

ngOnInit() {
    this.nameField.statusChanges.subscribe(newStatus => {
        if (newStatus == "INVALID") {
            this.categoryField.disable();
        } else {
            this.categoryField.enable();
        }
    })
}

handleStateChange(newState: StateUpdate) {
    this.editing = newState.mode == MODES.EDIT;
    if (this.editing && newState.id) {
        Object.assign(this.product, this.model.getProduct(newState.id)
            ?? new Product());
        this.messageService.reportMessage(
            new Message(`Editing ${this.product.name}`));
        this.nameField.setValue(this.product.name);
        this.categoryField.setValue(this.product.category);
    } else {
        this.product = new Product();
        this.messageService.reportMessage(new Message("Creating New Product"));
        this.nameField.setValue("");
        this.categoryField.setValue("");
    }
}
}
}

```

The category input element is enabled and disabled based on the validation status of the name element. To see the effect, start typing characters into the Name field, which will be disabled by the error produced by the minlength validator. Once the minimum length is reached, the Name field will pass validation, and the Category field will be enabled, as shown in Figure 21-8.

ID	Name	Category	Price		
1	Kayak	Watersports	\$275.00	<button>Delete</button>	<button>Edit</button>
2	Lifejacket	Watersports	\$48.95	<button>Delete</button>	<button>Edit</button>
3	Soccer Ball	Soccer	\$19.50	<button>Delete</button>	<button>Edit</button>
4	Corner Flags	Soccer	\$34.95	<button>Delete</button>	<button>Edit</button>
5	Thinking Cap	Chess	\$16.00	<button>Delete</button>	<button>Edit</button>

Create New Product

Figure 21-8. Working with multiple controls

Working with Multiple Form Controls

Manipulating individual FormControl objects can be a powerful technique, but it can also be cumbersome in more complex forms, where there can be many objects to create and manage. The reactive forms API includes the FormGroup class, which represents a group of form controls, which can be manipulated individually or as a combined group. Listing 21-14 introduces a FormGroup property to the example component.

Listing 21-14. Using a FormGroup in the form.component.ts File in the src/app/core Folder

```
import { Component } from "@angular/core";
import { FormControl, NgForm, Validators, FormGroup } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
```

```

export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;

  nameField: FormControl = new FormControl("", {
    validators: [
      Validators.required,
      Validators.minLength(3),
      Validators.pattern("^[A-Za-z ]+$")
    ],
    updateOn: "change"
  });
  categoryField: FormControl = new FormControl();

  productForm: FormGroup = new FormGroup({
    name: this.nameField, category: this.categoryField
  });

  constructor(private model: Model, private state: SharedState,
    private messageService: MessageService) {
    this.state.changes.subscribe((upd) => this.handleStateChange(upd))
    this.messageService.reportMessage(new Message("Creating New Product"));
  }

  ngOnInit() {
    this.productForm.statusChanges.subscribe(newStatus => {
      if (newStatus == "INVALID") {
        let invalidControls: string[] = [];
        for (let controlName in this.productForm.controls) {
          if (this.productForm.controls[controlName].invalid) {
            invalidControls.push(controlName)
          }
        }
        this.messageService.reportMessage(
          new Message(`INVALID: ${invalidControls.join(", ")}`))
      } else {
        this.messageService.reportMessage(new Message(newStatus));
      }
    })
  }

  handleStateChange(newState: StateUpdate) {
    this.editing = newState.mode == MODES.EDIT;
    if (this.editing && newState.id) {
      Object.assign(this.product, this.model.getProduct(newState.id)
        ?? new Product());
      this.messageService.reportMessage(
        new Message(`Editing ${this.product.name}`));
      // this.nameField.setValue(this.product.name);
      // this.categoryField.setValue(this.product.category);
    }
  }
}

```

```

    } else {
        this.product = new Product();
        this.messageService.reportMessage(new Message("Creating New Product"));
        // this.nameField.setValue("");
        // this.categoryField.setValue("");
    }
this.productForm.reset(this.product);
}
}

```

This is the simplest use of a `FormGroup`, where a property is used to group existing `FormControl` objects so they can be processed collectively. The individual `FormControl` objects are passed to the `FormGroup` constructor, in a map that assigns each a key. Controls can also be added, removed, and inspected in the `FormGroup` using the members defined in Table 21-8.

Table 21-8. *FormGroup Members for Adding and Removing Controls*

Name	Description
<code>addControl(name, ctrl)</code>	This method adds a control to the <code>FormGroup</code> with the specified name. No action is taken if there is already a control with this name.
<code>setControl(name, ctrl)</code>	This method adds a control to the <code>FormGroup</code> with the specified name, replacing any existing control with this name.
<code>removeControl(name)</code>	This method removes the control with the specified name.
<code>controls</code>	This property returns a map containing the controls in the group, using their names as keys.
<code>get(name)</code>	This property returns the control with the specified name.

In Listing 21-14, I created a `FormGroup` and added the existing `FormControl` objects with `name` and `category` keys:

```

...
productForm: FormGroup = new FormGroup({
    name: this.nameField, category: this.categoryField
});
...

```

The names used to register `FormControl` objects make it easy to get and set values for all the individual controls in a single step, using the property and methods described in Table 21-9.

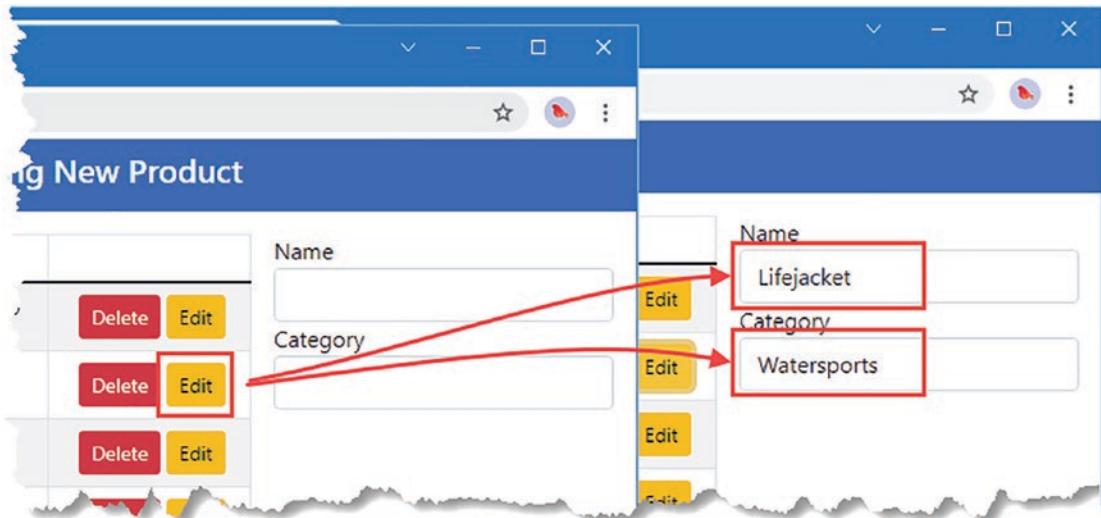
Table 21-9. FormGroup Methods for Managing Control Values

Name	Description
value	This method returns an object containing the values of the form controls in the group, using the names given to each control as the names of the properties.
setValue(val)	This method sets the contents of the form controls using an object, whose property names correspond to the names given to each control. The specified value object must define properties for all the form controls in the group.
patchValue(val)	This method sets the contents of the form controls using an object, whose property names correspond to the names given to each control. Unlike the setValue method, values are not required for all form controls.
reset(val)	This method resets the form to its pristine and untouched state and uses the specified value to populate the form controls.

Being able to get and set all the form control values together makes it easier to work with complex forms. I used the reset method to populate or clear the form controls when the user clicks the Create New Product or Edit button:

```
...
this.productForm.reset(this.product);
...
```

The reset method looks for properties in the object it receives whose names correspond to those used for regular FormControl objects with the FormGroup. The value of each property is used to set the control value, as shown in Figure 21-9.

**Figure 21-9.** Using a form group

The `FormGroup` and `FormControl` classes share a common base class, which means that many of the properties and methods provided by `FormControl` are also available on a `FormGroup` object, but applied to all of the controls in the group. In Listing 21-14, I subscribed to the `FormGroup`'s `statusChanges` observable to receive events that indicate the status of the form, which Angular determines by examining all of the controls in the group:

```
...
this.productForm.statusChanges.subscribe(newStatus => {
  if (newStatus == "INVALID") {
...
}
```

If any of the individual controls are invalid, then the overall form status will be invalid, which allows me to assess the validation results without needing to inspect controls individually. But access to the individual controls is still available using the `controls` property, which lets me build up a list of invalid controls:

```
...
for (let controlName in this.productForm.controls) {
  if (this.productForm.controls[controlName].invalid) {
    invalidControls.push(controlName)
  }
}
...
}
```

The result is that a list of invalid form controls is shown to the user, as illustrated by Figure 21-10.

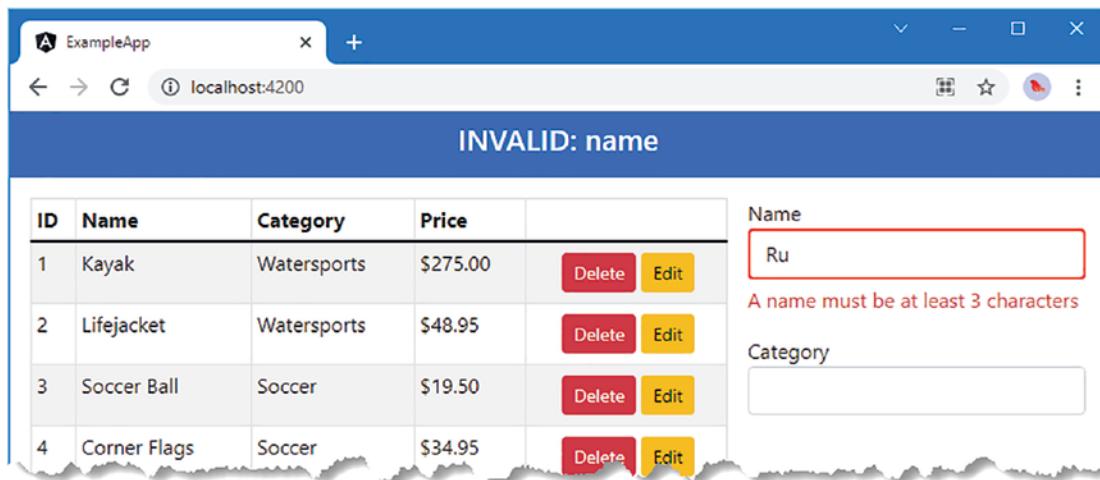


Figure 21-10. Assessing validation status using the form group

Using a Form Group with a Form Element

The `FormGroup` directive associates a `FormGroup` object with an element in the template, in the same way the `FormControl` directive is used with a `FormControl` object, as shown in Listing 21-15.

Listing 21-15. Introducing a Form Element in the form.component.html File in the src/app/core Folder

```
<form [formGroup]="productForm">
  <div class="form-group">
    <label>Name</label>
    <input class="form-control" formControlName="name" />
    <!-- <ul class="text-danger list-unstyled mt-1" *ngIf="name.dirty -->
    <!--     && name.invalid"> -->
    <!--      <li *ngFor="let err of name.errors | validationFormat:'name'" -->
    <!--        {{ err }} -->
    <!--      </li> -->
    <!-- </ul> -->
  </div>

  <div class="form-group">
    <label>Category</label>
    <input class="form-control" formControlName="category" />
  </div>
</form>
```

The `FormGroup` directive is used to specify the `FormGroup` object, and the individual form elements are associated with their `FormControl` objects using the `formControlName` attribute, specifying the name used when adding the `FormControl` to the `FormGroup`.

Using the `formControlName` attribute means that I don't have to define properties for each `FormControl` object in the controller class, allowing me to simplify the code, as shown in Listing 21-16.

Listing 21-16. Removing Properties in the form.component.ts File in the src/app/core Folder

```
import { Component } from "@angular/core";
import { FormControl, NgForm, Validators, FormGroup } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;

  // nameField: FormControl = new FormControl("", {
  //   validators: [
  //     Validators.required,
  //     Validators.minLength(3),
  //     Validators.pattern("^[A-Za-z ]+$")
  //   ],
}
```

```

//      updateOn: "change"
// });
// categoryField: FormControl = new FormControl();

// productForm: FormGroup = new FormGroup({
//   name: this.nameField, category: this.categoryField
// });

productForm: FormGroup = new FormGroup({
  name: new FormControl("", {
    validators: [
      Validators.required,
      Validators.minLength(3),
      Validators.pattern("^[A-Za-z ]+$")
    ],
    updateOn: "change"
  }),
  category: new FormControl()
});

// ...constructor and methods omitted for brevity...
}

```

There is no change in the output produced by the application, and this change just consolidates the individual form controls within the form group.

Accessing the Form Group from the Template

In addition to simplifying the application code, the `FormGroup` directive defines some useful properties that allow me to complete the transition to the reactive forms API, restoring the features that were present when the form was managed solely through a template. Table 21-10 describes the most useful features provided by the `FormGroup` directive.

Table 21-10. Use Features Provided by the `FormGroup` Directive

Name	Description
ngSubmit	This event is triggered when the form is submitted.
submitted	This property returns <code>true</code> if the form has been submitted.
control	This property returns the <code>FormControl</code> object that has been associated with the directive.

These features ensure that the reactive forms API can still be used effectively in a template, while still providing the ability to customize the form behavior in the component class. Listing 21-17 restores the price input element that was present at the start of the chapter, along with the buttons that submit and reset the form.

Listing 21-17. Using the FormGroup Features in the form.component.html File in the src/app/core Folder

```
<form [formGroup]="productForm" #form="ngForm"
      (ngSubmit)="submitForm()" (reset)="resetForm()">

  <div class="form-group">
    <label>Name</label>
    <input class="form-control" formControlName="name" />
  </div>

  <div class="form-group">
    <label>Category</label>
    <input class="form-control" formControlName="category" />
  </div>

  <div class="form-group">
    <label>Price</label>
    <input class="form-control" formControlName="price" />
  </div>

  <div class="mt-2">
    <button type="submit" class="btn btn-primary"
            [class.btn-warning]="editing"
            [disabled]="form.invalid">
      {{editing ? "Save" : "Create"}}
    </button>
    <button type="reset" class="btn btn-secondary m-1">Cancel</button>
  </div>
</form>
```

I used the directive's `ngForm` property to create a template variable named `form`, through which I can check the overall validation status for the Save/Create button. The `ngSubmit` form is used to invoke a method named `submitForm`, and I used the `form` element's `reset` event to invoke a method named `resetForm`. Listing 21-18 shows the changes required to the component class to support the additions to the template.

Listing 21-18. Completing the Form in the form.component.ts File in the src/app/core Folder

```
import { Component } from "@angular/core";
import { FormControl, NgForm, Validators, FormGroup } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;
```

```

productForm: FormGroup = new FormGroup({
  name: new FormControl("", {
    validators: [
      Validators.required,
      Validators.minLength(3),
      Validators.pattern("^[A-Za-z ]+$")
    ],
    updateOn: "change"
  }),
  category: new FormControl("", { validators: Validators.required }),
  price: new FormControl("", {
    validators: [Validators.required, Validators.pattern("^[0-9\\.]+$")]
  })
});

constructor(private model: Model, private state: SharedState,
  private messageService: MessageService) {
  this.state.changes.subscribe((upd) => this.handleStateChange(upd))
  this.messageService.reportMessage(new Message("Creating New Product"));
}

// ngOnInit() {
//   this.productForm.statusChanges.subscribe(newStatus => {
//     if (newStatus == "INVALID") {
//       let invalidControls: string[] = [];
//       for (let controlName in this.productForm.controls) {
//         if (this.productForm.controls[controlName].invalid) {
//           invalidControls.push(controlName)
//         }
//       }
//       this.messageService.reportMessage(
//         new Message(`INVALID: ${invalidControls.join(", ")}`))
//     } else {
//       this.messageService.reportMessage(new Message(newStatus));
//     }
//   })
// }

handleStateChange(newState: StateUpdate) {
  this.editing = newState.mode == MODES.EDIT;
  if (this.editing && newState.id) {
    Object.assign(this.product, this.model.getProduct(newState.id)
      ?? new Product());
    this.messageService.reportMessage(
      new Message(`Editing ${this.product.name}`));
  } else {
    this.product = new Product();
    this.messageService.reportMessage(new Message("Creating New Product"));
  }
  this.productForm.reset(this.product);
}

```

```

submitForm() {
  if (this.productForm.valid) {
    Object.assign(this.product, this.productForm.value);
    this.model.saveProduct(this.product);
    this.product = new Product();
    this.productForm.reset();
  }
}

resetForm() {
  this.editing = true;
  this.product = new Product();
  this.productForm.reset();
}
}

```

This listing adds a FormControl named `price`, adds validation to the `category` control, and defines the `submitForm` and `resetForm` method that will be invoked by the event bindings defined in Listing 21-17. The effect is to complete the form and restore the functionality that was previously defined using only the template form features, as shown in Figure 21-11.

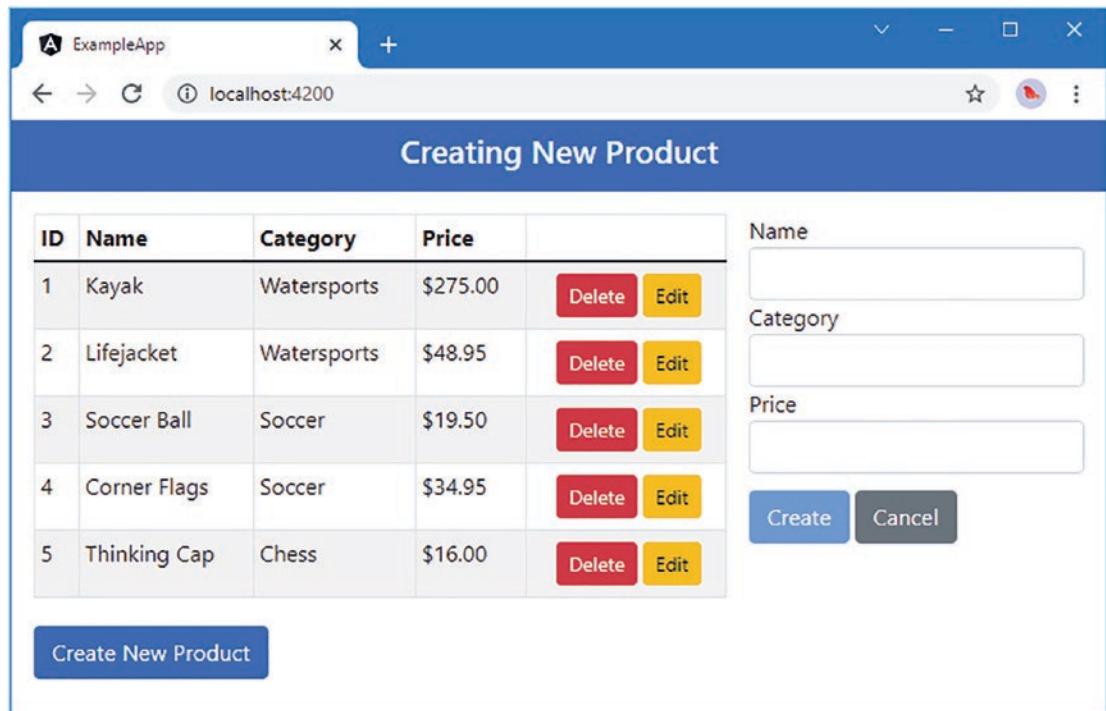


Figure 21-11. Using the reactive forms API

Displaying Validation Messages with a Form Group

The `formControlName` directive doesn't export an identifier for use in a template variable, which complicates the process of displaying validation messages. Instead, errors must be obtained through the `FormGroup`, using the optional path argument for the error-related methods, which I have repeated in Table 21-11 for quick reference.

Table 21-11. The `FormGroup` Methods for Dealing with Errors

Name	Description
<code>getError(v, path)</code>	This method returns the error message, if there is one, for the specified validator. The optional path argument is used to identify the control.
<code>hasError(v, path)</code>	This method returns true if the specified validator has generated an error message. The optional path argument is used to identify the control.

These methods require the error to be specified, which means that it is possible to determine if a specific control has a specific error, like this:

```
...
form.getError("required", "category")
...

```

This expression would return details of errors reported by the required validator on the category control, which is identified by the name used to register the control in the `FormGroup`. This isn't a useful approach for the example application, where I want to display validation messages by getting all of the errors for a single `FormControl` object. For this, I can use the `get` method defined by the `FormGroup` class, although this can produce verbose and repetitive templates and so the best approach is to create a directive. Add a file named `validationErrors.directive.ts` to the `src/app/core` folder with the code shown in Listing 21-19.

Listing 21-19. The Contents of the `validationErrors.directive.ts` File in the `src/app/core` Folder

```
import { Directive, Input, TemplateRef, ViewContainerRef } from "@angular/core";
import { FormGroup } from "@angular/forms";
import { ValidationHelper } from "./validation_helper";

@Directive({
  selector: "[validationErrors]"
})
export class ValidationErrorsDirective {

  constructor(private container: ViewContainerRef,
    private template: TemplateRef<Object>) {}

  @Input("validationErrorsControl")
  name: string = ""

  @Input("validationErrorsLabel")
  label?: string;

  @Input("validationErrors")
  formGroup?: FormGroup;
}
```

```
ngOnInit() {
    let formatter = new ValidationHelper();
    if (this.formGroup && this.name) {
        let control = this.formGroup?.get(this.name);
        if (control) {
            control.statusChanges.subscribe(() => {
                if (this.container.length > 0) {
                    this.container.clear();
                }
                if (control && control.dirty && control.invalid
                    && control.errors) {
                    formatter.formatMessages(control.errors,
                        this.label ?? this.name).forEach(err => {
                            this.container.createEmbeddedView(this.template,
                                { $implicit: err });
                        })
                }
            })
        }
    }
}
```

The new directive obtains a `FormControl` object via its `FormGroup` and subscribes to the observable for status changes. Each time the status changes, the validation state is checked, and any error messages are formatted and used to generate template content.

This is a simple task in code, but it is more difficult to achieve in a template without creating long expressions. Listing 21-20 registers the directive so that it can be used in the module.

Listing 21-20. Registering a Directive in the core.module.ts File in the src/app/core Folder

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "./table.component";
import { FormComponent } from "./form.component";
import { SharedState } from "./sharedState.service";
import { ValidationHelper } from "./validation_helper";
import { ValidationErrorsDirective } from "./validationErrors.directive";

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule, ReactiveFormsModule],
  declarations: [TableComponent, FormComponent, ValidationHelper,
    ValidationErrorsDirective],
  exports: [ModelModule, TableComponent, FormComponent],
  providers: [SharedState]
})
export class CoreModule { }
```

Listing 21-21 uses the new directive to display validation messages for each form element.

Listing 21-21. Displaying Validation Messages in the form.component.html File in the src/app/core Folder

```
<form [formGroup]="productForm" #form="ngForm"
      (ngSubmit)="submitForm()" (reset)="resetForm()">
  <div class="form-group">
    <label>Name</label>
    <input class="form-control" formControlName="name" />
    <ul class="text-danger list-unstyled mt-1">
      <li *validationErrors="productForm; control:'name'; let err">
        {{ err }}
      </li>
    </ul>
  </div>

  <div class="form-group">
    <label>Category</label>
    <input class="form-control" formControlName="category" />
    <ul class="text-danger list-unstyled mt-1">
      <li *validationErrors="productForm; control:'category'; let err">
        {{ err }}
      </li>
    </ul>
  </div>

  <div class="form-group">
    <label>Price</label>
    <input class="form-control" formControlName="price" />
    <ul class="text-danger list-unstyled mt-1">
      <li *validationErrors="productForm; control:'price'; let err">
        {{ err }}
      </li>
    </ul>
  </div>

  <div class="mt-2">
    <button type="submit" class="btn btn-primary"
           [class.btn-warning]="editing"
           [disabled]="form.invalid">
      {{editing ? "Save" : "Create"}}
    </button>
    <button type="reset" class="btn btn-secondary mt-1">Cancel</button>
  </div>
</form>
```

The directive is configured with the `FormGroup` property defined by the component class and the name of the `FormControl` object for which errors are required. This is an indirect way of working, but it works as seamlessly as dealing with individual elements once the building blocks are in place, as shown in Figure 21-12.

ID	Name	Category	Price	
1	Kayak	Watersports	\$275.00	<button>Delete</button> <button>Edit</button>
2	Lifejacket	Watersports	\$48.95	<button>Delete</button> <button>Edit</button>
3	Soccer Ball	Soccer	\$19.50	<button>Delete</button> <button>Edit</button>
4	Corner Flags	Soccer	\$34.95	<button>Delete</button> <button>Edit</button>
5	Thinking Cap	Chess	\$16.00	<button>Delete</button> <button>Edit</button>

Creating New Product

Name
r2
A name must be at least 3 characters
The name contains illegal characters

Category

Price
a
The price contains illegal characters

Create New Product **Create** **Cancel**

Figure 21-12. Displaying validation messages from a form group

Nesting Form Controls

The `FormGroup` methods described Table 21-8 accept the `AbstractControl` class, which is the base class for both `FormGroup` and `FormControl` and which allows `FormGroup` objects to be nested, which can be a useful way to group related controls. To prepare, Listing 21-22 adds features to the `Product` model class.

Listing 21-22. Expanding the Model in the `product.model.ts` File in the `src/app/model` Folder

```
export class Product {
    constructor(public id?: number,
        public name?: string,
        public category?: string,
        public price?: number,
        public details?: Details) { }

    export class Details {
        constructor(public supplier?: string,
            public keywords?: string) {}

    }
}
```

The `details` property will be used to collect additional information about a product. Listing 21-23 adds values to the static data source for the new properties.

Listing 21-23. Adding Data in the static.datasource.ts File in the src/app/model Folder

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";

@Injectable()
export class StaticDataSource {
  private data: Product[];

  constructor() {
    this.data = new Array<Product>(
      new Product(1, "Kayak", "Watersports", 275,
        { supplier: "Acme", keywords: "boat, small"}),
      new Product(2, "Lifejacket", "Watersports", 48.95),
      new Product(3, "Soccer Ball", "Soccer", 19.50),
      new Product(4, "Corner Flags", "Soccer", 34.95),
      new Product(5, "Thinking Cap", "Chess", 16));
  }

  getData(): Product[] {
    return this.data;
  }
}
```

Listing 21-24 adds elements to the table template to display the new properties.

Listing 21-24. Displaying Details in the table.component.html File in the src/app/core Folder

```
<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr>
      <th>ID</th><th>Name</th><th>Category</th><th>Price</th>
      <th>Details</th><th></th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of getProducts()">
      <td>{{item.id}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price | currency:"USD" }}</td>
      <td>
        <ng-container *ngIf="item.details else empty">
          {{ item.details?.supplier }}, {{ item.details?.keywords}}
        </ng-container>
        <ng-template #empty>(None)</ng-template>
      </td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm m-1"
          (click)="deleteProduct(item.id)">
          Delete
        </button>
      </td>
    
  </tbody>
</table>
```

```

        <button class="btn btn-warning btn-sm" (click)="editProduct(item.id)">
            Edit
        </button>
    </td>
</tr>
</tbody>
</table>
<button class="btn btn-primary mt-1" (click)="createProduct()">
    Create New Product
</button>

```

Listing 21-25 adds a nested FormGroup to the form component and populates it with FormControl objects that correspond to the new model properties.

Listing 21-25. Adding a Nested Form Group in the form.component.ts File in the src/app/core Folder

```

...
productForm: FormGroup = new FormGroup({
    name: new FormControl("", {
        validators: [
            Validators.required,
            Validators.minLength(3),
            Validators.pattern("^[A-Za-z ]+$")
        ],
        updateOn: "change"
    }),
    category: new FormControl("", { validators: Validators.required }),
    price: new FormControl("", {
        validators: [Validators.required, Validators.pattern("^[0-9\\.]+$")]
    }),
    details: new FormGroup({
        supplier: new FormControl("", { validators: Validators.required }),
        keywords: new FormControl("", { validators: Validators.required })
    })
});
...

```

To complete the process, Listing 21-26 adds new elements to the component's template.

Listing 21-26. Adding Elements in the form.component.html File in the src/app/core Folder

```

<form [formGroup]="productForm" #form="ngForm"
    (ngSubmit)="submitForm()" (reset)="resetForm()">

    <!-- existing input elements omitted for brevity... -->

    <ng-container formGroupName="details">
        <div class="form-group">
            <label>Supplier</label>
            <input class="form-control" formControlName="supplier" />
        </div>
        <div class="form-group">

```

```

<label>Keywords</label>
<input class="form-control" formControlName="keywords" />
</div>
</ng-container>

<div class="mt-2">
  <button type="submit" class="btn btn-primary"
    [class.btn-warning]="editing"
    [disabled]="form.invalid">
    {{editing ? "Save" : "Create"}}
  </button>
  <button type="reset" class="btn btn-secondary m-1">Cancel</button>
</div>
</form>

```

The nested FormGroup is associated with an element using the `formGroupName` directive, which I have applied to an `ng-container` element so that I don't introduce any elements into the form. Within the `ng-container` element, input elements are associated with FormControl objects using the `formControlName` directive. Angular takes care of dealing with the nested names, and the supplier input element within the details `ng-container` element is mapped to the supplier FormControl within the details FormGroup object.

When the value of the top-level FormGroup object is read or set, Angular also maps the nested FormGroup/FormControl objects to nested properties on the data object. Click the Edit button for the Kayak product, and you will see that the Supplier and Keywords fields are populated using the nested model properties defined, as shown in Figure 21-13. Change the values and click Save, and you will see that the mapping works in both directions.

The screenshot shows a browser window titled 'ExampleApp' at 'localhost:4200'. The main content is a table titled 'Editing Kayak' with columns: ID, Name, Category, Price, Details, Delete, and Edit. The first row (ID 1) has 'Kayak' in the Name column and 'Acme, boat, small' in the Details column. The 'Edit' button for this row is highlighted with a yellow box. To the right of the table is a detailed view of the product's properties:

- Name:** Kayak
- Category:** Watersports
- Price:** 275
- Supplier:** Acme
- Keywords:** boat, small

Below the table is a 'Create New Product' button. At the bottom right are 'Save' and 'Cancel' buttons.

Figure 21-13. Using a nested form group

Validating Nested Form Controls

Nested form groups can be used to assess the status of the elements they contain, which means that the top-level FormGroup will report on all of the FormControl objects, including the nested ones, and the nested FormGroup will report on just its controls. Listing 21-27 generates messages to denote the validation status of the nested controls.

Listing 21-27. Monitoring a Nested FormGroup in the form.component.ts File in the src/app/core Folder

```
import { Component } from "@angular/core";
import { FormControl, NgForm, Validators, FormGroup } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;

  // ...form groups and controls omitted for brevity...

  ngOnInit() {
    this.productForm.get("details")?.statusChanges.subscribe(newStatus => {
      this.messageService.reportMessage(new Message(`Details ${newStatus}`));
    })
  }

  // ...constructor and methods omitted for brevity...
}
```

Click the Edit button for the Kayak product and clear the Name field. The form is invalid, but no message is generated because the controls managed by the nested FormGroup have values. Clear the Supplier field to change the status of the nested group and produce a message, as shown in Figure 21-14.

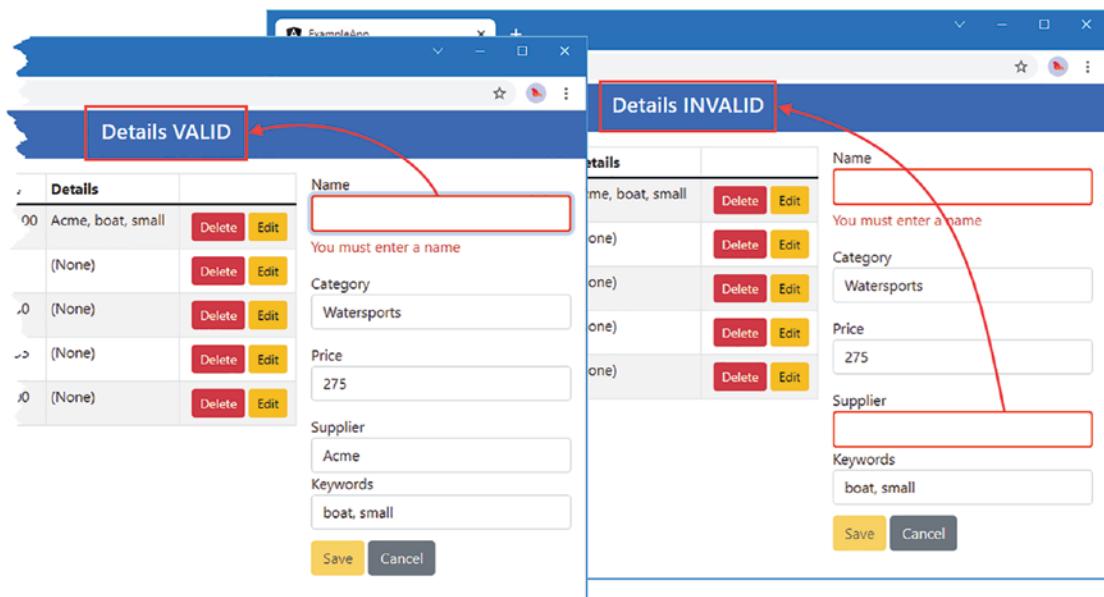


Figure 21-14. Monitoring a nested form group

Validation messages can be displayed in the template by specifying the path to the nested controls, which is done by combining the name of the nested group with the control, as shown in Listing 21-28.

Listing 21-28. Nested Validation Messages in the form.component.html File in the src/app/core Folder

```
...
<ng-container formGroupName="details">
  <div class="form-group">
    <label>Supplier</label>
    <input class="form-control" formControlName="supplier" />
    <ul class="text-danger list-unstyled mt-1">
      <li *validationErrors="productForm; control:'details.supplier';
          label: 'supplier'; let err">
          {{ err }}
      </li>
    </ul>
  </div>
  <div class="form-group">
    <label>Keywords</label>
    <input class="form-control" formControlName="keywords" />
    <ul class="text-danger list-unstyled mt-1">
      <li *validationErrors="productForm; control:'details.keywords';
          label: 'keyword'; let err">
          {{ err }}
      </li>
    </ul>
  </div>
</ng-container>
```

```
</ul>
</div>
</ng-container>
...
...
```

The path for the nested control combines the name of the form group, followed by the name of the control, separated by a period:

```
...
<li *validationErrors="productForm; control:'details.keywords'>
  label: 'keyword'; let err">
...
...
```

One issue with nested controls is that the control path can't be used in error messages, which is why I added an optional `label` property in the directive in Listing 21-19. Click the Edit button for the Kayak product and clear the Supplier and Keywords fields to see the error messages, as shown in Figure 21-15.

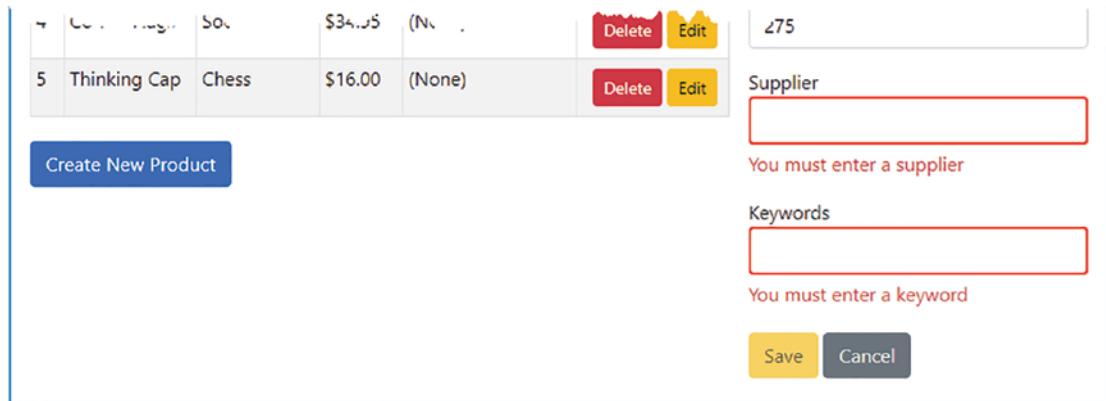


Figure 21-15. Displaying validation messages for nested controls

Summary

In this chapter, I introduced the Angular reactive forms API and showed you how it can be used to create and manage forms, providing a more code-centered approach than the standard template-driven forms described in Chapter 12. I explained the use of the `FormControl` and `FormGroup` classes and demonstrated their use in managing form elements. In the next chapter, I continue to describe the forms API, explaining how to create controls dynamically and how to create custom validators.

CHAPTER 22



Using the Forms API, Part 2

In this chapter, I continue to describe the Angular forms API, explaining how to create form controls dynamically and how to create custom validation. Table 22-1 summarizes the chapter.

Table 22-1. Chapter Summary

Problem	Solution	Listing
Creating and managing form controls dynamically	Use a <code>FormArray</code> object	1–6
Validating dynamically created form controls	Use a control's position in its enclosing <code>FormArray</code> as identification during the validation process	7, 8
Altering the values produced by dynamically created controls	Override the methods defined by the <code>FormArray</code> class	9, 10
Creating custom form validation	Create a function that returns an implementation of the <code>ValidatorFn</code> interface, which performs validation on a control's value	11–13
Applying a custom validator in a template-driven form	Create a directive that calls the validator function	14–17
Validating multiple related fields	Perform validation on a <code>FormGroup</code> or <code>FormArray</code>	18–23
Performing complex or remote validation	Create an asynchronous validator	24–27

Preparing for This Chapter

For this chapter, I will continue using the `exampleApp` project from Chapter 21. No changes are required for this chapter.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

To start the development server, open a command prompt, navigate to the exampleApp folder, and run the following command:

```
ng serve
```

Open a new browser window and navigate to `http://localhost:4200` to see the content shown in Figure 22-1.

ID	Name	Category	Price	Details	
1	Kayak	Watersports	\$275.00	Acme, boat, small	<button>Delete</button> <button>Edit</button>
2	Lifejacket	Watersports	\$48.95	(None)	<button>Delete</button> <button>Edit</button>
3	Soccer Ball	Soccer	\$19.50	(None)	<button>Delete</button> <button>Edit</button>
4	Corner Flags	Soccer	\$34.95	(None)	<button>Delete</button> <button>Edit</button>
5	Thinking Cap	Chess	\$16.00	(None)	<button>Delete</button> <button>Edit</button>

Creating New Product

Name

Category

Price

Supplier

Keywords

Create **Cancel**

Create New Product

Figure 22-1. Running the example application

Creating Form Components Dynamically

The `FormGroup` class is useful when the structure and number of elements in the form are known in advance. For applications that need to dynamically add and remove elements, Angular provides the `FormArray` class. Both `FormArray` and `FormControl` are derived from the `AbstractControl` class and provide the same features for managing `FormGroup` objects; the difference is that the `FormArray` class allows `FormControl` objects to be created without specifying names and stores its controls as an array, making it easier to add and remove controls. To prepare, Listing 22-1 changes a model property to an array so that it can be used to store multiple values.

Listing 22-1. Changing a Property Type in the product.model.ts File in the src/app/model Folder

```
export class Product {

    constructor(public id?: number,
        public name?: string,
        public category?: string,
        public price?: number,
        public details?: Details) { }

}

export class Details {
    constructor(public supplier?: string,
        public keywords?: string[] ) {}
}
```

Listing 22-2 updates the static example data to reflect the change to the keywords property.

Listing 22-2. Updating the Example Data in the static.datasource.ts File in the src/app/model Folder

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";

@Injectable()
export class StaticDataSource {
    private data: Product[];

    constructor() {
        this.data = new Array<Product>(
            new Product(1, "Kayak", "Watersports", 275,
                { supplier: "Acme", keywords: ["boat", "small"]}),
            new Product(2, "Lifejacket", "Watersports", 48.95,
                { supplier: "Smoot Co", keywords: ["safety"]}),
            new Product(3, "Soccer Ball", "Soccer", 19.50),
            new Product(4, "Corner Flags", "Soccer", 34.95),
            new Product(5, "Thinking Cap", "Chess", 16));
    }

    getData(): Product[] {
        return this.data;
    }
}
```

Using a Form Array

The `FormArray` class stores its child controls in an array and provides the properties and methods described in Table 22-2 for managing the array, in addition to those it inherits from the `AbstractControl` class and that are shared with the `FormGroup` and `FormControl` classes.

Table 22-2. Useful FormArray Members for Managing Controls

Name	Description
controls	This property returns an array containing the child controls.
length	This property returns the number of controls that are in the FormArray.
at(index)	This property returns the control at the specified index in the FormArray.
push(control)	This method adds a control to the end of the array.
insert(index, control)	This method inserts a control at the specified index.
setControl(index, control)	This method replaces the control at the specified index.
removeAt(index)	This method removes the control at the specified index.
clear()	This method removes all of the controls from the FormArray.

The FormArray class also provides methods for setting the values of the controls it manages using arrays, rather than name-value maps, as described in Table 22-3.

Table 22-3. The FormArray Methods for Setting Values

Name	Description
setValue(values)	This method accepts an array of values and uses them to set the values of the child controls based on the order in which they are defined. The number of elements in the values array must match the number of controls in the FormArray.
patchValue(values)	This method accepts an array of values and uses them to set the values of the child controls based on the order in which they are defined. Unlike the setValue method, the number of elements in the values array does not have to match the number of controls in the FormArray, and this method will ignore values for which there are no controls and will ignore controls for which there are no values.
reset(values)	This method resets the controls in the FormArray and sets their values using the optional array argument. The number of elements in the values array does not have to match the number of controls in the FormArray. Values for which there are no controls are ignored, and controls for which there are no values are reset to their default state.

Listing 22-3 uses the features described in these tables to vary the number of controls displayed for the keywords model property.

Listing 22-3. Using a FormArray in the form.components.ts File in the src/app/core Folder

```
import { Component } from "@angular/core";
import { FormControl, NgForm, Validators, FormGroup, FormArray }
  from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";
```

```

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;

  keywordGroup = new FormArray([
    this.createKeywordFormControl()
  ])

  productForm: FormGroup = new FormGroup({
    name: new FormControl("", {
      validators: [
        Validators.required,
        Validators.minLength(3),
        Validators.pattern("^[A-Za-z ]+$")
      ],
      updateOn: "change"
    }),
    category: new FormControl("", { validators: Validators.required }),
    price: new FormControl("", {
      validators: [Validators.required, Validators.pattern("^[0-9\\.]+$")]
    }),
    details: new FormGroup({
      supplier: new FormControl("", { validators: Validators.required }),
      keywords: this.keywordGroup
    })
  });
}

constructor(private model: Model, private state: SharedState,
  private messageService: MessageService) {
  this.state.changes.subscribe((upd) => this.handleStateChange(upd))
  this.messageService.reportMessage(new Message("Creating New Product"));
}

// ngOnInit() {
//   this.productForm.get("details")?.statusChanges.subscribe(newStatus => {
//     this.messageService.reportMessage(new Message(`Details ${newStatus}`));
//   })
// }

handleStateChange(newState: StateUpdate) {
  this.editing = newState.mode == MODES.EDIT;
  this.keywordGroup.clear();
  if (this.editing && newState.id) {
    Object.assign(this.product, this.model.getProduct(newState.id)
      ?? new Product());
    this.messageService.reportMessage(
      new Message(`Editing ${this.product.name}`));
  }
}

```

```

    this.product.details?.keywords?.forEach(val => {
      this.keywordGroup.push(this.createKeywordFormControl());
    })
  } else {
    this.product = new Product();
    this.messageService.reportMessage(new Message("Creating New Product"));
  }
  if (this.keywordGroup.length == 0) {
    this.keywordGroup.push(this.createKeywordFormControl());
  }
  this.productForm.reset(this.product);
}

submitForm() {
  if (this.productForm.valid) {
    Object.assign(this.product, this.productForm.value);
    this.model.saveProduct(this.product);
    this.product = new Product();
    this.keywordGroup.clear();
    this.keywordGroup.push(this.createKeywordFormControl());
    this.productForm.reset();
  }
}

resetForm() {
  this.keywordGroup.clear();
  this.keywordGroup.push(this.createKeywordFormControl());
  this.editing = true;
  this.product = new Product();
  this.productForm.reset();
}

createKeywordFormControl(): FormControl {
  return new FormControl();
}
}
}

```

I have defined a `FormArray` property so that I can access it in the template, which is important because there are no built-in directives that export the `FormArray` for use as a template variable:

```

...
keywordGroup = new FormArray([
  this.createKeywordFormControl()
])
...

```

The `FormArray` is initialized with the initial set of controls it will manage, expressed as an array. Notice that controls are not given names, and the features described in Table 22-2 and Table 22-3 all work on arrays. Consistency is important when creating controls, so I define the `createKeywordFormControl` method, which creates the `FormControl` objects:

```
...
createKeywordFormControl(): FormControl {
    return new FormControl();
}
...
...
```

Using a method to create the `FormControl` objects ensures that I can easily alter the control configuration without having to figure out all of the places where controls are created.

Note Angular includes the `FormBuilder` class, which can be used to simplify creating and configuring `FormArray` objects and the controls it contains. I don't find this class useful, which is why I don't describe it in this chapter, but you may feel differently. See <https://angular.io/api/forms/FormBuilder> for details.

The `FormArray` is added to the overall structure of controls in the same way as a nested `FormGroup`:

```
...
details: new FormGroup({
    supplier: new FormControl("", { validators: Validators.required }),
    keywords: this.keywordGroup
})
...
...
```

Within the component, I can manage the array of controls in the `FormArray` to match the selected `Product` object, ensuring that there is at least one control in the array so the user can add values to `Product` objects that don't currently have any keyword values.

Listing 22-4 uses the `FormArray` property to create the HTML elements to match the number of `FormControl` objects.

Listing 22-4. Creating Elements in the `form.component.html` File in the `src/app/core` Folder

```
<form [formGroup]="productForm" #form="ngForm"
      (ngSubmit)="submitForm()" (reset)="resetForm()">

    <div class="form-group">
        <label>Name</label>
        <input class="form-control" formControlName="name" />
        <ul class="text-danger list-unstyled mt-1">
            <li *validationErrors="productForm; control:'name'; let err">
                {{ err }}
            </li>
        </ul>
    </div>

    <div class="form-group">
        <label>Category</label>
        <input class="form-control" formControlName="category" />
        <ul class="text-danger list-unstyled mt-1">
            <li *validationErrors="productForm; control:'category'; let err">
```

```

        {{ err }}
    </li>
</ul>
</div>

<div class="form-group">
    <label>Price</label>
    <input class="form-control" formControlName="price" />
    <ul class="text-danger list-unstyled mt-1">
        <li *validationErrors="productForm; control:'price'; let err">
            {{ err }}
        </li>
    </ul>
</div>

<ng-container formGroupName="details">
    <div class="form-group">
        <label>Supplier</label>
        <input class="form-control" formControlName="supplier" />
        <ul class="text-danger list-unstyled mt-1">
            <li *validationErrors="productForm; control:'details.supplier';
                label: 'supplier'; let err">
                {{ err }}
            </li>
        </ul>
    </div>

    <ng-container formGroupName="keywords">
        <div class="form-group" *ngFor="let c of keywordGroup.controls;
            let i = index">
            <label>Keyword {{ i + 1 }}</label>
            <input class="form-control" [formControlName]="i" [value]="c.value" />
        </div>
    </ng-container>
</ng-container>

<div class="mt-2">
    <button type="submit" class="btn btn-primary"
        [class.btn-warning]="editing"
        [disabled]="form.invalid">
        {{editing ? "Save" : "Create"}}
    </button>
    <button type="reset" class="btn btn-secondary m-1">Cancel</button>
</div>
</form>

```

It is important to reflect the structure of the FormGroup and FormArray objects when creating HTML elements, ensuring that each is correctly configured with the formGroupName directive. I used the ng-container element to avoid introducing an HTML element for the FormArray object and used the ngFor directive to create elements for each FormControl in the FormArray:

```
...
<div class="form-group" *ngFor="let c of keywordGroup.controls; let i = index">
...

```

Each input element must be configured with the `formControlName` directive, using an array position as its value, instead of a name:

```
...
<input class="form-control" [formControlName]="i" [value]="c.value" />
...
```

The result is that the number of form controls displayed to the user varies based on the Product value that is selected, as shown in Figure 22-2. Notice that Angular correctly populates the `input` elements through the `FormArray`, mapping the values in the `keywords` model array to the elements in the form.

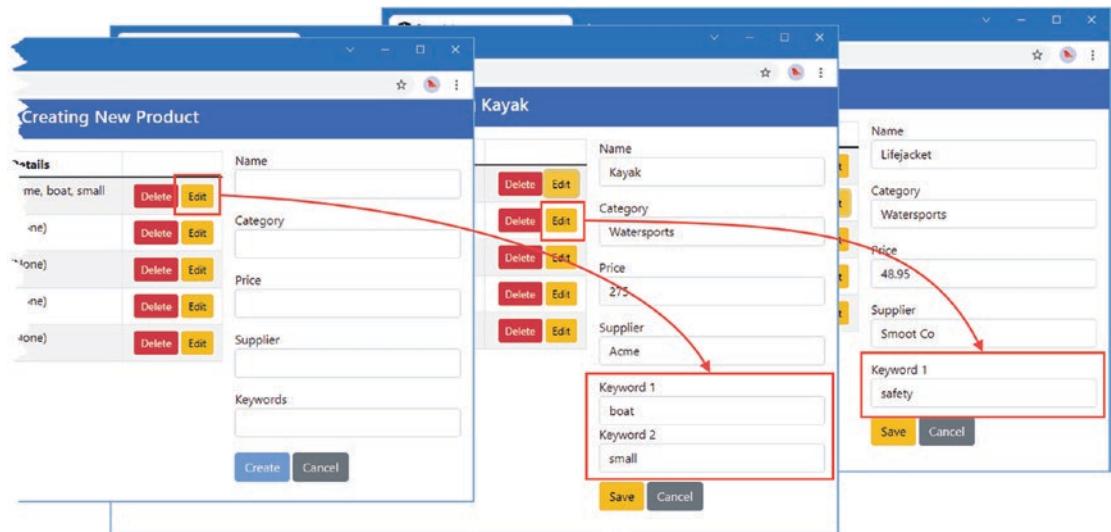


Figure 22-2. Using a form array

Adding and Removing Form Controls

To complete the support for multiple keywords, I am going to allow the user to add and remove controls. Listing 22-5 adds methods to the control class.

Listing 22-5. Adding Methods in the `form.component.ts` File in the `src/app/core` Folder

```
import { Component } from "@angular/core";
import { FormControl, NgForm, Validators, FormGroup, FormArray }
    from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";
```

```

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;

  // ...formgroup, constructor, and methods omitted for brevity...

  createKeywordFormControl(): FormControl {
    return new FormControl();
  }

  addKeywordControl() {
    this.keywordGroup.push(this.createKeywordFormControl());
  }

  removeKeywordControl(index: number) {
    this.keywordGroup.removeAt(index);
  }
}

```

The new methods use the `FormArray` features described in Table 22-2 to add and remove `FormGroup` objects. Listing 22-6 adds elements to the template that will invoke the new component methods and allow the user to manage the number of keywords fields.

Listing 22-6. Adding Elements in the `form.component.html` File in the `src/app/core` Folder

```

...
<ng-container formGroupName="keywords">
  <button class="btn btn-sm btn-primary my-2"
    (click)="addKeywordControl()" type="button">
    Add Keyword
  </button>
  <div class="form-group" *ngFor="let c of keywordGroup.controls;
    let i = index; let count = count">
    <label>Keyword {{ i + 1 }}</label>
    <div class="input-group">
      <input class="form-control" [formControlName]="i" [value]="c.value" />
      <button class="btn btn-danger" type="button" *ngIf="count > 1"
        (click)="removeKeywordControl(i)">
        Delete
      </button>
    </div>
  </div>
</ng-container>
...

```

I use the count variable exported by the `ngForm` directive to display a Delete button only when there are multiple controls in the form array. The number of keyword fields will be initially determined by the selected Product object, after which the user can add and remove fields, as shown in Figure 22-3.

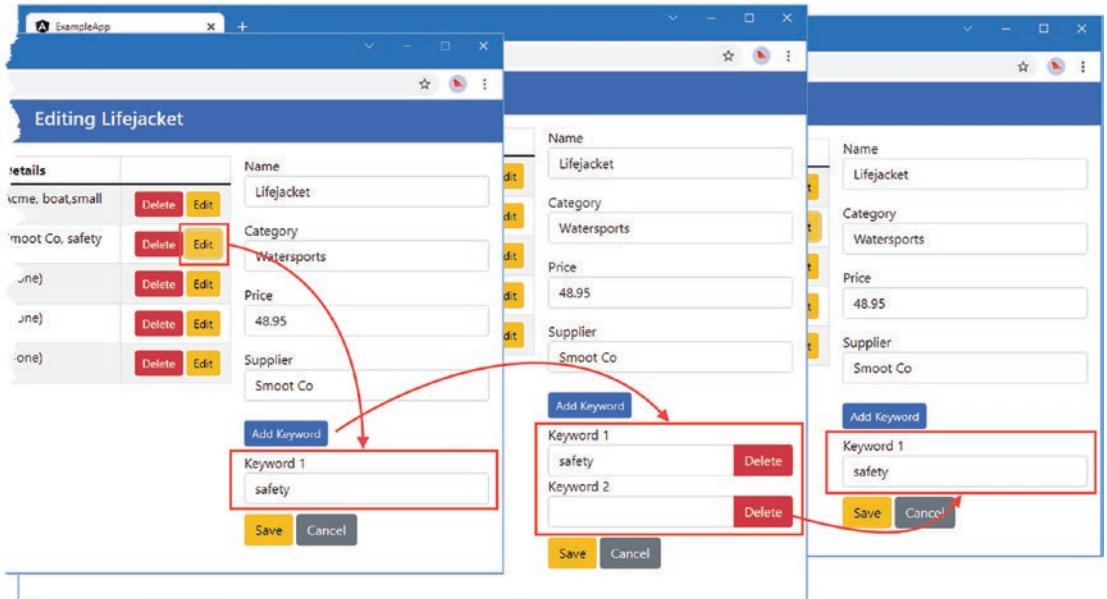


Figure 22-3. Adding and removing form controls in a form array

Validating Dynamically Created Form Controls

Validation for the controls in a `FormArray` is similar to validating the controls in a `FormGroup`, as shown in Listing 22-7.

Listing 22-7. Adding Validation in the `form.component.ts` File in the `src/app/core` Folder

```
...
createKeywordFormControl(): FormControl {
  return new FormControl("", { validators: Validators.pattern("^[A-Za-z ]+$") });
}
...
```

The advantage of using a method to create `FormControl` objects for the `FormArray` is that I can define the validation policy in a single place. Listing 22-8 displays validation messages to the user.

Listing 22-8. Displaying Validation Messages in the `form.component.html` File in the `src/app/core` Folder

```
...
<ng-container formGroupName="keywords">
  <button class="btn btn-sm btn-primary my-2" (click)="addKeywordControl()"
    type="button">
    Add Keyword
  </button>
```

```

<div class="form-group" *ngFor="let c of keywordGroup.controls;
    let i = index; let count = count">
    <label>Keyword {{ i + 1 }}</label>
    <div class="input-group">
        <input class="form-control" [FormControlName]="i" [value]="c.value" />
        <button class="btn btn-danger" type="button" *ngIf="count > 1"
            (click)="removeKeywordControl(i)">
            Delete
        </button>
    </div>
    <ul class="text-danger list-unstyled mt-1">
        <li *validationErrors="productForm; control:'details.keywords.' + i;
            label: 'keyword'; let err">
            {{ err }}
        </li>
    </ul>
</div>
</ng-container>
...

```

The path to the control uses the position in the array, rather than a name, like this:

```

...
<li *validationErrors="productForm; control:'details.keywords.' + i;
    label: 'keyword'; let err">
...

```

It is important to ensure you specify the correct position; otherwise, you will display validation messages for a different control. The user is presented with a validation error if a disallowed character is entered into a keyword field, as shown in Figure 22-4.

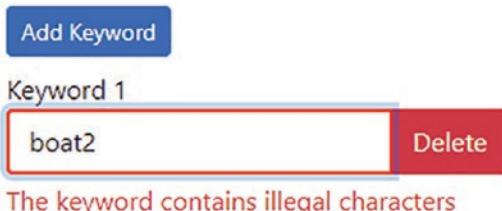


Figure 22-4. Validation for a form array control

Filtering the FormArray Values

When dealing with variable numbers of controls in a FormArray, the user may create controls and then not use them, which can cause a problem when processing the contents of the form. Figure 22-5 illustrates the problem.

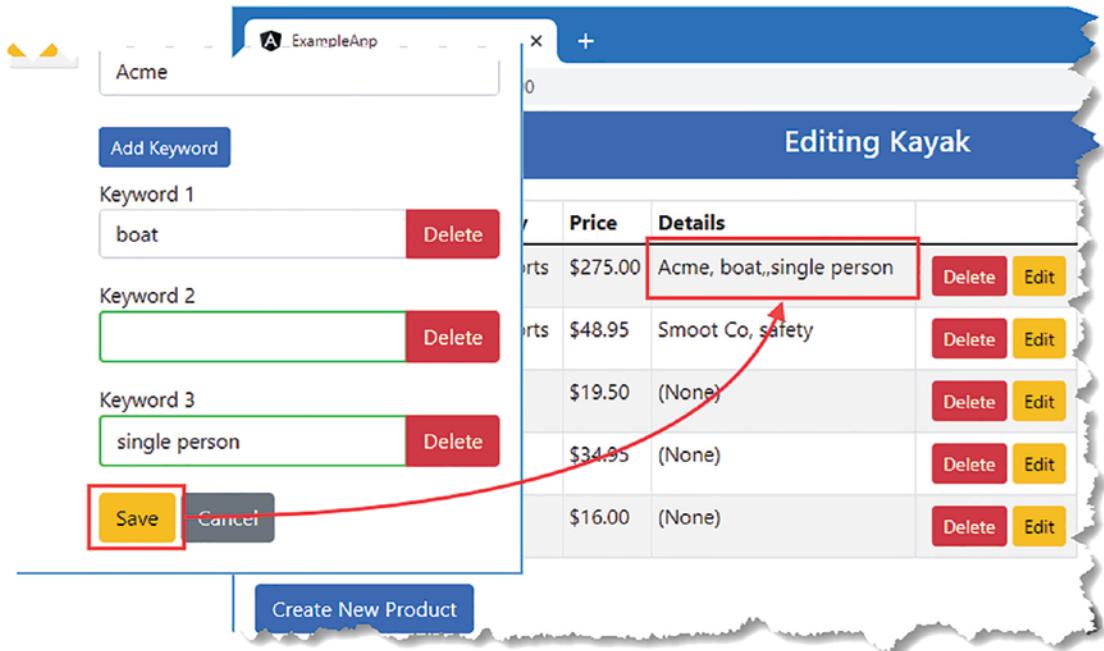


Figure 22-5. The effect of empty fields in form array controls

I left one of the keyword fields empty when I submitted the form, which means that an empty string has been included in the array of values assigned to the Product model object's keywords field.

I could prevent this problem using the required validator, but this requires the user to remove any empty controls before submitting the form, which would be an awkward interruption to their workflow.

My preference is to give the user some flexibility and create a custom class that will filter out unwanted values. Add a file named `filteredFormArray.ts` to the `src/app/core` folder with the contents shown in Listing 22-9.

Listing 22-9. The Contents of the `filteredFormArray.ts` File in the `src/app/core` Folder

```
import { FormArray } from "@angular/forms";

export type ValueFilter = (value: any) => boolean;

export class FilteredFormArray extends FormArray {
    filter: ValueFilter | undefined = (val) => val === "" || val === null;

    _updateValue() {
        (this as {value: any}).value =
            this.controls.filter((control) =>
                (control.enabled || this.disabled) && !this.filter?(control.value)
            ).map((control) => control.value);
    }
}
```

The `FilteredFormArray` class defines an `_updateValue` method, which applies a filter function that, by default, excludes empty string and null values.

The code in Listing 22-9 is on the edge of what I would consider acceptable meddling with the Angular API. You won't see the `_updateValue` method in the API documentation for the `FormArray` class because it is part of the internal API defined, which was originally defined as an abstract method in the `AbstractControl` class and then overridden in the `FormArray` class. These methods are marked as internal, and I located them by looking at the Angular source code to figure out how these classes set the `value` property.

There are two issues with using methods like this. The first is that internal methods are subject to change or removal without notice, which means that figure releases of Angular may remove the `_updateValue` method and break the code in Listing 22-9.

The second issue is that the Angular packages are compiled using a TypeScript setting that excludes internal methods from the type declaration files that are used during project development. This means that the TypeScript compiler doesn't know that the `FormArray` class defines an `_updateValue` method and won't allow the use of the `override` or the `super` keywords. For this reason, I have had to copy the original code from the `FormArray` class and integrate support for filtering, rather than just calling the `FormArray` implementation of the method and filtering the result.

I am comfortable with these issues when it comes to small changes in functionality. You must make your own assessment of the issues and decide whether relying on internal features is reasonable for your projects.

But, even if you are not comfortable using this approach in your projects, this example does let me illustrate that, once again, there is nothing magical about the way that Angular works. In this case, I relied on the fact that Angular applications are compiled into pure JavaScript and that the JavaScript rules for locating a method apply, even if TypeScript has been configured to exclude the method from the type declaration files.

Listing 22-10 replaces the standard `FormArray` object with one that filters values.

Listing 22-10. Using a Customized Form Array in the `form.component.ts` File in the `src/app/core` Folder

```
import { Component } from "@angular/core";
import { FormControl, NgForm, Validators, FormGroup, FormArray }
  from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";
import { FilteredFormArray } from "./filteredFormArray";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;

  keywordGroup = new FilteredFormArray([
    this.createKeywordFormControl()
  ])
}
```

```

productForm: FormGroup = new FormGroup({
  name: new FormControl("", {
    validators: [
      Validators.required,
      Validators.minLength(3),
      Validators.pattern("^[A-Za-z ]+$")
    ],
    updateOn: "change"
  }),
  category: new FormControl("", { validators: Validators.required }),
  price: new FormControl("", {
    validators: [Validators.required, Validators.pattern("^[0-9\\.]+$")]
  }),
  details: new FormGroup({
    supplier: new FormControl("", { validators: Validators.required }),
    keywords: this.keywordGroup
  })
});

// ...constructor and methods omitted for brevity...
}

```

The use of the filter prevents empty values from being included in the keywords array, as shown in Figure 22-6.

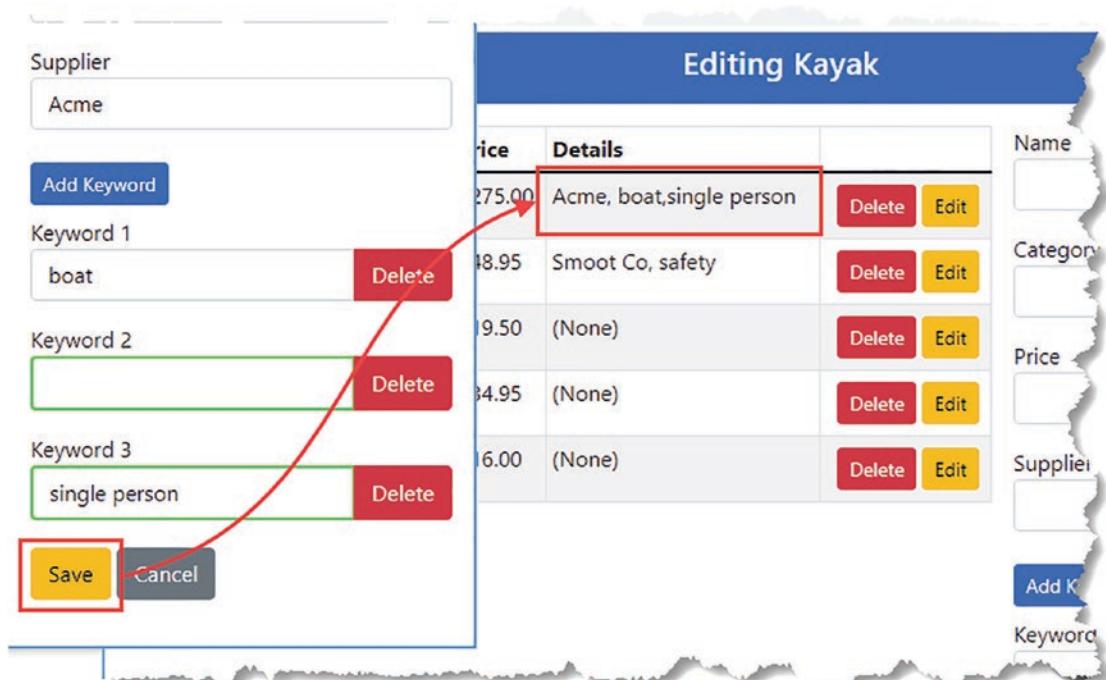


Figure 22-6. Filtering values

Creating Custom Form Validation

Angular supports custom form validators, which can be used to enforce a validation policy that is specific to the application, rather than the general-purpose validation that the built-in validators provide. A good way to understand how custom validation works is to re-create the functionality provided by one of the built-in validators.

Create the `src/app/validation` folder and add to it a file named `limit.ts` with the contents shown in Listing 22-11.

Listing 22-11. The Contents of the limit.ts File in the src/app/validation Folder

```
import { AbstractControl, ValidationErrors, ValidatorFn } from "@angular/forms";

export class LimitValidator {

    static Limit(limit:number) : ValidatorFn {
        return (control: AbstractControl) : ValidationErrors | null => {
            let val = parseFloat(control.value);
            if (isNaN(val) || val > limit) {
                return {"limit": {"limit": limit, "actualValue": val}};
            }
            return null;
        }
    }
}
```

Custom validators are functions that implement the `ValidatorFn` interface, which describes a factory function that creates functions that perform validation. The factory function accepts parameters that allow validation to be configured and returns functions that accept an `AbstractControl` parameter and return a `ValidationErrors | null` result:

```
...
static Limit(limit:number) : ValidatorFn {
    return (control: AbstractControl) : ValidationErrors | null => {
...
}
```

The factory function in this example is named `Limit`, and it defines a parameter named `limit` that specifies a maximum acceptable value, similar to the way that the built-in `min` validator works.

Listing 22-12 adds support for translating the validation results produced by the custom validator into messages that can be displayed to the user.

Listing 22-12. Adding Support for a New Validator in the validation_helper.ts File in the src/app/core Folder

```
import { Pipe } from "@angular/core";
import { FormControl, ValidationErrors } from "@angular/forms";

@Pipe({
    name: "validationFormat"
})
export class ValidationHelper {
```

```

transform(source: any, name: any) : string[] {
    if (source instanceof FormControl) {
        return this.formatMessages((source as FormControl).errors, name)
    }
    return this.formatMessages(source as ValidationErrors, name)
}

formatMessages(errors: ValidationErrors | null, name: string): string[] {
    let messages: string[] = [];
    for (let errorName in errors) {
        switch (errorName) {
            case "required":
                messages.push(`You must enter a ${name}`);
                break;
            case "minlength":
                messages.push(`A ${name} must be at least
                    ${errors['minlength'].requiredLength}
                    characters`);
                break;
            case "pattern":
                messages.push(`The ${name} contains
                    illegal characters`);
                break;
            case "limit":
                messages.push(`The ${name} must be less than
                    ${errors['limit'].limit}`);
                break;
        }
    }
    return messages;
}
}

```

Custom validators are applied in the same way as those built into Angular, as shown in Listing 22-13.

Listing 22-13. Using a Custom Validator in the form.component.ts File in the src/app/core Folder

```

import { Component } from "@angular/core";
import { FormControl, NgForm, Validators, FormGroup, FormArray }
    from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";
import { FilteredFormArray } from "./filteredFormArray";
import { LimitValidator } from "../validation/limit";

@Component({
    selector: "paForm",
    templateUrl: "form.component.html",

```

```

        styleUrls: ["form.component.css"]
    })
export class FormComponent {
    product: Product = new Product();
    editing: boolean = false;

    keywordGroup = new FilteredFormArray([
        this.createKeywordFormControl()
    ])

    productForm: FormGroup = new FormGroup({
        name: new FormControl("", {
            validators: [
                Validators.required,
                Validators.minLength(3),
                Validators.pattern("^[A-Za-z ]+$")
            ],
            updateOn: "change"
        }),
        category: new FormControl("", { validators: Validators.required }),
        price: new FormControl("", {
            validators: [
                Validators.required, Validators.pattern("^[0-9\\.]+$"),
                LimitValidator.limit(300)
            ]
        }),
        details: new FormGroup({
            supplier: new FormControl("", { validators: Validators.required }),
            keywords: this.keywordGroup
        })
    });
}

// ...constructor and methods omitted for brevity...
}

```

To see the effect of the custom validator, enter a value in the Price field that exceeds the limit set in Listing 22-13, as shown in Figure 22-7.

The screenshot shows a simple form with a single input field labeled "Price". The input field contains the value "500". A red rectangular border highlights the input field, indicating it is invalid. Below the input field, a red message in a sans-serif font states "The price must be less than 300".

Figure 22-7. Using a custom validator

Creating a Directive for a Custom Validator

A directive is required to apply a custom validator to a template element when the reactive forms API isn't used. Add a file named `hilow.ts` to the `src/app/validation` folder with the content shown in Listing 22-14.

Listing 22-14. The Contents of the hilow.ts File in the src/app/validation Folder

```

import { Directive, Input, SimpleChanges } from "@angular/core";
import { AbstractControl, NG_VALIDATORS, ValidationErrors,
    Validator, ValidatorFn } from "@angular/forms";

export class HiLowValidator {

    static HiLow(high:number, low: number) : ValidatorFn {
        return (control: AbstractControl) : ValidationErrors | null => {
            let val = parseFloat(control.value);
            if (isNaN(val) || val > high || val < low) {
                return {"hilow": {"high": high, "low": low, "actualValue": val}};
            }
            return null;
        }
    }
}

@Directive({
    selector: 'input[high][low]',
    providers: [{provide: NG_VALIDATORS, useExisting: HiLowValidatorDirective,
        multi: true}]
})
export class HiLowValidatorDirective implements Validator {

    @Input()
    high: number | string | undefined

    @Input()
    low: number | string | undefined

    validator?: (control: AbstractControl) => ValidationErrors | null;

    ngOnChanges(changes: SimpleChanges): void {
        if ("high" in changes || "low" in changes) {
            let hival = typeof(this.high) == "string"
                ? parseInt(this.high) : this.high;
            let loval = typeof(this.low) == "string"
                ? parseInt(this.low) : this.low;
            this.validator = HiLowValidator.HiLow(hival ?? Number.MAX_VALUE,
                loval ?? 0);
        }
    }

    validate(control: AbstractControl): ValidationErrors | null {
        return this.validator?(control) ?? null;
    }
}

```

The `HiLowValidator` class defines a `HiLow` factory function that can be used with reactive forms. This listing also defines the `HiLowValidatorDirective` class, which implements the `Validator` interface and the `validate` method it defines. The `ngOnChanges` method is used to create a new validator when the input value changes, which is used in the `validate` method to assess the contents of a control.

Validation directives must use the `providers` property to register the `NG_VALIDATORS` service, like this:

```
...
providers: [{provide: NG_VALIDATORS,
    useExisting: HiLowValidatorDirective, multi: true}]
...
...
```

Without this property, Angular will not use the directive for validation. Listing 22-15 registers the validation directive so that it can be used in templates.

Listing 22-15. Registering a Directive in the core.module.ts File in the src/app/core Folder

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "./table.component";
import { FormComponent } from "./form.component";
import { SharedState } from "./sharedState.service";
import { ValidationHelper } from "./validation_helper";
import { ValidationErrorsDirective } from "./validationErrors.directive";
import { HiLowValidatorDirective } from "../validation/hilow";

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule, ReactiveFormsModule],
  declarations: [TableComponent, FormComponent, ValidationHelper,
    ValidationErrorsDirective, HiLowValidatorDirective],
  exports: [ModelModule, TableComponent, FormComponent],
  providers: [SharedState]
})
export class CoreModule { }
```

Listing 22-16 adds support for producing a validation message that can be displayed to the user.

Listing 22-16. Adding a Validation Message in the validation_helper.ts File in the src/app/core Folder

```
import { Pipe } from "@angular/core";
import { FormControl, ValidationErrors } from "@angular/forms";

@Pipe({
  name: "validationFormat"
})
export class ValidationHelper {

  transform(source: any, name: any) : string[] {
    if (source instanceof FormControl) {
      return this.formatMessages((source as FormControl).errors, name)
    }
  }
}
```

```

        }
        return this.formatMessages(source as ValidationErrors, name)
    }

formatMessages(errors: ValidationErrors | null, name: string): string[] {
    let messages: string[] = [];
    for (let errorName in errors) {
        switch (errorName) {
            case "required":
                messages.push(`You must enter a ${name}`);
                break;
            case "minlength":
                messages.push(`A ${name} must be at least
                    ${errors['minlength'].requiredLength}
                    characters`);
                break;
            case "pattern":
                messages.push(`The ${name} contains
                    illegal characters`);
                break;
            case "limit":
                messages.push(`The ${name} must be less than
                    ${errors['limit'].limit}`);
                break;
            case "hilow":
                messages.push(`The ${name} must be between
                    ${errors['hilow'].low} and ${errors['hilow'].high}`);
                break;
        }
    }
    return messages;
}
}

```

Finally, Listing 22-17 applies the validation directive in a template.

Listing 22-17. Applying a Directive in the form.component.html File in the src/app/core Folder

```

...
<div class="form-group">
    <label>Price</label>
    <input class="form-control" formControlName="price" [high]=300 [low]=10 />
    <ul class="text-danger list-unstyled mt-1">
        <li *validationErrors="productForm; control:'price'; let err">
            {{ err }}
        </li>
    </ul>
</div>
...

```

As you enter values into the Price field, the `HiLowValidatorDirective` uses the `HiLow` factory function for validation, as shown in Figure 22-8.

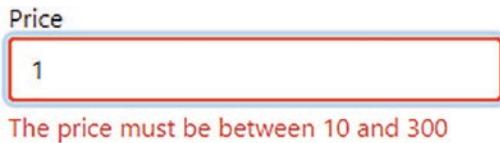


Figure 22-8. Applying a custom validator with a directive

Validating Across Multiple Fields

Custom validators can be used to enforce policies that apply to multiple fields. This type of validator is applied to the `FormGroup` or `FormArray` that is the parent to the controls that are validated. Add a file named `unique.ts` to the `src/app/validation` folder with the contents shown in Listing 22-18.

Listing 22-18. The Contents of the `unique.ts` File in the `src/app/validation` Folder

```
import { AbstractControl, FormArray, ValidationErrors, ValidatorFn }  
from "@angular/forms";  
  
export class UniqueValidator {  
  
    static unique() : ValidatorFn {  
        return (control: AbstractControl) : ValidationErrors | null => {  
            if (control instanceof FormArray) {  
                let badElems = control.controls.filter((child, index) => {  
                    return control.controls.filter((c, i2) => i2 != index)  
                        .some(target => target.value != ""  
                            && target.value == child.value);  
                });  
                if (badElems.length > 0) {  
                    return {"unique": {}};  
                }  
            }  
            return null;  
        }  
    }  
}
```

The validator function looks for duplicate values in the array of controls managed by the `FormArray` and produces an error if there are duplicates. Listing 22-19 applies the validator to the `FormArray` in the component class.

Listing 22-19. Applying a Validator in the `form.component.ts` File in the `src/app/core` Folder

```
import { Component } from "@angular/core";  
import { FormControl, NgForm, Validators, FormGroup, FormArray }  
from "@angular/forms";  
import { Product } from "../model/product.model";
```

```

import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";
import { FilteredFormArray } from "./filteredFormArray";
import { LimitValidator } from "../validation/limit";
import { UniqueValidator } from "../validation/unique";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;

  keywordGroup = new FilteredFormArray([
    this.createKeywordFormControl(),
  ], {
    validators: UniqueValidator.unique()
  })

  // ...form structure, constructor and methods omitted for brevity...
}

```

Listing 22-20 adds a validation message that can be presented to the user.

Listing 22-20. Adding a Validation Message in the validation_helper.ts File in the src/app/core Folder

```

import { Pipe } from "@angular/core";
import { FormControl, ValidationErrors } from "@angular/forms";

@Pipe({
  name: "validationFormat"
})
export class ValidationHelper {

  transform(source: any, name: any) : string[] {
    if (source instanceof FormControl) {
      return this.formatMessages((source as FormControl).errors, name)
    }
    return this.formatMessages(source as ValidationErrors, name)
  }

  formatMessages(errors: ValidationErrors | null, name: string): string[] {
    let messages: string[] = [];
    for (let errorName in errors) {
      switch (errorName) {
        case "required":
          messages.push(`You must enter a ${name}`);
          break;
      }
    }
  }
}

```

```
        case "minlength":
            messages.push(`A ${name} must be at least
${errors['minlength'].requiredLength}
characters`);
            break;
        case "pattern":
            messages.push(`The ${name} contains
illegal characters`);
            break;
        case "limit":
            messages.push(`The ${name} must be less than
${errors['limit'].limit}`);
            break;
        case "hilow":
            messages.push(`The ${name} must be between
${errors['hilow'].low} and ${errors['hilow'].high}`);
            break;
        case "unique":
            messages.push(`The ${name} must be unique`);
            break;
    }
}

return messages;
}
```

The final step is to display validation messages for the form array in the template, as shown in Listing 22-21.

Listing 22-21. Displaying Validation Messages in the form.component.html File in the src/app/core Folder

```
...
<ng-container formGroupName="keywords">
  <button class="btn btn-sm btn-primary my-2" (click)="addKeywordControl()"
    type="button">
    Add Keyword
  </button>
  <ul class="text-danger list-unstyled mt-1">
    <li *validationErrors="productForm; control:'details.keywords';"
        label: 'keywords' let err>
      {{ err }}
    </li>
  </ul>
  <div class="form-group" *ngFor="let c of keywordGroup.controls;
    let i = index; let count = count">
    <label>Keyword {{ i + 1 }}</label>
    <div class="input-group">
      <input class="form-control" [formControlName]="i" [value]="c.value" />
      <button class="btn btn-danger" type="button" *ngIf="count > 1"
        (click)="removeKeywordControl(i)">
        Delete
      </button>
    </div>
  </div>
</ng-container>
```

```

        </div>
        <ul class="text-danger list-unstyled mt-1">
            <li *validationErrors="productForm; control:'details.keywords.' + i;
                label: 'keyword'; let err">
                {{ err }}
            </li>
        </ul>
    </div>
</ng-container>
...

```

To see the effect, click the Edit button for the Kayak product and change the value of the second keyword field to boat. The validator will detect the duplicate value and display a validation message, as shown in Figure 22-9.

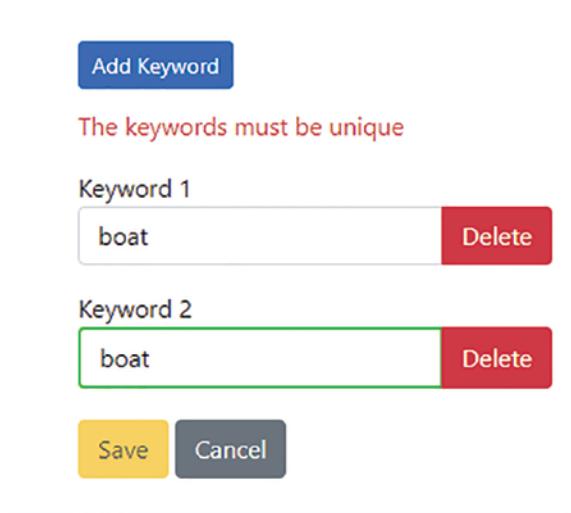


Figure 22-9. Validating across fields

The validator works as expected, but there is an important mismatch between the validation message and the color coding applied to the individual elements, which I will improve upon in the next section.

Improving Cross-Field Validation

Improving the cross-field validation experience can be done, but it requires careful navigation around the way that Angular expects groups of form controls to behave. Unlike an earlier example in this chapter, no internal methods are used, but the code relies on the `setTimeout` method to trigger changes after the current update cycle to perform updates without creating an infinite update loop.

The problem is that Angular expects changes to propagate up through the structure of form controls so that the user edits a field, which triggers validation in the `FormControl`, and then in its enclosing `FormGroup` or `FormArray`, working its way to the top-level `FormGroup`. To achieve the effect I want, I have to push updates in the opposite direction so that a change in validation status in the `FormArray` triggers validation updates in the enclosed `FormControl` objects. Listing 22-22 updates the `unique` custom validator so that it alters the validation status of contained `FormControl` elements that contain the same value.

Listing 22-22. Improving Validation in the unique.ts File in the src/app/validation Folder

```

import { AbstractControl, FormArray, ValidationErrors, ValidatorFn }  
from "@angular/forms";  
  
export class UniqueValidator {  
  
    static uniquechild(control: AbstractControl) : ValidationErrors | null {  
        return control.parent?.hasError("unique") ? {"unique-child": {}} : null;  
    }  
  
    static unique() : ValidatorFn {  
        return (control: AbstractControl) : ValidationErrors | null => {  
            let badElems: AbstractControl[] = [];  
            let goodElems: AbstractControl[] = [];  
            if (control instanceof FormArray) {  
                control.controls.forEach((child, index) => {  
                    if (control.controls.filter((c, i2) => i2 != index)  
                        .some(target => target.value != ""  
                            && target.value == child.value)) {  
                        badElems.push(child);  
                    } else {  
                        goodElems.push(child);  
                    }  
                })  
                setTimeout(() => {  
                    badElems.forEach(c => {  
                        if (!c.hasValidator(this.uniquechild)) {  
                            c.markAsDirty();  
                            c.addValidators(this.uniquechild)  
                            c.updateValueAndValidity({onlySelf: true,  
                                emitEvent: false});  
                        }  
                    })  
                    goodElems.forEach(c => {  
                        if (c.hasValidator(this.uniquechild)) {  
                            c.removeValidators(this.uniquechild);  
                        }  
                        c.updateValueAndValidity({ onlySelf: true,  
                            emitEvent: false})  
                    })  
                }, 0);  
            }  
            return badElems.length > 0 ? {"unique": {}} : null;  
        }  
    }  
}

```

The approach I have chosen is to add a validator to child controls that have duplicate values, which will ensure they are marked in red. The additional code in Listing 22-22 takes care of adding and removing the validator and triggering validation updates when there are changes, which I do through the `updateValueAndValidity` method. This method, which I have described in Table 22-4 for quick reference, updates a control's `value` property and performs validation. This method is defined by the `AbstractControl` class, which means that it can be used on `FormControl`, `FormGroup`, and `FormArray` objects.

Table 22-4. The `AbstractControl` Method for Manual Updates

Name	Description
<code>updateValueAndValidity(opts)</code>	This method causes a control to update its value and perform validation. The optional argument can be used to restrict propagating the change up the form hierarchy by setting the <code>onlySelf</code> property to <code>true</code> and preventing events by setting the <code>emitEvent</code> property to <code>false</code> .

To ensure that the validation status is maintained correctly when the user adds and removes keyword fields, Listing 22-23 overrides the `push` and `removeAt` methods defined by the `FormArray` class.

Listing 22-23. Refreshing Controls in the `filteredFormArray.ts` File in the `src/app/core` Folder

```
import { AbstractControl, FormArray } from "@angular/forms";

export type ValueFilter = (value: any) => boolean;

export class FilteredFormArray extends FormArray {

    filter: ValueFilter | undefined = (val) => val == "" || val == null;

    _updateValue() {
        (this as {value: any}).value =
            this.controls.filter((control) =>
                (control.enabled || this.disabled) && !this.filter?(control.value)
            ).map((control) => control.value);
    }

    override push(control: AbstractControl,
        options?: { emitEvent?: boolean | undefined; }): void {
        super.push(control, options);
        this.controls.forEach(c => c.updateValueAndValidity());
    }

    override removeAt(index: number,
        options?: { emitEvent?: boolean | undefined; }): void {
        super.removeAt(index, options);
        this.controls.forEach(c => c.updateValueAndValidity());
    }
}
```

The changes in Listing 22-22 and Listing 22-23 ensure that individual controls with the form array are marked as invalid when they contain duplicate values, as shown in Figure 22-10.

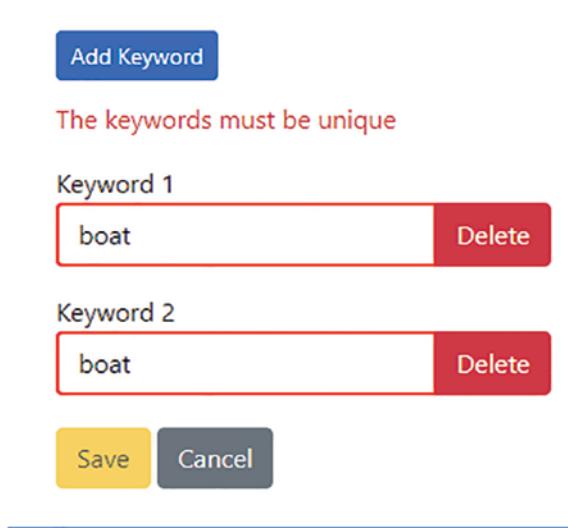


Figure 22-10. Improving the cross-field validation experience

Performing Validation Asynchronously

Asynchronous validation is useful for complex validation tasks or where the amount of time taken to perform validation is subject to delay, such as when a call to a remote HTTP service is required.

Add a file named `prohibited.ts` to the `src/app/validation` folder with the contents shown in Listing 22-24.

Listing 22-24. The Contents of the `prohibited.ts` File in the `src/app/validation` Folder

```
import { AbstractControl, AsyncValidatorFn, ValidationErrors } from "@angular/forms";
import { Observable, Subject } from "rxjs";

export class ProhibitedValidator {

    static prohibitedTerms: string[] = ["ski", "swim"]

    static prohibited(): AsyncValidatorFn {
        return (control: AbstractControl): Promise<ValidationErrors | null>
            | Observable<ValidationErrors | null> => {
            let subject = new Subject<ValidationErrors | null>();
            setTimeout(() => {
                let match = false;
                this.prohibitedTerms.forEach(word => {
                    if ((control.value as string).toLowerCase().indexOf(word) > -1) {
                        subject.next({ "prohibited": { prohibited: word } })
                        match = true;
                    }
                });
            });
        }
    }
}
```

```
        if (!match) {
            subject.next(null);
        }
        subject.complete();
    }, 1000);
    return subject;
}
}
```

Asynchronous validators produce their results through a `Promise`, which is useful when using non-Angular packages, or an `Observable`, which is useful when using the Angular features provided for making HTTP requests. For simplicity, the validator in Listing 22-24 simulates an asynchronous operation using the `setTimeout` function and compares the value it receives with a list of prohibited terms.

When performing asynchronous validation, it is important to produce a `ValidationErrors` object or, if there are no errors, `null`, so that Angular knows that the validation process is complete. Listing 22-25 introduces a new validation message that can be displayed to the user.

Listing 22-25. Adding a Message in the validate_helper.ts File in the src/app/core Folder

```
...
switch (errorName) {
  case "required":
    messages.push(`You must enter a ${name}`);
    break;
  case "minlength":
    messages.push(`A ${name} must be at least
      ${errors['minlength'].requiredLength}
      characters`);
    break;
  case "pattern":
    messages.push(`The ${name} contains
      illegal characters`);
    break;
  case "limit":
    messages.push(`The ${name} must be less than
      ${errors['limit'].limit}`);
    break;
  case "hilow":
    messages.push(`The ${name} must be between
      ${errors['hilow'].low} and ${errors['hilow'].high}`);
    break;
  case "unique":
    messages.push(`The ${name} must be unique`);
    break;
  case "prohibited":
    messages.push(`The ${name} may not contain
      "${errors["prohibited"].prohibited}"`);
    break;
}
...

```

Listing 22-26 applies the asynchronous validator to a FormControl, which is done using the `asyncValidators` property.

Listing 22-26. Applying a Validator in the form.component.ts File in the src/app/core Folder

```
import { Component } from "@angular/core";
import { FormControl, NgForm, Validators, FormGroup, FormArray }
  from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";
import { FilteredFormArray } from "./filteredFormArray";
import { LimitValidator } from "../validation/limit";
import { UniqueValidator } from "../validation/unique";
import { ProhibitedValidator } from "../validation/prohibited";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;

  keywordGroup = new FilteredFormArray([
    this.createKeywordFormControl(),
  ], {
    validators: UniqueValidator.unique()
  })

  productForm: FormGroup = new FormGroup({
    name: new FormControl("", {
      validators: [
        Validators.required,
        Validators.minLength(3),
        Validators.pattern("^[A-Za-z ]+$")
      ],
      updateOn: "change"
    }),
    category: new FormControl("", {
      validators: Validators.required,
      asyncValidators: ProhibitedValidator.prohibited()
    }),
    price: new FormControl("", {
      validators: [
        Validators.required, Validators.pattern("^[0-9\\.]+$"),
        LimitValidator.Limit(300)
      ]
    })
  });
}
```

```

details: new FormGroup({
    supplier: new FormControl("", { validators: Validators.required }),
    keywords: this.keywordGroup
})
});

// ...constructor and methods omitted for brevity...
}

```

While waiting for an asynchronous validation result, Angular puts the `FormControl` object into the pending state, which adds the `ng-pending` class. Listing 22-27 defines a new CSS style that will be applied to pending elements.

Listing 22-27. Adding a Style in the `form.component.css` File in the `src/app/core` Folder

```

input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
input.ng-pending { border: 2px solid #ffc107 }

```

To see the asynchronous validator working, start typing into the Category field. The HTML element will be displayed with an amber border during validation, and an error will be displayed if the text you enter contains the terms *ski* or *swim*, as shown in Figure 22-11.



Figure 22-11. Using an asynchronous validator

There are two points of note when using an asynchronous validator. The first point is that asynchronous validation is performed only when no synchronous validator has returned an error, which can create a confusing sequence of validation messages unless the interaction between validators has been thought through.

The second point is that asynchronous validation is performed after every change to the form control (as long as there are no synchronous validation errors). This can result in a large number of validation operations, which may be slow or expensive to perform. If this is a concern, then you can change the update frequency using the `updateOn` option described in Chapter 21.

Summary

In this chapter, I described the Angular forms API features for creating controls dynamically using a `FormArray` and explained the different ways in which custom validation can be performed, including the use of asynchronous validators. In the next chapter, I describe the features that Angular provides for making HTTP requests.

CHAPTER 23



Making HTTP Requests

All the examples since Chapter 9 have relied on static data that has been hardwired into the application. In this chapter, I demonstrate how to use asynchronous HTTP requests, often called Ajax requests, to interact with a web service to get real data into an application. Table 23-1 puts HTTP requests in context.

Table 23-1. Putting Asynchronous HTTP Requests in Context

Question	Answer
What are they?	Asynchronous HTTP requests are HTTP requests sent by the browser on behalf of the application. The term <i>asynchronous</i> refers to the fact that the application continues to operate while the browser is waiting for the server to respond.
Why are they useful?	Asynchronous HTTP requests allow Angular applications to interact with web services so that persistent data can be loaded into the application and changes can be sent to the server and saved.
How are they used?	Requests are made using the <code>HttpClient</code> class, which is delivered as a service through dependency injection. This class provides an Angular-friendly wrapper around the browser's <code>XMLHttpRequest</code> feature.
Are there any pitfalls or limitations?	Using the Angular HTTP feature requires the use of Reactive Extensions <code>Observable</code> objects.
Are there any alternatives?	You can work directly with the browser's <code>XMLHttpRequest</code> object if you prefer, and some applications—those that don't need to deal with persistent data—can be written without making HTTP requests at all.

Table 23-2 summarizes the chapter.

Table 23-2. Chapter Summary

Problem	Solution	Listing
Sending HTTP requests in an Angular application	Use the <code>Http</code> service	1-8
Performing REST operations	Use the HTTP method and URL to specify an operation and a target for that operation	9-11
Making cross-origin requests	Use the <code>HttpClient</code> service to support CORS automatically (JSONP requests are also supported)	12-13
Including headers in a request	Set the <code>headers</code> property in the <code>Request</code> object	14-15
Responding to an HTTP error	Create an error handler class	16-19

Preparing the Example Project

This chapter uses the exampleApp project created in Chapter 22. For this chapter, I rely on a server that responds to HTTP requests with JSON data. Run the command shown in Listing 23-1 in the exampleApp folder to add the `json-server` package to the project.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 23-1. Adding a Package to the Project

```
npm install json-server@0.17.0
```

I added an entry in the `scripts` section of the `package.json` file to run the `json-server` package, as shown in Listing 23-2.

Listing 23-2. Adding a Script Entry in the `package.json` File in the exampleApp Folder

```
...
"scripts": {
  "ng": "ng",
  "start": "ng serve",
  "build": "ng build",
  "watch": "ng build --watch --configuration development",
  "test": "ng test",
  "json": "json-server --p 3500 restData.js"
},
...
```

Configuring the Model Feature Module

The `@angular/common/http` JavaScript module contains an Angular module called `HttpClientModule`, which must be imported into the application in either the root module or one of the feature modules before HTTP requests can be created. In Listing 23-3, I imported the module to the `model` module, which is the natural place in the example application because I will be using HTTP requests to populate the model with data.

Listing 23-3. Importing a Module in the `model.module.ts` File in the `src/app/model` Folder

```
import { NgModule } from "@angular/core";
import { StaticDataSource } from "./static.datasource";
import { Model } from "./repository.model";
import { HttpClientModule } from "@angular/common/http";

@NgModule({
  imports: [HttpClientModule],
  providers: [Model, StaticDataSource]
})
export class ModelModule { }
```

Creating the Data File

To provide the `json-server` package with some data, I added a file called `restData.js` to the `exampleApp` folder and added the code shown in Listing 23-4.

Listing 23-4. The Contents of the `restData.js` File in the `exampleApp` Folder

```
module.exports = function () {
  var data = {
    products: [
      { id: 1, name: "Kayak", category: "Watersports", price: 275,
        details: { supplier: "Acme", keywords: ["boat", "small"] } },
      { id: 2, name: "Lifejacket", category: "Watersports", price: 48.95,
        details: { supplier: "Smoot Co", keywords: ["safety"] } },
      { id: 3, name: "Soccer Ball", category: "Soccer", price: 19.50 },
      { id: 4, name: "Corner Flags", category: "Soccer", price: 34.95 },
      { id: 5, name: "Stadium", category: "Soccer", price: 79500 },
      { id: 6, name: "Thinking Cap", category: "Chess", price: 16 },
      { id: 7, name: "Unsteady Chair", category: "Chess", price: 29.95 },
      { id: 8, name: "Human Chess Board", category: "Chess", price: 75 },
      { id: 9, name: "Bling Bling King", category: "Chess", price: 1200 }
    ]
  }
  return data
}
```

The `json-server` package can work with JSON or JavaScript files. If you use a JSON file, then its contents will be modified to reflect change requests made by clients. I have chosen the JavaScript option, which allows data to be generated programmatically and means that restarting the process will return to the original data.

Running the Example Project

Open a new command prompt, navigate to the `exampleApp` folder, and run the following command to start the data server:

```
npm run json
```

This command will start the `json-server`, which will listen for HTTP requests on port 3500. Open a new browser window and navigate to `http://localhost:3500/products/2`. The server will respond with the following data:

```
{
  "id": 2,
  "name": "Lifejacket",
  "category": "Watersports",
  "price": 48.95,
  "details": {
    "supplier": "Smoot Co",
    "keywords": [
      "safety"
    ]
  }
}
```

Leave the `json-server` running and use a separate command prompt to start the Angular development tools by running the following command in the `exampleApp` folder:

```
ng serve
```

Use the browser to navigate to `http://localhost:4200` to see the content illustrated in Figure 23-1.

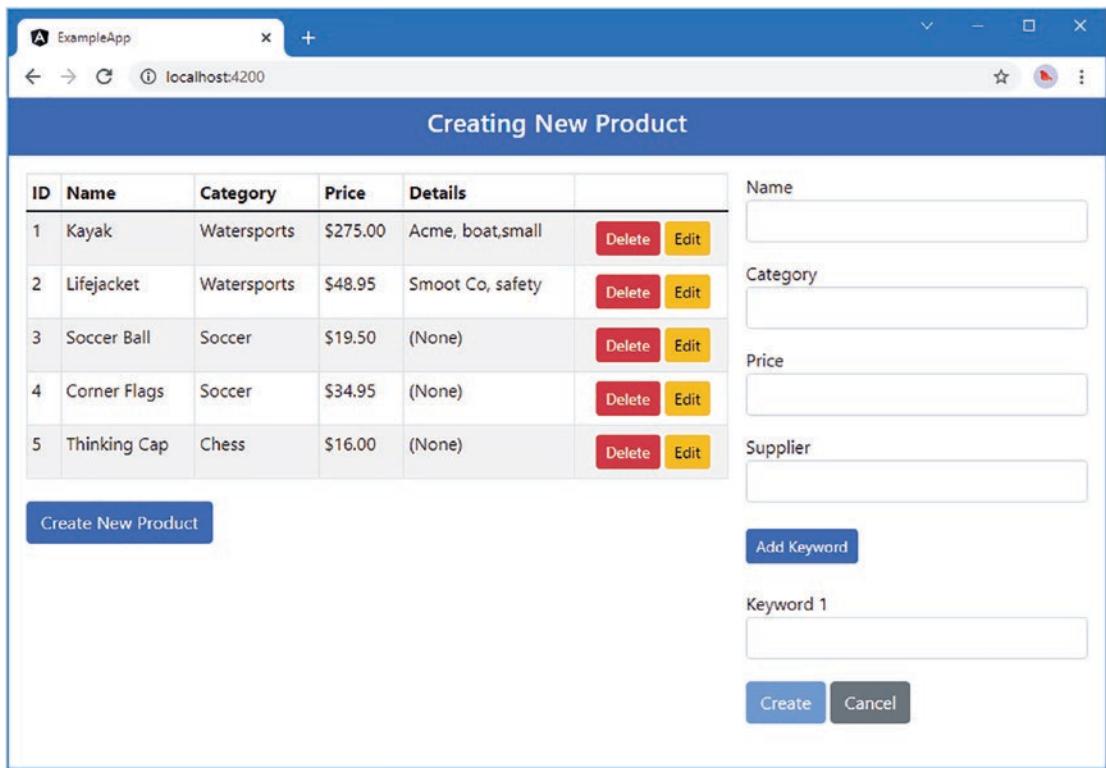


Figure 23-1. Running the example application

Understanding RESTful Web Services

The most common approach for delivering data to an application is to use the Representational State Transfer pattern, known as REST, to create a data web service. There is no detailed specification for REST, which leads to a lot of different approaches that fall under the RESTful banner. There are, however, some unifying ideas that are useful in web application development.

The core premise of a RESTful web service is to embrace the characteristics of HTTP so that request methods—also known as *verbs*—specify an operation for the server to perform, and the request URL specifies one or more data objects to which the operation will be applied.

As an example, here is a URL that might refer to a specific product in the example application:

`http://localhost:3500/products/2`

The first segment of the URL—`products`—is used to indicate the collection of objects that will be operated on and allows a single server to provide multiple services, each with separate data. The second segment—`2`—selects an individual object within the `products` collection. In the example, it is the value of the `id` property that uniquely identifies an object and that will be used in the URL, in this case, specifying the `Lifejacket` object.

The HTTP verb or method used to make the request tells the RESTful server what operation should be performed on the specified object. When you tested the RESTful server in the previous section, the browser sent an HTTP GET request, which the server interprets as an instruction to retrieve the specified object and send it to the client. It is for this reason that the browser displayed a JSON representation of the `Lifejacket` object.

Table 23-3 shows the most common combination of HTTP methods and URLs and explains what each of them does when they are sent to a RESTful server.

Table 23-3. Common HTTP Verbs and Their Effect in a RESTful Web Service

Verb	URL	Description
GET	/products	This combination retrieves all the objects in the products collection.
GET	/products/2	This combination retrieves the object whose <code>id</code> is 2 from the products collection.
POST	/products	This combination is used to add a new object to the products collection. The request body contains a JSON representation of the new object.
PUT	/products/2	This combination is used to replace the object in the products collection whose <code>id</code> is 2. The request body contains a JSON representation of the replacement object.
PATCH	/products/2	This combination is used to update a subset of the properties of the object in the products collection whose <code>id</code> is 2. The request body contains a JSON representation of the properties to update and the new values.
DELETE	/products/2	This combination is used to delete the product whose <code>id</code> is 2 from the products collection.

Caution is required because there can be considerable differences in the way that some RESTful web services work, caused by differences in the frameworks used to create them and the preferences of the development team. It is important to confirm how a web service uses verbs and what is required in the URL and request body to perform operations.

Some common variations include web services that won't accept any request bodies that contain `id` values (to ensure they are generated uniquely by the server's data store) or any web services that don't support all of the verbs (it is common to ignore PATCH requests and only accept updates using the PUT verb).

Replacing the Static Data Source

The best place to start with HTTP requests is to replace the static data source in the example application with one that retrieves data from the RESTful web service. This will provide a foundation for describing how Angular supports HTTP requests and how they can be integrated into an application.

Creating the New Data Source Service

To create a new data source, I added a file called `rest.datasource.ts` in the `src/app/model` folder and added the statements shown in Listing 23-6.

Listing 23-6. The Contents of the rest.datasource.ts File in the src/app/model Folder

```
import { Injectable, Inject, InjectionToken } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { Observable } from "rxjs";
import { Product } from "./product.model";

export const REST_URL = new InjectionToken("rest_url");

@Injectable()
export class RestDataSource {

    constructor(private http: HttpClient,
        @Inject(REST_URL) private url: string) { }

    getData(): Observable<Product[]> {
        return this.http.get<Product[]>(this.url);
    }
}
```

This is a simple-looking class, but there are some important features at work, which I described in the sections that follow.

Setting Up the HTTP Request

Angular provides the ability to make asynchronous HTTP requests through the `HttpClient` class, which is defined in the `@angular/common/http` JavaScript module and is provided as a service in the `HttpClientModule` feature module. The data source declared a dependency on the `HttpClient` class using its constructor, like this:

```
...
constructor(private http: HttpClient, @Inject(REST_URL) private url: string) { }
...
```

The other constructor argument is used so that the URL that requests are sent to doesn't have to be hardwired into the data source. I'll create a provider using the `REST_URL` opaque token when I configure the feature module. The `HttpClient` object received through the constructor is used to make an HTTP GET request in the data source's `getData` method, like this:

```
...
getData(): Observable<Product[]> {
    return this.http.get<Product[]>(this.url);
}
...
```

The `HttpClient` class defines a set of methods for making HTTP requests, each of which uses a different HTTP verb, as described in Table 23-4.

Tip The methods in Table 23-4 accept an optional configuration object, as demonstrated in the “Configuring Request Headers” section.

Table 23-4. The HttpClient Methods

Name	Description
get(url)	This method sends a GET request to the specified URL.
post(url, body)	This method sends a POST request using the specified object as the body.
put(url, body)	This method sends a PUT request using the specified object as the body.
patch(url, body)	This method sends a PATCH request using the specified object as the body.
delete(url)	This method sends a DELETE request to the specified URL.
head(url)	This method sends a HEAD request, which has the same effect as a GET request except that the server will return only the headers and not the request body.
options(url)	This method sends an OPTIONS request to the specified URL.
request(method, url, options)	This method can be used to send a request with any verb, as described in the “Consolidating HTTP Requests” section.

Processing the Response

The methods described in Table 23-4 accept a type parameter, which the HttpClient class uses to parse the response received from the server. The RESTful web server returns JSON data, which has become the de facto standard used by web services, and the HttpClient object will automatically convert the response into an Observable that yields an instance of the type parameter when it completes. This means that if you call the get method, for example, with a Product[] type parameter, then the response from the get method will be an Observable<Product[]> that represents the eventual response from the HTTP request.

```
...
getData(): Observable<Product[]> {
    return this.http.get<Product[]>(this.url);
}
...
```

Caution The methods in Table 23-4 prepare an HTTP request, but it isn’t sent to the server until the Observer object’s subscribe method is invoked. Be careful, though, because the request will be sent once per call to the subscribe method, which makes it easy to inadvertently send the same request multiple times.

Configuring the Data Source

The next step is to configure a provider for the new data source and to create a value-based provider to configure it with a URL to which requests will be sent. Listing 23-7 shows the changes to the `model.module.ts` file.

Listing 23-7. Configuring the Data Source in the `model.module.ts` File in the `src/app/model` Folder

```
import { NgModule } from "@angular/core";
import { StaticDataSource } from "./static.datasource";
import { Model } from "./repository.model";
import { HttpClientModule } from "@angular/common/http";
import { RestDataSource, REST_URL } from "./rest.datasource";

@NgModule({
  imports: [HttpClientModule],
  providers: [Model, RestDataSource,
    { provide: REST_URL, useValue: `http://${location.hostname}:3500/products` }]
})
export class ModelModule { }
```

The two new providers enable the `RestDataSource` class as a service and use the `REST_URL` opaque token to configure the URL for the web service. I removed the provider for the `StaticDataSource` class, which is no longer required.

Using the REST Data Source

The final step is to update the repository class so that it declares a dependency on the new data source and uses it to get the application data, as shown in Listing 23-8.

Listing 23-8. Using the New Data Source in the `repository.model.ts` File in the `src/app/model` Folder

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { StaticDataSource } from "./static.datasource";
import { Observable } from "rxjs";
import { RestDataSource } from "./rest.datasource";

@Injectable()
export class Model {
  private products: Product[];
  private locator = (p: Product, id?: number) => p.id == id;

  constructor(private dataSource: RestDataSource) {
    this.products = new Array<Product>();
    // this.dataSource.getData().forEach(p => this.products.push(p));
    this.dataSource.getData().subscribe(data => this.products = data);
  }
}
```

```

getProducts(): Product[] {
    return this.products;
}

getProduct(id: number): Product | undefined {
    return this.products.find(p => this.locator(p, id));
}

saveProduct(product: Product) {
    if (product.id == 0 || product.id == null) {
        product.id = this.generateID();
        this.products.push(product);
    } else {
        let index = this.products
            .findIndex(p => this.locator(p, product.id));
        this.products.splice(index, 1, product);
    }
}

deleteProduct(id: number) {
    let index = this.products.findIndex(p => this.locator(p, id));
    if (index > -1) {
        this.products.splice(index, 1);
    }
}

private generateID(): number {
    let candidate = 100;
    while (this.getProduct(candidate) != null) {
        candidate++;
    }
    return candidate;
}
}

```

The constructor dependency has changed so that the repository will receive a `RestDataSource` object when it is created. Within the constructor, the data source's `getData` method is called, and the `subscribe` method is used to receive the data objects that are returned from the server and process them.

When you save the changes, the browser will reload the application, and the new data source will be used. An asynchronous HTTP request will be sent to the RESTful web service, which will return the larger set of data objects shown in Figure 23-2.

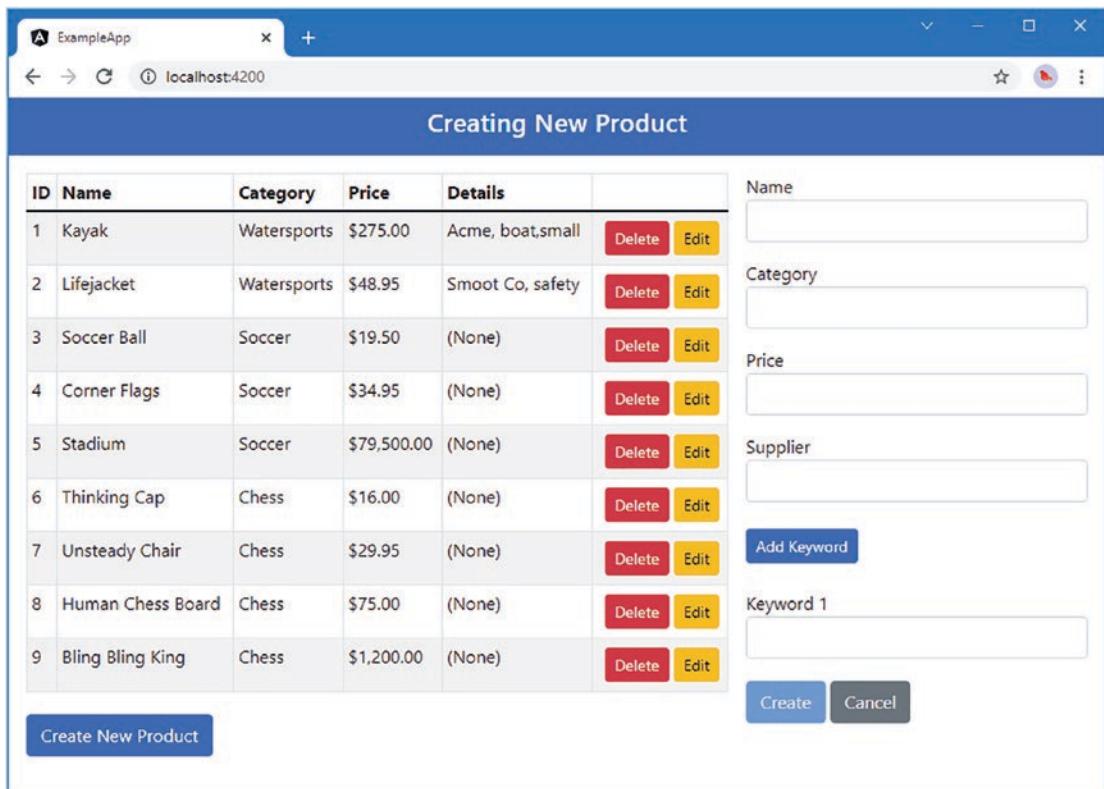


Figure 23-2. Getting the application data

Saving and Deleting Data

The data source can get data from the server, but it also needs to send data the other way, persisting changes that the user makes to objects in the model and storing new objects that are created. Listing 23-9 adds methods to the data source class to send HTTP requests to save or update objects using the Angular HttpClient class.

Listing 23-9. Sending Data in the rest.datasource.ts File in the src/app/model Folder

```
import { Injectable, Inject, InjectionToken } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { Observable } from "rxjs";
import { Product } from "./product.model";

export const REST_URL = new InjectionToken("rest_url");

@Injectable()
export class RestDataSource {

  constructor(private http: HttpClient,
    @Inject(REST_URL) private url: string) { }
```

```

    getData(): Observable<Product[]> {
        return this.http.get<Product[]>(this.url);
    }

    saveProduct(product: Product): Observable<Product> {
        return this.http.post<Product>(this.url, product);
    }

    updateProduct(product: Product): Observable<Product> {
        return this.http.put<Product>(`${this.url}/${product.id}`, product);
    }

    deleteProduct(id: number): Observable<Product> {
        return this.http.delete<Product>(`${this.url}/${id}`);
    }
}

```

The `saveProduct`, `updateProduct`, and `deleteProduct` methods follow the same pattern: they call one of the `HttpClient` class methods and return an `Observable<Product>` as the result.

When saving a new object, the ID of the object is generated by the server so that it is unique and clients don't inadvertently use the same ID for different objects. In this situation, the POST method is used, and the request is sent to the `/products` URL. When updating or deleting an existing object, the ID is already known, and a PUT request is sent to a URL that includes the ID. So, a request to update the object whose ID is 2, for example, is sent to the `/products/2` URL. Similarly, to remove that object, a DELETE request would be sent to the same URL.

What these methods have in common is that the server is the authoritative data store, and the response from the server contains the official version of the object that has been saved by the server. It is this object that is returned as the result of these methods, provided through the `Observable<Product>`.

Listing 23-10 shows the corresponding changes in the repository class that take advantage of the new data source features.

Listing 23-10. Using the Data Source Features in the repository.model.ts File in the src/app/model Folder

```

import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { StaticDataSource } from "./static.datasource";
import { Observable } from "rxjs";
import { RestDataSource } from "./rest.datasource";

@Injectable()
export class Model {
    private products: Product[];
    private locator = (p: Product, id?: number) => p.id == id;

    constructor(private dataSource: RestDataSource) {
        this.products = new Array<Product>();
        // this.dataSource.getData().forEach(p => this.products.push(p));
        this.dataSource.getData().subscribe(data => this.products = data);
    }
}

```

```

getProducts(): Product[] {
    return this.products;
}

getProduct(id: number): Product | undefined {
    return this.products.find(p => this.locator(p, id));
}

saveProduct(product: Product) {
    if (product.id == 0 || product.id == null) {
        this.dataSource.saveProduct(product)
            .subscribe(p => this.products.push(p));
    } else {
        this.dataSource.updateProduct(product).subscribe(p => {
            let index = this.products
                .findIndex(item => this.locator(item, p.id));
            this.products.splice(index, 1, p);
        });
    }
}

deleteProduct(id: number) {
    this.dataSource.deleteProduct(id).subscribe(() => {
        let index = this.products.findIndex(p => this.locator(p, id));
        if (index > -1) {
            this.products.splice(index, 1);
        }
    });
}

// private generateID(): number {
//     let candidate = 100;
//     while (this.getProduct(candidate) != null) {
//         candidate++;
//     }
//     return candidate;
// }
}

```

The changes use the data source to send updates to the server and use the results to update the locally stored data so that it is displayed by the rest of the application. To test the changes, click the Edit button for the Kayak product and change its name to Green Kayak. Click the Save button, and the browser will send an HTTP PUT request to the server, which will return a modified object that is added to the repository's products array and is displayed in the table, as shown in Figure 23-3.

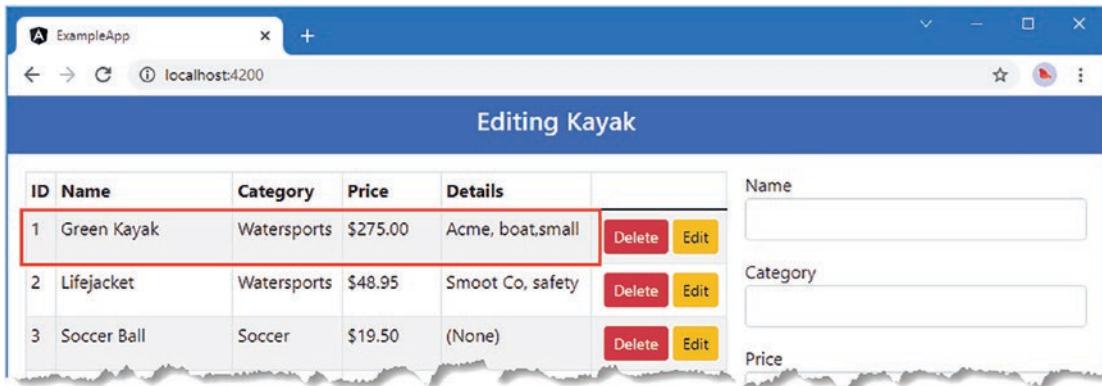


Figure 23-3. Sending a PUT request to the server

You can check that the server has stored the changes by using the browser to request `http://localhost:3500/products/1`, which will produce the following representation of the object:

```
{
  "id": 1,
  "name": "Green Kayak",
  "category": "Watersports",
  "price": 275,
  "details": {
    "supplier": "Acme",
    "keywords": [
      "boat",
      "small"
    ]
  }
}
```

Consolidating HTTP Requests

Each of the methods in the data source class duplicates the same basic pattern of sending an HTTP request using a verb-specific `HttpClient` method. This means that any change to the way that HTTP requests are made has to be repeated in four different places, ensuring that the requests that use the GET, POST, PUT, and DELETE verbs are all correctly updated and performed consistently.

The `HttpClient` class also defines the `request` method, which allows the HTTP verb to be specified as an argument. Listing 23-11 uses the `request` method to consolidate the HTTP requests in the data source class.

Listing 23-11. Consolidating HTTP Requests in the `rest.datasource.ts` File in the `src/app/model` Folder

```
import { Injectable, Inject, InjectionToken } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { Observable } from "rxjs";
import { Product } from "./product.model";
```

```

export const REST_URL = new InjectionToken("rest_url");

@Injectable()
export class RestDataSource {
  constructor(private http: HttpClient,
    @Inject(REST_URL) private url: string) { }

  getData(): Observable<Product[]> {
    return this.sendRequest<Product[]>("GET", this.url);
  }

  saveProduct(product: Product): Observable<Product> {
    return this.sendRequest<Product>("POST", this.url, product);
  }

  updateProduct(product: Product): Observable<Product> {
    return this.sendRequest<Product>("PUT",
      `${this.url}/${product.id}`, product);
  }

  deleteProduct(id: number): Observable<Product> {
    return this.sendRequest<Product>("DELETE", `${this.url}/${id}`);
  }

  private sendRequest<T>(verb: string, url: string, body?: Product)
    : Observable<T> {
    return this.http.request<T>(verb, url,
      body: body
    ));
  }
}

```

The request method accepts the HTTP verb, the URL for the request, and an optional object that is used to configure the request. The configuration object is used to set the request body using the body property, and the HttpClient will automatically take care of encoding the body object and including a serialized representation of it in the request.

Table 23-5 describes the most useful properties that can be specified to configure an HTTP request made using the request method.

Table 23-5. Useful Request Method Configuration Object Properties

Name	Description
headers	This property returns an <code>HttpHeaders</code> object that allows the request headers to be specified, as described in the “Configuring Request Headers” section.
body	This property is used to set the request body. The object assigned to this property will be serialized as JSON when the request is sent.
withCredentials	When true, this property is used to include authentication cookies when making cross-site requests. This setting must be used only with servers that include the <code>Access-Control-Allow-Credentials</code> header in responses, as part of the Cross-Origin Resource Sharing (CORS) specification. See the “Making Cross-Origin Requests” section for details.
responseType	This property is used to specify the type of response expected from the server. The default value is <code>json</code> , indicating the JSON data format.

Making Cross-Origin Requests

By default, browsers enforce a security policy that allows JavaScript code to make asynchronous HTTP requests only within the same *origin* as the document that contains them. This policy is intended to reduce the risk of cross-site scripting (CSS) attacks, where the browser is tricked into executing malicious code. The details of this attack are beyond the scope of this book, but the article available at http://en.wikipedia.org/wiki/Cross-site_scripting provides a good introduction to the topic.

For Angular developers, the same-origin policy can be a problem when using web services because they are typically outside of the origin that contains the application’s JavaScript code. Two URLs are considered to be in the same origin if they have the same protocol, host, and port and have different origins if this is not the case. The URL for the HTML file that contains the example application’s JavaScript code is `http://localhost:3000/index.html`. Table 23-6 summarizes how similar URLs have the same or different origins, compared with the application’s URL.

Table 23-6. URLs and Their Origins

URL	Origin Comparison
<code>http://localhost:3000/otherfile.html</code>	Same origin
<code>http://localhost:3000/app/main.js</code>	Same origin
<code>https://localhost:3000/index.html</code>	Different origin; protocol differs
<code>http://localhost:3500/products</code>	Different origin; port differs
<code>http://angular.io/index.html</code>	Different origin; host differs

As the table shows, the URL for the RESTful web service, `http://localhost:3500/products`, has a different origin because it uses a different port from the main application.

HTTP requests made using the Angular `HttpClient` class will automatically use Cross-Origin Resource Sharing to send requests to different origins. With CORS, the browser includes headers in the asynchronous

HTTP request that provide the server with the origin of the JavaScript code. The response from the server includes headers that tell the browser whether it is willing to accept the request. The details of CORS are outside the scope of this book, but there is a good introduction to the topic at https://en.wikipedia.org/wiki/Cross-origin_resource_sharing.

For the Angular developer, CORS is something that is taken care of automatically, just as long as the server that receives asynchronous HTTP requests supports the specification. The `json-server` package that has been providing the RESTful web service for the examples supports CORS and will accept requests from any origin, which is why the examples have been working. If you want to see CORS in action, use the browser's F12 developer tools to watch the network requests that are made when you edit or create a product. You may see a request made using the `OPTIONS` verb, known as the *preflight request*, which the browser uses to check that it is allowed to make the `POST` or `PUT` request to the server. This request and the subsequent request that sends the data to the server will contain an `Origin` header, and the response will contain one or more `Access-Control-Allow` headers, through which the server sets out what it is willing to accept from the client.

All of this happens automatically, and the only configuration option is the `withCredentials` property that was described in Table 23-5. When this property is true, the browser will include authentication cookies, and headers from the origin will be included in the request to the server.

Using JSONP Requests

CORS is available only if the server to which the HTTP requests are sent supports it. For servers that don't implement CORS, Angular also provides support for JSONP, which allows a more limited form of cross-origin requests.

JSONP works by adding a `script` element to the Document Object Model that specifies the cross-origin server in its `src` attribute. The browser sends a `GET` request to the server, which returns JavaScript code that, when executed, provides the application with the data it requires. JSONP is, essentially, a hack that works around the browser's same-origin security policy. JSONP can be used only to make `GET` requests, and it presents greater security risks than CORS. As a consequence, JSONP should be used only when CORS isn't available.

The Angular support for JSONP is defined in a feature module called `HttpClientJsonpModule`, which is defined in the `@angular/common/http` JavaScript module. To enable JSONP, Listing 23-12 adds `HttpClientJsonpModule` to the set of imports for the model module.

Listing 23-12. Enabling JSONP in the `model.module.ts` File in the `src/app/model` Folder

```
import { NgModule } from "@angular/core";
import { StaticDataSource } from "./static.datasource";
import { Model } from "./repository.model";
import { HttpClientJsonpModule, HttpClientModule } from "@angular/common/http";
import { RestDataSource, REST_URL } from "./rest.datasource";

@NgModule({
  imports: [HttpClientModule, HttpClientJsonpModule],
  providers: [Model, RestDataSource,
    { provide: REST_URL, useValue: `http://${location.hostname}:3500/products` }]
})
export class ModelModule { }
```

Angular provides support for JSONP through the `HttpClient` service, which takes care of managing the JSONP HTTP request and processing the response, which can otherwise be a tedious and error-prone process. Listing 23-13 shows the data source using JSONP to request the initial data for the application.

Listing 23-13. Making a JSONP Request in the rest.datasource.ts File in the src/app/model Folder

```

import { Injectable, Inject, InjectionToken } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { Observable } from "rxjs";
import { Product } from "./product.model";

export const REST_URL = new InjectionToken("rest_url");

@Injectable()
export class RestDataSource {
  constructor(private http: HttpClient,
    @Inject(REST_URL) private url: string) { }

  getData(): Observable<Product[]> {
    return this.http.jsonp<Product[]>(this.url, "callback");
  }

  saveProduct(product: Product): Observable<Product> {
    return this.sendRequest<Product>"POST", this.url, product);
  }

  updateProduct(product: Product): Observable<Product> {
    return this.sendRequest<Product>"PUT",
      `${this.url}/${product.id}`, product);
  }

  deleteProduct(id: number): Observable<Product> {
    return this.sendRequest<Product>"DELETE", `${this.url}/${id}`);
  }

  private sendRequest<T>(verb: string, url: string, body?: Product)
    : Observable<T> {
    return this.http.request<T>(verb, url, {
      body: body
    });
  }
}

```

JSONP can be used only for GET requests, which are sent using the `HttpClient.jsonp` method. When you call this method, you must provide the URL for the request and the name for the `callback` parameter, which must be set to `callback`, like this:

```

...
return this.http.jsonp<Product[]>(this.url, "callback");
...

```

When Angular makes the HTTP request, it creates a URL with the name of a dynamically generated function. If you look at the network requests that the browser makes, you will see that the initial request is sent to a URL like this one:

http://localhost:3500/products?callback=ng_jsonp_callback_0

The server JavaScript function matches the name used in the URL and passes it the data received from the request. JSONP is a more limited way to make cross-origin requests, and, unlike CORS, it skirts around the browser's security policy, but it can be a useful fallback in a pinch.

Configuring Request Headers

If you are using a commercial RESTful web service, you will often have to set a request header to provide an API key so that the server can associate the request with your application for access control and billing. You can set this kind of header—or any other header—by configuring the configuration object that is passed to the `request` method, as shown in Listing 23-14. (This listing also returns to using the `request` method for all requests, rather than JSONP.)

Listing 23-14. Setting a Request Header in the `rest.datasource.ts` File in the `src/app/model` Folder

```
import { Injectable, Inject, InjectionToken } from "@angular/core";
import { HttpClient, HttpHeaders } from "@angular/common/http";
import { Observable } from "rxjs";
import { Product } from "./product.model";

export const REST_URL = new InjectionToken("rest_url");

@Injectable()
export class RestDataSource {
    constructor(private http: HttpClient,
        @Inject(REST_URL) private url: string) { }

    getData(): Observable<Product[]> {
        return this.sendRequest<Product[]>("GET", this.url);
    }

    saveProduct(product: Product): Observable<Product> {
        return this.sendRequest<Product>("POST", this.url, product);
    }

    updateProduct(product: Product): Observable<Product> {
        return this.sendRequest<Product>("PUT",
            `${this.url}/${product.id}`, product);
    }

    deleteProduct(id: number): Observable<Product> {
        return this.sendRequest<Product>("DELETE", `${this.url}/${id}`);
    }

    private sendRequest<T>(verb: string, url: string, body?: Product)
        : Observable<T> {
        return this.http.request<T>(verb, url, {
            body: body,
        })
    }
}
```

```

        headers: new HttpHeaders({
            "Access-Key": "<secret>",
            "Application-Name": "exampleApp"
        })
    });
}
}

```

The `headers` property is set to an `HttpHeaders` object, which can be created using a map object of properties that correspond to header names and the values that should be used for them. If you use the browser's F12 developer tools to inspect the asynchronous HTTP requests, you will see that the two headers specified in the listing are sent to the server along with the standard headers that the browser creates, like this:

```

...
Accept:/*
Accept-Encoding:gzip, deflate, sdch, br
Accept-Language:en-US,en;q=0.8
access-key:<secret>
application-name:exampleApp
Connection:keep-alive
...

```

If you have more complex demands for request headers, then you can use the methods defined by the `HttpHeaders` class, as described in Table 23-7.

Table 23-7. The `HttpHeaders` Methods

Name	Description
<code>keys()</code>	Returns all the header names in the collection
<code>get(name)</code>	Returns the first value for the specified header
<code>getAll(name)</code>	Returns all the values for the specified header
<code>has(name)</code>	Returns <code>true</code> if the collection contains the specified header
<code>set(header, value)</code>	Returns a new <code>HttpHeaders</code> object that replaces all existing values for the specified header with a single value
<code>set(header, values)</code>	Returns a new <code>HttpHeaders</code> object that replaces all existing values for the specified header with an array of values
<code>append(name, value)</code>	Appends a value to the list of values for the specified header
<code>delete(name)</code>	Removes the specified header from the collection

HTTP headers can have multiple values, which is why there are methods that append values for headers or replace all the values in the collection. Listing 23-15 creates an empty `HttpHeaders` object and populates it with headers that have multiple values.

Listing 23-15. Setting Multiple Header Values in the rest.datasource.ts File in the src/app/model Folder

```

...
private sendRequest<T>(verb: string, url: string, body?: Product)
  : Observable<T> {

  let myHeaders = new HttpHeaders();
  myHeaders = myHeaders.set("Access-Key", "<secret>");
  myHeaders = myHeaders.set("Application-Names", ["exampleApp", "proAngular"]);

  return this.http.request<T>(verb, url, {
    body: body,
    headers: myHeaders
  });
}
...

```

When the browser sends requests to the server, they will include the following headers:

```

...
Accept:/*
Accept-Encoding:gzip, deflate, sdch, br
Accept-Language:en-US,en;q=0.8
access-key:<secret>
application-names:exampleApp,proAngular
Connection:keep-alive
...

```

Handling Errors

At the moment, there is no error handling in the application, which means that Angular doesn't know what to do if there is a problem with an HTTP request. To make it easy to generate an error, I have added a button to the product table that will lead to an HTTP request to delete an object that doesn't exist at the server, as shown in Listing 23-16.

Listing 23-16. Adding an Error Button in the table.component.html File in the src/app/core Folder

```





```

```

<td>
  <ng-container *ngIf="item.details else empty">
    {{ item.details?.supplier }}, {{ item.details?.keywords}}
  </ng-container>
  <ng-template #empty>(None)</ng-template>
</td>
<td class="text-center">
  <button class="btn btn-danger btn-sm m-1"
         (click)="deleteProduct(item.id)">
    Delete
  </button>
  <button class="btn btn-warning btn-sm" (click)="editProduct(item.id)">
    Edit
  </button>
</td>
</tr>
</tbody>
</table>
<button class="btn btn-primary m-1" (click)="createProduct()">
  Create New Product
</button>
<button class="btn btn-danger m-1" (click)="deleteProduct(-1)">
  Generate HTTP Error
</button>

```

The button element invokes the component's `deleteProduct` method with an argument of `-1`. The component will ask the repository to delete this object, which will lead to an HTTP DELETE request being sent to `/products/-1`, which does not exist. If you open the browser's JavaScript console and click the Generate HTTP Error button, you will see the response from the server displayed, like this:

```
DELETE http://localhost:3500/products/-1 404 (Not Found)
```

Improving this situation means detecting this kind of error when one occurs and notifying the user, who won't typically be looking at the JavaScript console. A real application might also respond to errors by logging them so they can be analyzed later, but I am going to keep things simple and just display an error message.

Generating User-Ready Messages

The first step in handling errors is to convert the HTTP exception into something that can be displayed to the user. The default error message, which is the one written to the JavaScript console, contains too much information to display to the user. Users don't need to know the URL that the request was sent to; just having a sense of the kind of problem that has occurred will be enough.

The best way to transform error messages is to use the `catchError` method. The `catchError` method is used with the `pipe` method to receive any errors that occur within an Observable sequence, as shown in Listing 23-17.

Listing 23-17. Transforming Errors in the rest.datasource.ts File in the src/app/model Folder

```

import { Injectable, Inject, InjectionToken } from "@angular/core";
import { HttpClient, HttpHeaders } from "@angular/common/http";
import { catchError, Observable } from "rxjs";
import { Product } from "./product.model";

export const REST_URL = new InjectionToken("rest_url");

@Injectable()
export class RestDataSource {
    constructor(private http: HttpClient,
        @Inject(REST_URL) private url: string) { }

    getData(): Observable<Product[]> {
        return this.sendRequest<Product[]>("GET", this.url);
    }

    saveProduct(product: Product): Observable<Product> {
        return this.sendRequest<Product>("POST", this.url, product);
    }

    updateProduct(product: Product): Observable<Product> {
        return this.sendRequest<Product>("PUT",
            `${this.url}/${product.id}`, product);
    }

    deleteProduct(id: number): Observable<Product> {
        return this.sendRequest<Product>("DELETE", `${this.url}/${id}`);
    }

    private sendRequest<T>(verb: string, url: string, body?: Product)
        : Observable<T> {

        let myHeaders = new HttpHeaders();
        myHeaders = myHeaders.set("Access-Key", "<secret>");
        myHeaders = myHeaders.set("Application-Names", ["exampleApp", "proAngular"]);

        return this.http.request<T>(verb, url, {
            body: body,
            headers: myHeaders
        }).pipe(catchError((error: Response) => {
            throw(`Network Error: ${error.statusText} (${error.status})`)
        }));
    }
}

```

The function passed to the `catchError` method is invoked when there is an error and receives the `Response` object that describes the outcome, which in this case is used to generate an error message that contains the HTTP status code and status text from the response.

If you save the changes and then click the Generate HTTP Error button again, the error message will still be written to the browser's JavaScript console but will have changed to the format produced by the `catchError` method.

EXCEPTION: Network Error: Not Found (404)

Handling the Errors

The errors have been transformed but not handled, which is why they are still being reported as exceptions in the browser's JavaScript console. There are two ways in which the errors can be handled. The first is to provide an error-handling function to the `subscribe` method for the `Observable` objects created by the `HttpClient` object. This is a useful way to localize the error and provide the repository with the opportunity to retry the operation or try to recover in some other way.

The second approach is to replace the built-in Angular error-handling feature, which responds to any unhandled errors in the application and, by default, writes them to the console. It is this feature that writes out the messages shown in the previous sections.

For the example application, I want to override the default error handler with one that uses the message service. I created a file called `errorHandler.ts` in the `src/app/messages` folder and used it to define the class shown in Listing 23-18.

Listing 23-18. The Contents of the `errorHandler.ts` File in the `src/app/messages` Folder

```
import { ErrorHandler, Injectable, NgZone } from "@angular/core";
import { MessageService } from "./message.service";
import { Message } from "./message.model";

@Injectable()
export class MessageErrorHandler implements ErrorHandler {

    constructor(private messageService: MessageService, private ngZone: NgZone) {}

    handleError(error: any) {
        let msg = error instanceof Error ? error.message : error.toString();
        this.ngZone.run(() => this.messageService
            .reportMessage(new Message(msg, true)), 0);
    }
}
```

The `ErrorHandler` class is defined in the `@angular/core` module and responds to errors through a `handleError` method. The class shown in the listing replaces the default implementation of this method with one that uses the `MessageService` to report an error.

Redefining the error handler presents a problem. I want to display a message to the user, which requires the Angular change detection process to be triggered. But the message is produced by a service, and Angular doesn't keep track of the state of services as it does for components and directives. To resolve this issue, I defined an `NgZone` constructor parameter and used its `run` method to create the error message:

```
...
this.ngZone.run(() => this.messageService.reportMessage(new Message(msg, true)), 0);
...
```

The run method executes the function it receives and then triggers the Angular change detection process. For this example, the result is that the new message will be displayed to the user. Without the use of the NgZone object, the error message would be created but would not be displayed to the user until the next time the Angular detection process runs, which is usually in response to user interaction.

To replace the default ErrorHandler, I used a class provider in the message module, as shown in Listing 23-19.

Listing 23-19. Configuring an Error Handler in the message.module.ts File in the src/app/messages Folder

```
import { NgModule, ErrorHandler } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { MessageComponent } from "./message.component";
import { MessageService } from "./message.service";
import { MessageErrorHandler } from "./errorHandler";

@NgModule({
  imports: [BrowserModule],
  declarations: [MessageComponent],
  exports: [MessageComponent],
  providers: [MessageService,
    { provide: ErrorHandler, useClass: MessageErrorHandler }]
})
export class MessageModule { }
```

The error handling function uses the MessageService to report an error message to the user. Once these changes have been saved, clicking the Generate HTTP Error button produces an error that the user can see, as shown in Figure 23-4.

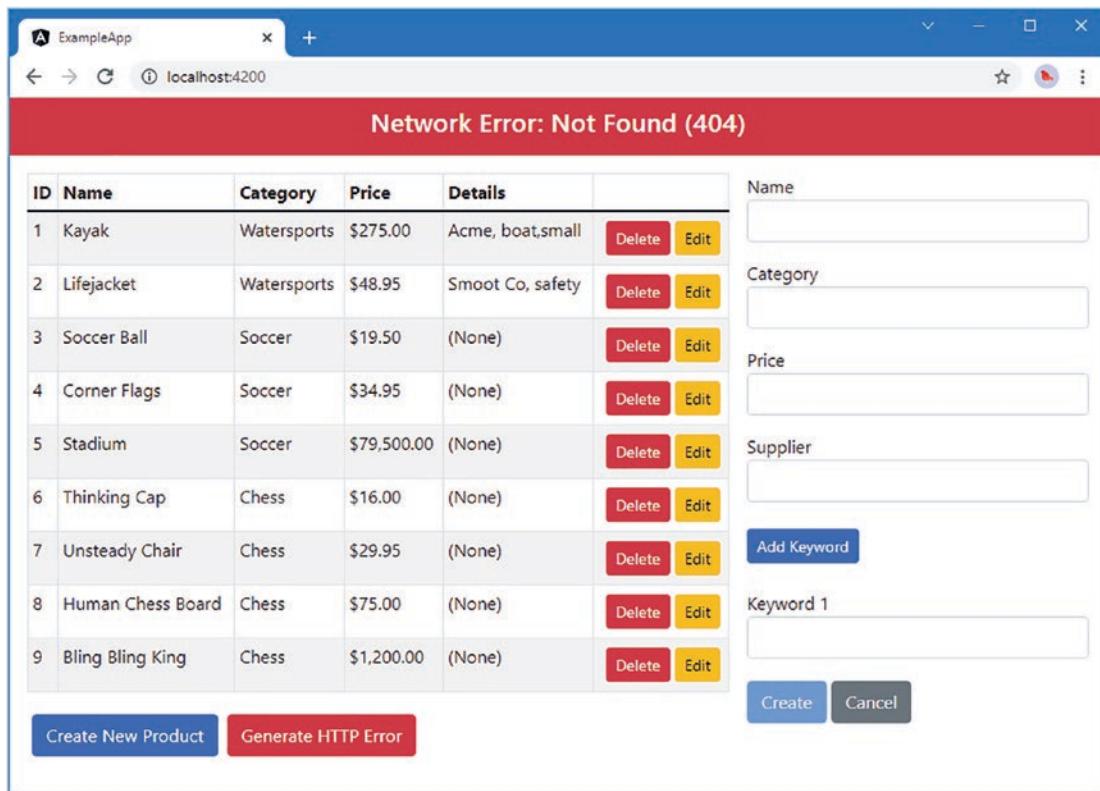


Figure 23-4. Handling an HTTP error

Summary

In this chapter, I explained how to make asynchronous HTTP requests in Angular applications. I introduced RESTful web services and the methods provided by the Angular `HttpClient` class that can be used to interact with them. I explained how the browser restricts requests to different origins and how Angular supports CORS and JSONP to make requests outside of the application's origin. In the next chapter, I introduce the URL routing feature, which allows for navigating complex applications.

CHAPTER 24



Routing and Navigation: Part 1

The Angular routing feature allows applications to change the components and templates that are displayed to the user by responding to changes to the browser's URL. This allows complex applications to be created that adapt the content they present openly and flexibly, with minimal coding. To support this feature, data bindings and services can be used to change the browser's URL, allowing the user to navigate around the application.

Routing is useful as the complexity of a project increases because it allows the structure of an application to be defined separately from the components and directives, meaning that changes to the structure can be made in the routing configuration and do not have to be applied to the individual components.

In this chapter, I demonstrate how the basic routing system works and apply it to the example application. In Chapters 25 and 26, I explain the more advanced routing features. Table 24-1 puts routing in context.

Table 24-1. Putting Routing and Navigation in Context

Question	Answer
What is it?	Routing uses the browser's URL to manage the content displayed to the user.
Why is it useful?	Routing allows the structure of an application to be kept apart from the components and templates in the application. Changes to the structure of the application are made in the routing configuration rather than in individual components and directives.
How is it used?	The routing configuration is defined as a set of fragments that are used to match the browser's URL and to select a component whose template is displayed as the content of an HTML element called router-outlet.
Are there any pitfalls or limitations?	The routing configuration can become unmanageable, especially if the URL schema is being defined gradually on an ad hoc basis.
Are there any alternatives?	You don't have to use the routing feature. You could achieve similar results by creating a component whose view selects the content to display to the user with the ngIf or ngSwitch directive, although this approach becomes more difficult than using routing as the size and complexity of an application increases.

Table 24-2 summarizes the chapter.

Table 24-2. Chapter Summary

Problem	Solution	Listing
Using URL navigation to select the content shown to users	Use URL routing	1-4
Navigating using an HTML element	Apply the <code>routerLink</code> attribute	5-7
Responding to route changes	Use the routing services to receive notifications	8
Including information in URLs	Use route parameters	9-17
Navigating using code	Use the <code>Router</code> service	18
Receiving notifications of routing activity	Handle the routing events	19-22

Preparing the Example Project

This chapter uses the `exampleApp` project created in Chapter 23. No changes are required for this chapter.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Open a new command prompt, navigate to the `exampleApp` folder, and run the following command to start the server that provides the RESTful web server:

```
npm run json
```

Open a separate command prompt, navigate to the `exampleApp` folder, and run the following command to start the Angular development tools:

```
ng serve
```

Open a new browser window and navigate to `http://localhost:4200` to see the content shown in Figure 24-1.

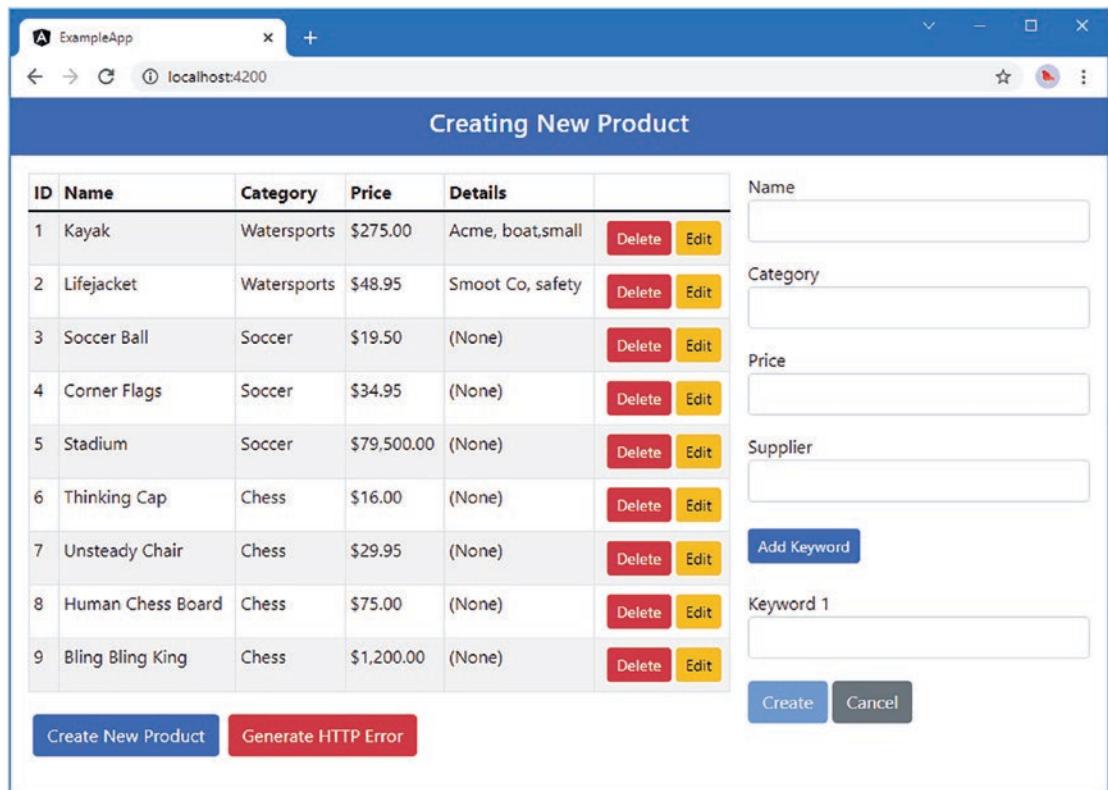


Figure 24-1. Running the example application

Getting Started with Routing

At the moment, all the content in the application is visible to the user all of the time. For the example application, this means that both the table and the form are always visible, and it is up to the user to keep track of which part of the application they are using for the task at hand.

That's fine for a simple application, but it becomes unmanageable in a complex project, which can have many areas of functionality that would be overwhelming if they were all displayed at once.

URL routing adds structure to an application using a natural and well-understood aspect of web applications: the URL. In this section, I am going to introduce URL routing by applying it to the example application so that either the table or the form is visible, with the active component being chosen based on the user's actions. This will provide a good basis for explaining how routing works and set the foundation for more advanced features.

Creating a Routing Configuration

The first step when applying routing is to define the *routes*, which are mappings between URLs and the components that will be displayed to the user. Routing configurations are conventionally defined in a file called `app.routing.ts`, defined in the `src/app` folder. I created this file and added the statements shown in Listing 24-1.

Listing 24-1. The Contents of the `app.routing.ts` File in the `src/app` Folder

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";

const routes: Routes = [
  { path: "form/edit", component: FormComponent },
  { path: "form/create", component: FormComponent },
  { path: "", component: TableComponent }]

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

The `Routes` class defines a collection of routes, each of which tells Angular how to handle a specific URL. This example uses the most basic properties, where the `path` specifies the URL and the `component` property specifies the component that will be displayed to the user.

The `path` property is specified relative to the rest of the application, which means that the configuration in Listing 24-1 sets up the routes shown in Table 24-3.

Table 24-3. The Routes Created in the Example

URL	Displayed Component
<code>http://localhost:4200/form/edit</code>	<code>FormComponent</code>
<code>http://localhost:4200/form/create</code>	<code>FormComponent</code>
<code>http://localhost:4200/</code>	<code>TableComponent</code>

The routes are packaged into a module using the `RouterModule.forRoot` method. The `forRoot` method produces a module that includes the routing service. There is also a `forChild` method that doesn't include the service and that is demonstrated in Chapter 25, where I explain how to create routes for feature modules.

Although the `path` and `component` properties are the most commonly used when defining routes, there is a range of additional properties that can be used to define routes with advanced features. These properties are described in Table 24-4, along with details of where they are described.

Table 24-4. The Routes Properties Used to Define Routes

Name	Description
path	This property specifies the path for the route.
component	This property specifies the component that will be selected when the active URL matches the path.
pathMatch	This property tells Angular how to match the current URL to the path property. There are two allowed values: full, which requires the path value to completely match the URL, and prefix, which allows the path value to match the URL, even if the URL contains additional segments that are not part of the path value. This property is required when using the redirectTo property, as demonstrated in Chapter 25.
redirectTo	This property is used to create a route that redirects the browser to a different URL when activated. See Chapter 25 for details.
children	This property is used to specify child routes, which display additional components in nested router-outlet elements contained in the template of the active component, as demonstrated in Chapter 25.
outlet	This property is used to support multiple outlet elements, as described in Chapter 26.
resolve	This property is used to define work that must be completed before a route can be activated, as described in Chapter 26.
canActivate	This property is used to control when a route can be activated, as described in Chapter 26.
canActivateChild	This property is used to control when a child route can be activated, as described in Chapter 26.
canDeactivate	This property is used to control when a route can be deactivated so that a new route can be activated, as described in Chapter 26.
loadChildren	This property is used to configure a module that is loaded only when it is needed, as described in Chapter 26.
canLoad	This property is used to control when an on-demand module can be loaded.

UNDERSTANDING ROUTE ORDERING

The order in which routes are defined is significant. Angular compares the URL to which the browser has navigated with the path property of each route in turn until it finds a match. This means that the most specific routes should be defined first, with the routes that follow decreasing in specificity. This isn't a big deal for the routes in Listing 24-1, but it becomes significant when using route parameters (described in the "Using Route Parameters" section of this chapter) or adding child routes (described in Chapter 25).

If you find that your routing configuration doesn't result in the behavior you expect, then the order in which the routes have been defined is the first thing to check.

Creating the Routing Component

When using routing, the root component is dedicated to managing the navigation between different parts of the application. This is the typical purpose of the `app.component.ts` file that was added to the project by the `ng new` command when it was created. This component is a vehicle for its template, which is the `app.component.html` file in the `src/app` folder. In Listing 24-2, I have replaced the default contents.

Listing 24-2. Replacing the Contents of the `app.component.html` File in the `src/app` File

```
<paMessages></paMessages>
<router-outlet></router-outlet>
```

The `paMessages` element displays any messages and errors in the application. For the purposes of routing, it is the `router-outlet` element—known as the *outlet*—that is important because it tells Angular that this is where the component matched by the routing configuration should be displayed.

Updating the Root Module

The next step is to update the root module so that the new root component is used to bootstrap the application, as shown in Listing 24-3, which also imports the module that contains the routing configuration.

Listing 24-3. Enabling Routing in the `app.module.ts` File in the `src/app` Folder

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { ModelModule } from './model/model.module';
import { CoreModule } from './core/core.module';
import { TableComponent } from './core/table.component';
import { FormComponent } from './core/form.component';
import { MessageModule } from './messages/message.module';
import { MessageComponent } from './messages/message.component';
import { AppComponent } from './app.component';
import { routing } from './app.routing';

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

Completing the Configuration

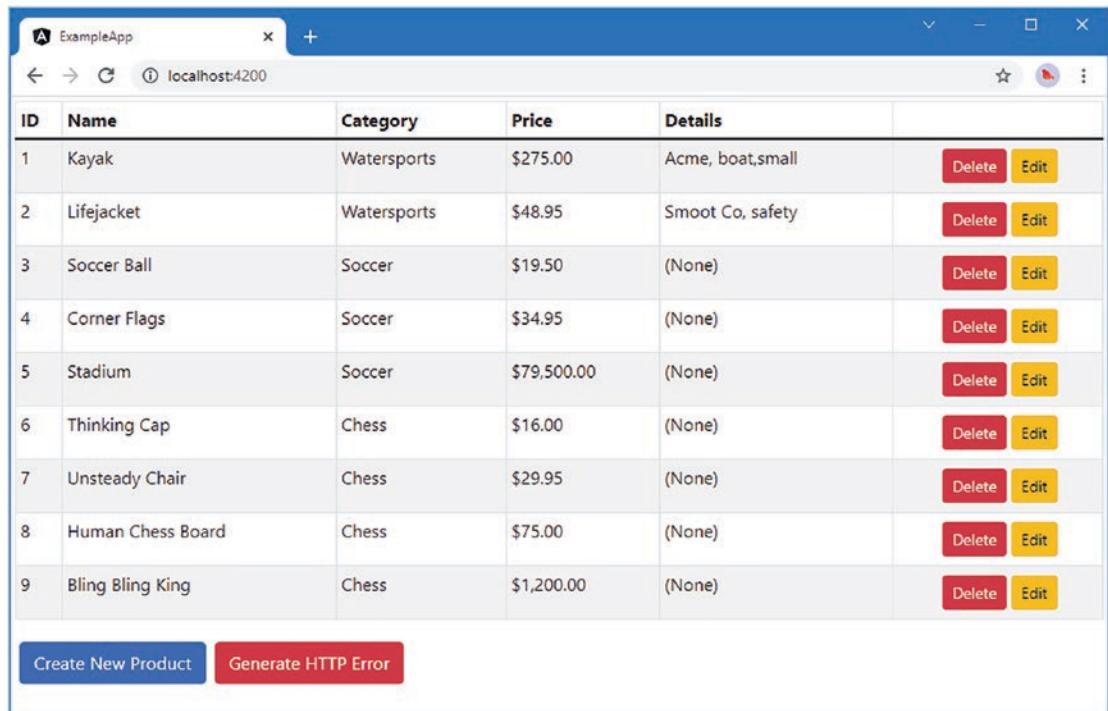
The final step is to update the `index.html` file, as shown in Listing 24-4.

Listing 24-4. Configuring Routing in the index.html File in the src Folder

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>ExampleApp</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body class="m-1">
  <app-root></app-root>
</body>
</html>
```

The app element applies the new root component, whose template contains the router-outlet element. When you save the changes and the browser reloads the application, you will see just the product table, as illustrated by Figure 24-2. The default URL for the application corresponds to the route that shows the product table.

Tip You may need to stop the Angular development tools, start them again using the `ng serve` command, and then reload the browser for this example.



A screenshot of a web browser window titled "ExampleApp". The address bar shows "localhost:4200". The main content area displays a table with 9 rows of product data. The columns are labeled "ID", "Name", "Category", "Price", and "Details". Each row has a "Delete" and "Edit" button in the last column. At the bottom of the table, there are two buttons: "Create New Product" and "Generate HTTP Error".

ID	Name	Category	Price	Details	
1	Kayak	Watersports	\$275.00	Acme, boat, small	<button>Delete</button> <button>Edit</button>
2	Lifejacket	Watersports	\$48.95	Smooth Co, safety	<button>Delete</button> <button>Edit</button>
3	Soccer Ball	Soccer	\$19.50	(None)	<button>Delete</button> <button>Edit</button>
4	Corner Flags	Soccer	\$34.95	(None)	<button>Delete</button> <button>Edit</button>
5	Stadium	Soccer	\$79,500.00	(None)	<button>Delete</button> <button>Edit</button>
6	Thinking Cap	Chess	\$16.00	(None)	<button>Delete</button> <button>Edit</button>
7	Unsteady Chair	Chess	\$29.95	(None)	<button>Delete</button> <button>Edit</button>
8	Human Chess Board	Chess	\$75.00	(None)	<button>Delete</button> <button>Edit</button>
9	Bling Bling King	Chess	\$1,200.00	(None)	<button>Delete</button> <button>Edit</button>

[Create New Product](#) [Generate HTTP Error](#)

Figure 24-2. Using routing to display components to the user

Adding Navigation Links

The basic routing configuration is in place, but there is no way to navigate around the application: nothing happens when you click the Create New Product or Edit button.

The next step is to add links to the application that will change the browser's URL and, in doing so, trigger a routing change that will display a different component to the user. Listing 24-5 adds these links to the table component's template.

Listing 24-5. Adding Navigation Links in the table.component.html File in the src/app/core Folder

```
<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr>
      <th>ID</th><th>Name</th><th>Category</th><th>Price</th>
      <th>Details</th><th></th></th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of getProducts()">
      <td>{{item.id}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price | currency:"USD" }}</td>
      <td>
        <ng-container *ngIf="item.details else empty">
          {{ item.details?.supplier }}, {{ item.details?.keywords}}
        </ng-container>
        <ng-template #empty>(None)</ng-template>
      </td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm m-1"
               (click)="deleteProduct(item.id)">
          Delete
        </button>
        <button class="btn btn-warning btn-sm" (click)="editProduct(item.id)"
               routerLink="/form/edit">
          Edit
        </button>
      </td>
    </tr>
  </tbody>
</table>
<button class="btn btn-primary m-1" (click)="createProduct()"
       routerLink="/form/create">
  Create New Product
</button>
<button class="btn btn-danger m-1" (click)="deleteProduct(-1)">
  Generate HTTP Error
</button>
```

The `routerLink` attribute applies a directive from the routing package that performs the navigation change. This directive can be applied to any element, although it is typically applied to button and anchor (`a`) elements. The expression for the `routerLink` directive applied to the Edit buttons tells Angular to target the `/form/edit` route.

```
...
<button class="btn btn-warning btn-sm" (click)="editProduct(item.id)"
    routerLink="/form/edit">
    Edit
</button>
...
...
```

The same directive applied to the Create New Product button tells Angular to target the `/create` route.

```
...
<button class="btn btn-primary m-1" (click)="createProduct()"
    routerLink="/form/create">
    Create New Product
</button>
...
...
```

The routing links added to the table component's template will allow the user to navigate to the form. The addition to the form component's template shown in Listing 24-6 will allow the user to navigate back again using the Cancel button.

Listing 24-6. Adding a Navigation Link in the `form.component.html` File in the `src/app/core` Folder

```
...
<div class="mt-2">
    <button type="submit" class="btn btn-primary"
        [class.btn-warning]="editing"
        [disabled]="form.invalid">
        {{editing ? "Save" : "Create"}}
    </button>
    <button type="reset" class="btn btn-secondary m-1" routerLink="/">
        Cancel
    </button>
</div>
...
...
```

The value assigned to the `routerLink` attribute targets the route that displays the product table. Listing 24-7 updates the feature module that contains the template so that it imports the `RouterModule`, which is the Angular module that contains the directive that selects the `routerLink` attribute.

Listing 24-7. Enabling the Routing Directive in the `core.module.ts` File in the `src/app/core` Folder

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "./table.component";
```

```

import { FormComponent } from "./form.component";
import { SharedState } from "./sharedState.service";
import { ValidationHelper } from "./validation_helper";
import { ValidationErrorsDirective } from "./validationErrors.directive";
import { HilowValidatorDirective } from "../validation/hilow";
import { RouterModule } from "@angular/router";

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule, ReactiveFormsModule,
    RouterModule],
  declarations: [TableComponent, FormComponent, ValidationHelper,
    ValidationErrorsDirective, HilowValidatorDirective],
  exports: [ModelModule, TableComponent, FormComponent],
  providers: [SharedState]
})
export class CoreModule { }

```

Understanding the Effect of Routing

Restart the Angular development tools, and you will be able to navigate around the application using the Edit, Create New Product, and Cancel buttons, as shown in Figure 24-3.

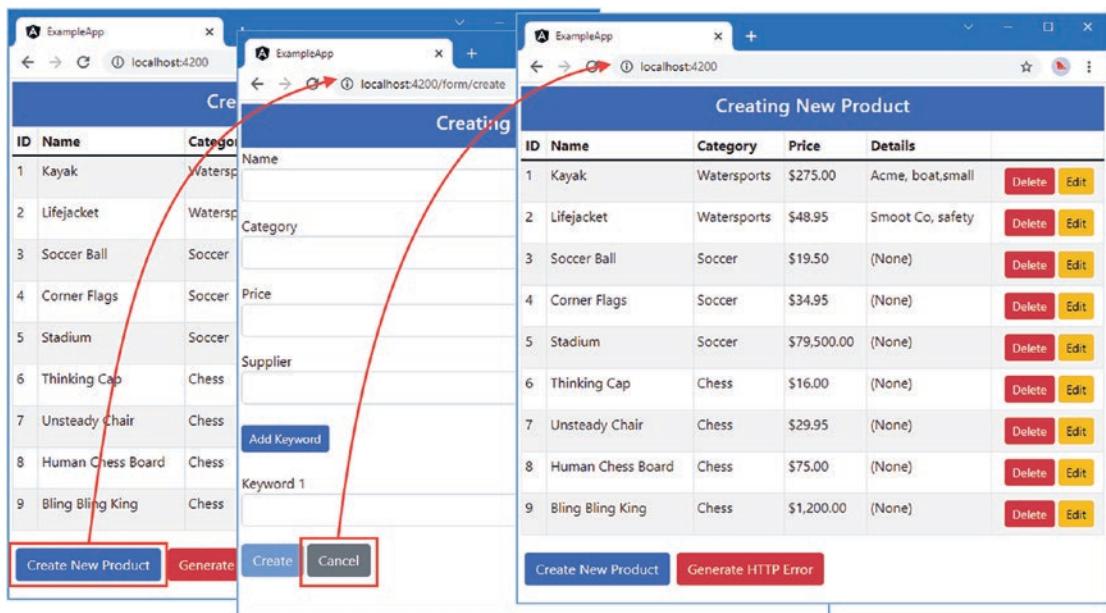


Figure 24-3. Using routes to navigate around the application

Not all the features in the application work yet, but this is a good time to explore the effect of adding routing to the application. Enter the root URL for the application (`http://localhost:4200`) and then click the Create New Product button. When you clicked the button, the Angular routing system changed the URL that the browser displays to this:

```
http://localhost:4200/form/create
```

If you watch the requests made by the application in the F12 development tools during the transition, you will notice that no requests are sent to the server for new content. This change is done entirely within the Angular application and does not produce any new HTTP requests.

The new URL is processed by the Angular routing system, which can match the new URL to this route from the `app.routing.ts` file.

```
{ path: "form/create", component: FormComponent },
```

The routing system takes into account the `base` element in the `index.html` file when it matches the URL to a route. The `base` element is configured with an `href` value of `/` that is combined with the `path` in the route to make a match when the URL is `/form/create`.

The `component` property tells the Angular routing system that it should display the `FormComponent` to the user. A new instance of the `FormComponent` class is created, and its template content is used as the content for the `router-outlet` element in the root component's template.

If you click the Cancel button below the form, then the process is repeated, but this time, the browser returns to the root URL for the application, which is matched by the route whose path component is the empty string.

```
{ path: "", component: TableComponent }
```

This route tells Angular to display the `TableComponent` to the user. A new instance of the `TableComponent` class is created, and its template is used as the content of the `router-outlet` element, displaying the model data to the user.

This is the essence of routing: the browser's URL changes, which causes the routing system to consult its configuration to determine which component should be displayed to the user. Lots of options and features are available, but this is the core purpose of routing, and you won't go too far wrong if you keep this in mind.

THE PERILS OF CHANGING THE URL MANUALLY

The `routerLink` directive sets the URL using a JavaScript API that tells the browser that this is a change relative to the current document and not a change that requires an HTTP request to the server.

If you enter a URL that matches the routing system into the browser window, you will see an effect that looks like the expected change but is actually something else entirely. Keep an eye on the network requests in the F12 development tools while manually entering the following URL into the browser:

`http://localhost:4200/form/create`

Rather than handling the change within the Angular application, the browser sends an HTTP request to the server, which reloads the application. Once the application is loaded, the routing system inspects the browser's URL, matches one of the routes in the configuration, and then displays the `FormComponent`.

The reason this works is that the development HTTP server will return the contents of the `index.html` file for URLs that don't correspond to files on the disk. As an example, request this URL:

```
http://localhost:4200/this/does/not/exist
```

The browser will display an error because the request has provided the browser with the contents of the `index.html` file, which it has used to load and start the example Angular application. When the routing system inspects the URL, it finds no matching route and creates an error.

There are two important points to note. The first is that when you test your application's routing configuration, you should check the HTTP requests that the browser is making because you will sometimes see the right result for the wrong reasons. On a fast machine, you may not even realize that the application has been reloaded and restarted by the browser.

Second, you must remember that the URL must be changed using the `routerLink` directive (or one of the similar features provided by the router module) and not manually, using the browser's URL bar.

Finally, since users won't know about the difference between programmatic and manual URL changes, your routing configuration should be able to deal with URLs that don't correspond to routes, as described in Chapter 25.

Completing the Routing Implementation

Adding routing to the application is a good start, but a lot of the application features just don't work. For example, clicking an Edit button displays the form, but it isn't populated, and it doesn't show the color cue that indicates editing. In the sections that follow, I use features provided by the routing system to finish wiring up the application so that everything works as expected.

Handling Route Changes in Components

The form component isn't working properly because it isn't being notified that the user has clicked a button to edit a product. This problem occurs because the routing system creates new instances of component classes only when it needs them, which means the `FormComponent` object is created only after the Edit button is clicked. If you click the Cancel button under the form and then click an Edit button in this table again, a second instance of the `FormComponent` will be created.

This leads to a timing issue in the way that the product component and the table component communicate, via a Reactive Extensions Subject. A Subject only passes events to subscribers that arrive after the `subscribe` method has been called. The introduction of routing means that the `FormComponent` object is created after the event describing the edit operation has already been sent.

This problem could be solved by replacing the Subject with a `BehaviorSubject`, which sends the most recent event to subscribers when they call the `subscribe` method. But a more elegant approach—especially since this is a chapter on the routing system—is to use the URL to collaborate between components.

Angular provides a service that components can receive to get details of the current route. The relationship between the service and the types that it provides access to may seem complicated at first, but it will make sense as you see how the examples unfold and some of the different ways that routing can be used.

The class on which components declare a dependency is called `ActivatedRoute`. For this section, it defines one important property, which is described in Table 24-5. There are other properties, too, which are described later in the chapter but which you can ignore for the moment.

Table 24-5. The `ActivatedRoute` Property

Name	Description
<code>snapshot</code>	This property returns an <code>ActivatedRouteSnapshot</code> object that describes the current route.

The `snapshot` property returns an instance of the `ActivatedRouteSnapshot` class, which provides information about the route that led to the current component being displayed to the user using the properties described in Table 24-6.

Table 24-6. The Basic `ActivatedRouteSnapshot` Properties

Name	Description
<code>url</code>	This property returns an array of <code>UrlSegment</code> objects, each of which describes a single segment in the URL that matched the current route.
<code>params</code>	This property returns a <code>Params</code> object, which describes the URL parameters, indexed by name.
<code>queryParams</code>	This property returns a <code>Params</code> object, which describes the URL query parameters, indexed by name.
<code>fragment</code>	This property returns a string containing the URL fragment.

The `url` property is the one that is most important for this example because it allows the component to inspect the segments of the current URL and extract the information from them that is required to perform an operation. The `url` property returns an array of `UrlSegment` objects, which provide the properties described in Table 24-7.

Table 24-7. The `UrlSegment` Properties

Name	Description
<code>path</code>	This property returns a string that contains the segment value.
<code>parameters</code>	This property returns an indexed collection of parameters, as described in the “Using Route Parameters” section.

To determine what route has been activated by the user, the form component can declare a dependency on `ActivatedRoute` and then use the object it receives to inspect the segments of the URL, as shown in Listing 24-8.

Listing 24-8. Inspecting the Active Route in the `form.component.ts` File in the `src/app/core` Folder

```
import { Component } from "@angular/core";
import { FormControl, NgForm, Validators, FormGroup, FormArray }
    from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
```

```

import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";
import { FilteredFormArray } from "./filteredFormArray";
import { LimitValidator } from "../validation/limit";
import { UniqueValidator } from "../validation/unique";
import { ProhibitedValidator } from "../validation/prohibited";
import { ActivatedRoute } from "@angular/router";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;

  // ...form structure omitted for brevity...

  constructor(private model: Model, activeRoute: ActivatedRoute) {
    this.editing = activeRoute.snapshot.url[1].path == "edit";
  }

  // handleStateChange(newState: StateUpdate) {
  //   this.editing = newState.mode == MODES.EDIT;
  //   this.keywordGroup.clear();
  //   if (this.editing && newState.id) {
  //     Object.assign(this.product, this.model.getProduct(newState.id)
  //       ?? new Product());
  //     this.messageService.reportMessage(
  //       new Message(`Editing ${this.product.name}`));
  //     this.product.details?.keywords?.forEach(val => {
  //       this.keywordGroup.push(this.createKeywordFormControl());
  //     })
  //   } else {
  //     this.product = new Product();
  //     this.messageService.reportMessage(new Message("Creating New Product"));
  //   }
  //   if (this.keywordGroup.length == 0) {
  //     this.keywordGroup.push(this.createKeywordFormControl());
  //   }
  //   this.productForm.reset(this.product);
  // }

  // ...other methods omitted for brevity...
}

```

The component no longer uses the shared state service to receive events. Instead, it inspects the second segment of the active route's URL to set the value of the `editing` property, which determines whether it should display its create or edit mode. If you click an Edit button in the table, you will now see the correct coloring displayed, as shown in Figure 24-4, although the fields are not yet populated with data.

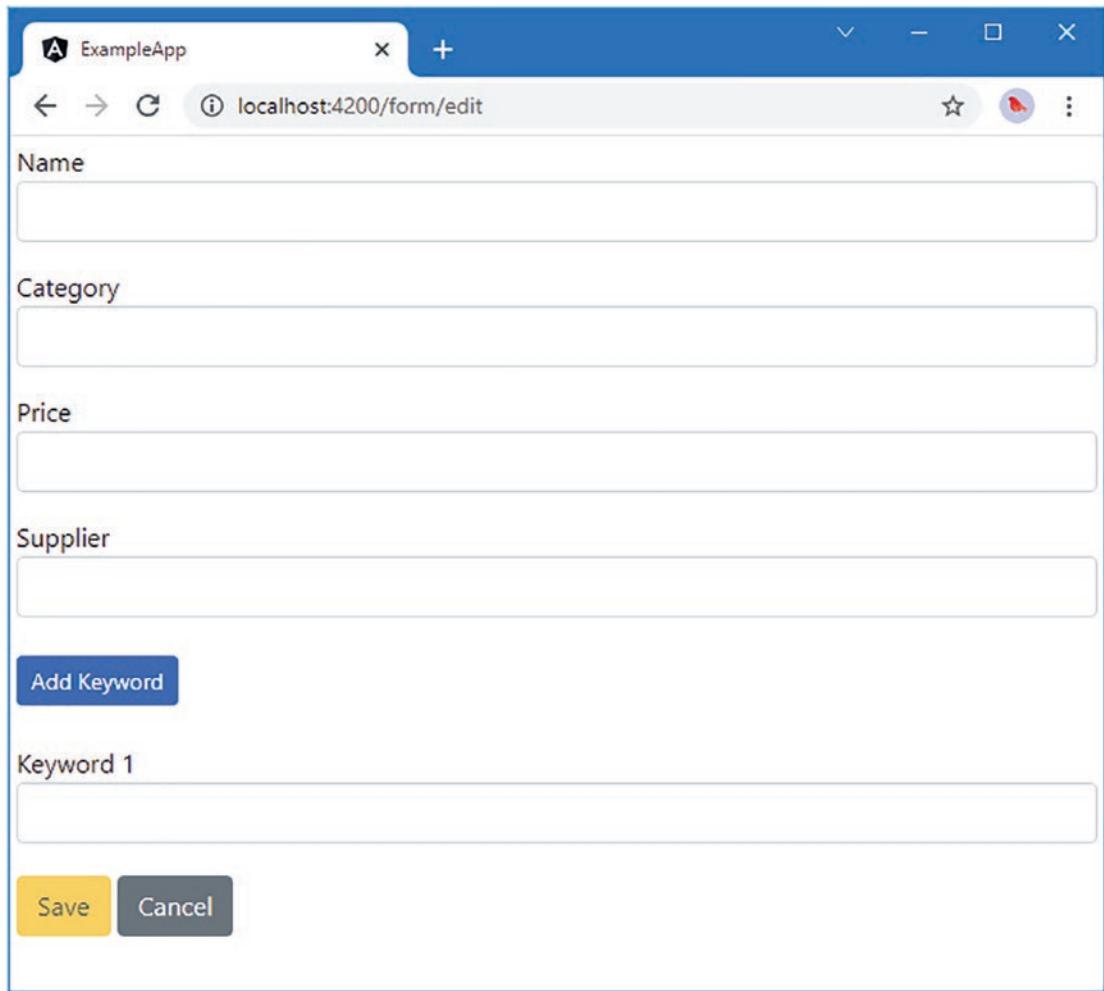


Figure 24-4. Using the active route in a component

Using Route Parameters

When I set up the routing configuration for the application, I defined two routes that targeted the form component, like this:

```
...
{ path: "form/edit", component: FormComponent },
{ path: "form/create", component: FormComponent },
...
```

When Angular is trying to match a route to a URL, it looks at each segment in turn and checks to see that it matches the URL that is being navigated to. Both of these URLs are made up of *static segments*, which means they have to match the navigated URL exactly before Angular will activate the route.

Angular routes can be more flexible and include *route parameters*, which allow any value for a segment to match the corresponding segment in the navigated URL. This means routes that target the same component with similar URLs can be consolidated into a single route, as shown in Listing 24-9.

Listing 24-9. Consolidating Routes in the app.routing.ts File in the src/app Folder

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";

const routes: Routes = [
  // { path: "form/edit", component: FormComponent },
  // { path: "form/create", component: FormComponent },
  { path: "form/:mode", component: FormComponent },
  { path: "", component: TableComponent }
]

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

The second segment of the modified URL defines a route parameter, denoted by the colon (the : character) followed by a name. In this case, the route parameter is called mode. This route will match any URL that has two segments where the first segment is form, as summarized in Table 24-8. The content of the second segment will be assigned to a parameter called mode.

Table 24-8. URL Matching with the Route Parameter

URL	Result
http://localhost:4200/form	No match—too few segments
http://localhost:4200/form/create	Matches, with create assigned to the mode parameter
http://localhost:4200/form/london	Matches, with london assigned to the mode parameter
http://localhost:4200/product/edit	No match—the first segment is not form
http://localhost:4200/form/edit/1	No match—too many segments

Using route parameters makes it simpler to handle routes programmatically because the value of the parameter can be obtained using its name, as shown in Listing 24-10.

Listing 24-10. Reading a Route Parameter in the form.component.ts File in the src/app/core Folder

```
...
constructor(private model: Model, activeRoute: ActivatedRoute) {
  this.editing = activeRoute.snapshot.params["mode"] == "edit";
}
...
```

The component doesn't need to know the structure of the URL to get the information it needs. Instead, it can use the params property provided by the ActivatedRouteSnapshot class to get a collection of the parameter values, indexed by name. The component gets the value of the mode parameter and uses it to set the editing property.

Using Multiple Route Parameters

To tell the form component which product has been selected when the user clicks an Edit button, I need to use a second route parameter. Since Angular matches URLs based on the number of segments they contain, this means I need to split up the routes that target the form component again, as shown in Listing 24-11. This cycle of consolidating and then expanding routes is typical of most development projects as you increase the amount of information that is included in routed URLs to add functionality to the application.

Listing 24-11. Adding a Route in the app.routing.ts File in the src/app Folder

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";

const routes: Routes = [
  { path: "form/:mode/:id", component: FormComponent },
  { path: "form/:mode", component: FormComponent },
  { path: "", component: TableComponent }];

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

The new route will match any URL that has three segments where the first segment is `form`. To create URLs that target this route, I need to use a different approach for the `routerLink` expressions in the template because I need to generate the third segment dynamically for each Edit button in the product table, as shown in Listing 24-12.

Listing 24-12. Generating Dynamic URLs in the table.component.html File in the src/app/core Folder

```
...
<td class="text-center">
  <button class="btn btn-danger btn-sm m-1"
    (click)="deleteProduct(item.id)">
    Delete
  </button>
  <button class="btn btn-warning btn-sm" (click)="editProduct(item.id)"
    [routerLink]="['/form', 'edit', item.id]">
    Edit
  </button>
</td>
...
```

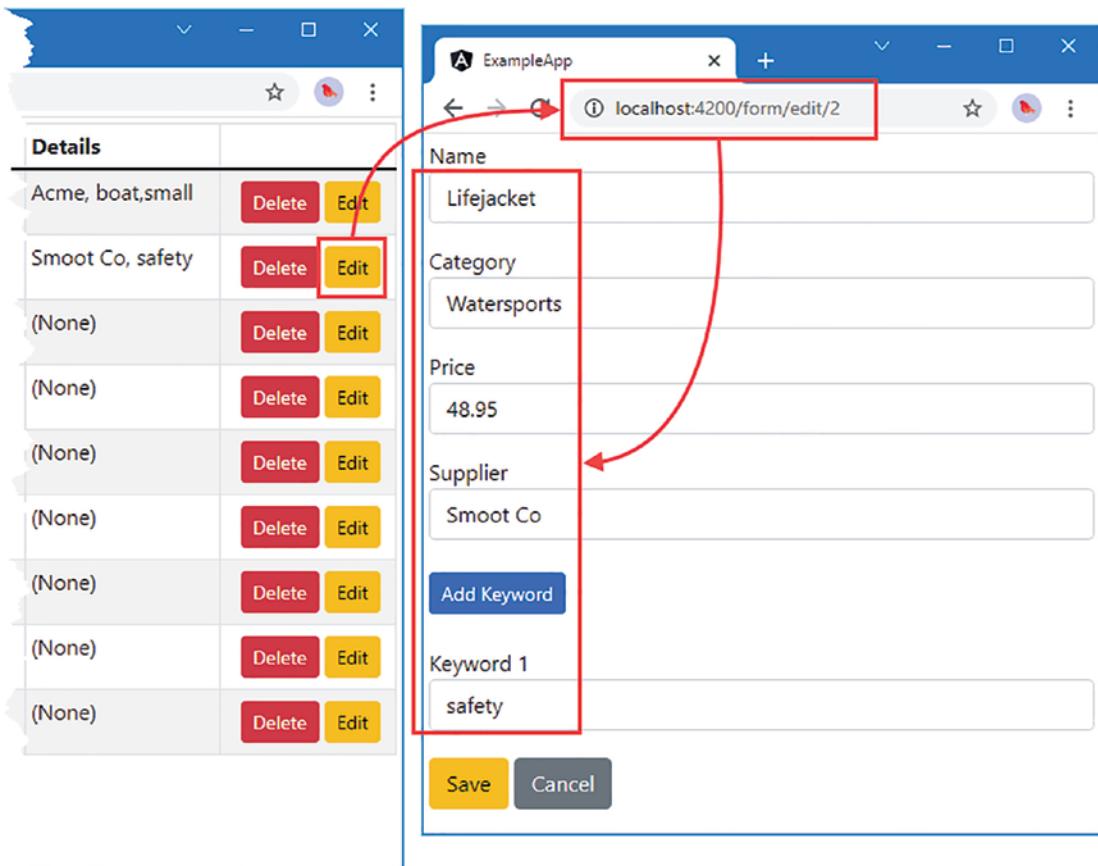
The `routerLink` attribute is now enclosed in square brackets, telling Angular that it should treat the attribute value as a data binding expression. The expression is set out as an array, with each element containing the value for one segment. The first two segments are literal strings and will be included in the target URL without modification. The third segment will be evaluated to include the `id` property value for the current `Product` object being processed by the `ngIf` directive, just like the other expressions in the template. The `routerLink` directive will combine the individual segments to create a URL such as `/form/edit/2`.

Listing 24-13 shows how the form component gets the value of the new route parameter and uses it to select the product that is to be edited.

Listing 24-13. Using the New Route Parameter in the form.component.ts File in the src/app/core Folder

```
...
constructor(private model: Model, activeRoute: ActivatedRoute) {
  this.editing = activeRoute.snapshot.params["mode"] == "edit";
  let id = activeRoute.snapshot.params["id"];
  if (id != null) {
    Object.assign(this.product, model.getProduct(id) || new Product());
    this.productForm.patchValue(this.product);
  }
}
...
```

When the user clicks an Edit button, the routing URL that is activated tells the form component that an edit operation is required and specifies the product is to be modified, allowing the form to be populated correctly, as shown in Figure 24-5.

**Figure 24-5.** Using URLs segments to provide information

Dealing with Direct Data Access

The introduction of routing has revealed a problem with the way that data is obtained from the web service. If the user starts by requesting `http://localhost:4200` and clicks one of the Edit buttons, then the application works as expected and the form is correctly populated with data.

But if the user navigates directly to the URL for editing a product, such as `http://localhost:4200/form/edit/2`, then the form is never populated with data. This is because the `RestDataSource` class has been written to assume that individual `Product` objects will be accessed only by clicking an Edit button, which can be done only once the data has been received from the web service.

In Chapter 26, I explain how you can stop routes from being activated until a specific condition is true, such as the arrival of the data, but another approach is to use observables to ensure that data values can be requested directly. The first step is to enhance the repository, as shown in Listing 24-14.

Listing 24-14. Using Observables in the `repository.model.ts` File in the `src/app/model` Folder

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { StaticDataSource } from "./static.datasource";
import { Observable, ReplaySubject } from "rxjs";
import { RestDataSource } from "./rest.datasource";

@Injectable()
export class Model {
    private products: Product[];
    private locator = (p: Product, id?: number) => p.id == id;
    private replaySubject: ReplaySubject<Product[]>;

    constructor(private dataSource: RestDataSource) {
        this.products = new Array<Product>();
        this.replaySubject = new ReplaySubject<Product[]>(1);
        this.dataSource.getData().subscribe(data => {
            this.products = data
            this.replaySubject.next(data);
            this.replaySubject.complete();
        });
    }

    getProducts(): Product[] {
        return this.products;
    }

    getProduct(id: number): Product | undefined {
        return this.products.find(p => this.locator(p, id));
    }
}
```

```

getProductObservable(id: number): Observable<Product | undefined> {
  let subject = new ReplaySubject<Product | undefined>(1);
  this.replaySubject.subscribe(products => {
    subject.next(products.find(p => this.locator(p, id)));
    subject.complete();
  });
  return subject;
}

// ...other methods omitted for brevity...
}

```

The changes rely on the ReplaySubject class to ensure that individual Product objects can be received even if the call to the new getProductObservable method is made before the data requested by the constructor has arrived. The ReplaySubject is useful for this problem because it allows subsequent calls to the getProductObservable method to benefit from the data already produced. Listing 24-15 updates the form component to use the new repository method.

Listing 24-15. Supporting Direct Access in the form.component.ts File in the src/app/core Folder

```

...
constructor(private model: Model, activeRoute: ActivatedRoute) {
  this.editing = activeRoute.snapshot.params["mode"] == "edit";
  let id = activeRoute.snapshot.params["id"];
  if (id != null) {
    model.getProductObservable(id).subscribe(p => {
      Object.assign(this.product, p || new Product());
      this.productForm.patchValue(this.product);
    });
  }
}
...

```

The use of multiple observables is a little awkward, but the effect is that the user can request a URL such as <http://localhost:4200/form/edit/2> directly and see the data they expect, as shown in Figure 24-6.

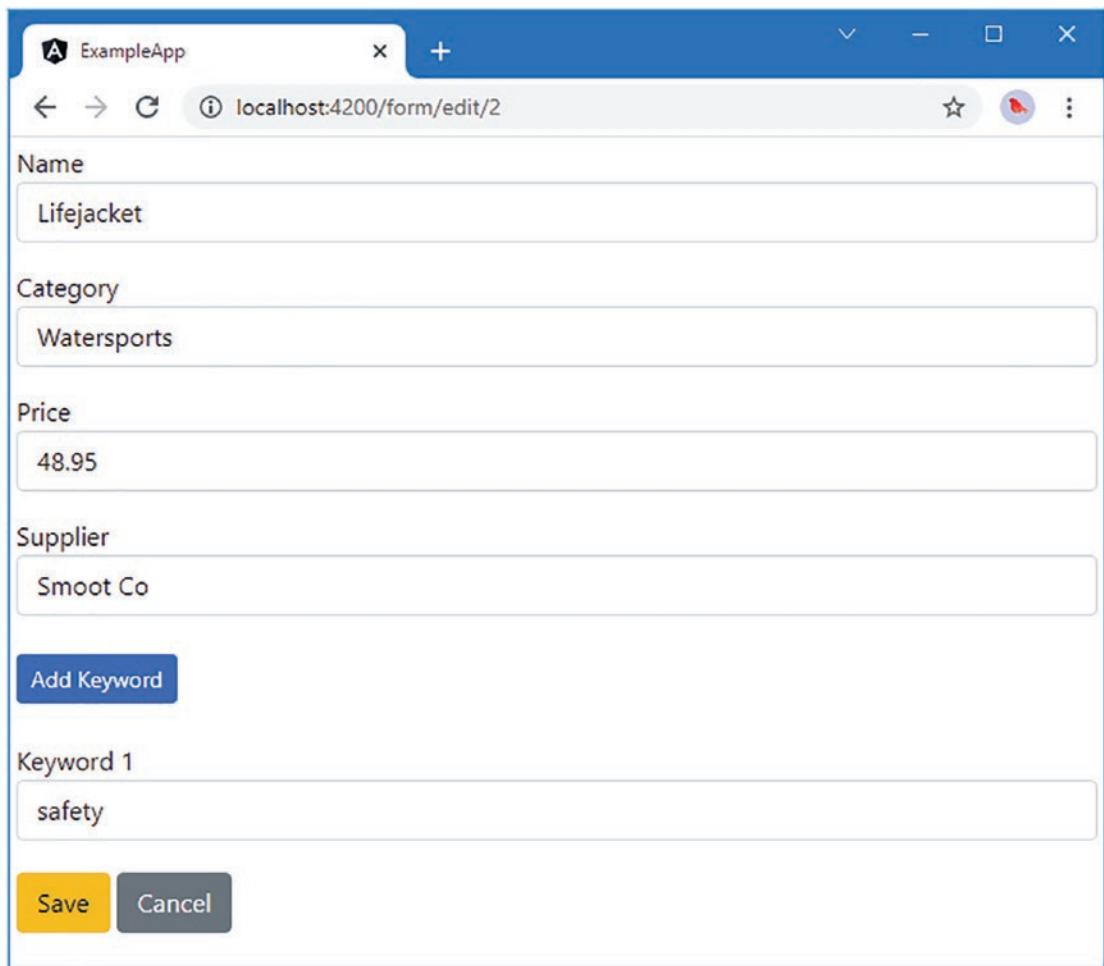


Figure 24-6. Providing direct access to data

Using Optional Route Parameters

Optional route parameters allow URLs to include information to provide hints or guidance to the rest of the application, but this is not essential for the application to work.

This type of route parameter is expressed using URL matrix notation, which isn't part of the specification for URLs but which browsers support nonetheless. Here is an example of a URL that has optional route parameters:

`http://localhost:4200/form/edit/2;name=Lifejacket;price=48.95`

The optional route parameters are separated by semicolons (the ; character), and this URL includes optional parameters called name and price.

As a demonstration of how to use optional parameters, Listing 24-16 shows the addition of an optional route parameter that includes the object to be edited as part of the URL.

Listing 24-16. An Optional Route Parameter in the table.component.html File in the src/app/core Folder

```
...
<button class="btn btn-warning btn-sm" (click)="editProduct(item.id)"
    [routerLink]=["/form", 'edit', item.id,
    {name: item.name, category: item.category, price: item.price}]">
    Edit
</button>
...
```

The optional values are expressed as literal objects, where property names identify the optional parameter. In this example, there are name, category, and price properties, and their values are set.

Listing 24-17 shows how the form component checks to see whether the optional parameters are present. If they have been included in the URL, then the parameter values are used to avoid a request to the data model.

Listing 24-17. Receiving Optional Parameters in the form.component.ts File in the src/app/core Folder

```
...
constructor(private model: Model, activeRoute: ActivatedRoute) {
    this.editing = activeRoute.snapshot.params["mode"] == "edit";
    let id = activeRoute.snapshot.params["id"];
    if (id != null) {
        model.getProductObservable(id).subscribe(p => {
            Object.assign(this.product, p || new Product());
            this.product.name = activeRoute.snapshot.params["name"]
                ?? this.product.name;
            this.product.category = activeRoute.snapshot.params["category"]
                ?? this.product.category;
            let price = activeRoute.snapshot.params["price"];
            if (price != null) {
                this.product.price = Number.parseFloat(price);
            }
            this.productForm.patchValue(this.product);
        });
    }
}
```

The optional parameters in Listing 24-16 will produce a URL like this one for the Edit buttons:

<http://localhost:4200/form/edit/5;name=Stadium;category=Soccer;price=79500>

Optional route parameters are accessed in the same way as required parameters, and it is the responsibility of the component to check to see whether they are present and to proceed anyway if they are not part of the URL. In this case, the component uses the optional parameter values to override the values from the repository, which you can see by requesting this URL:

```
http://localhost:4200/form/edit/5;category=Football
```

The supplied value for the category parameter overrides the value provided by the repository, as shown in Figure 24-7.

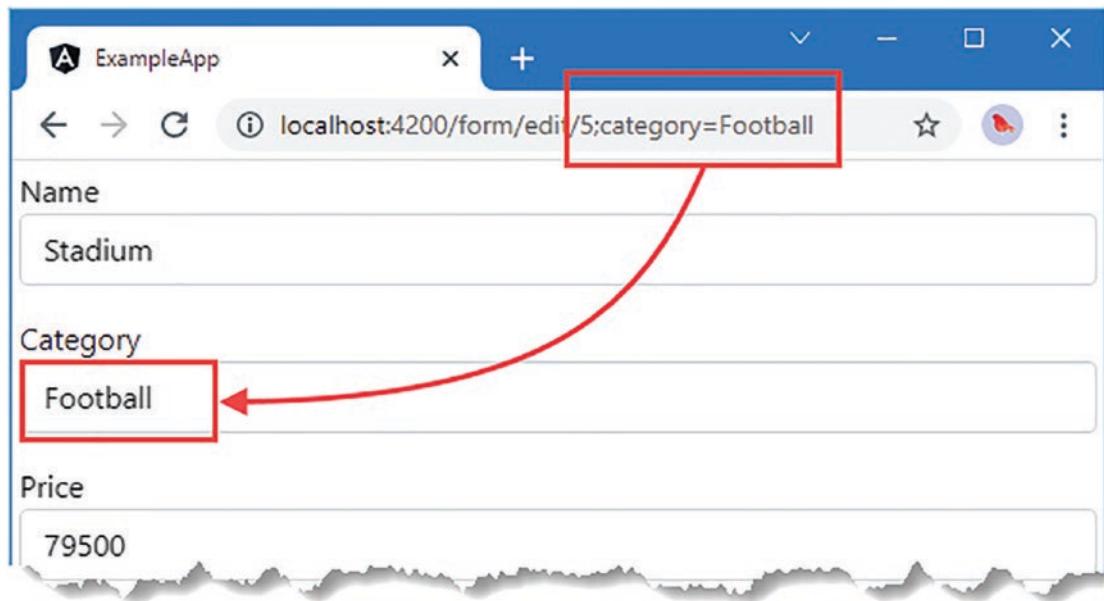


Figure 24-7. Using optional route parameters

Navigating in Code

Using the `routerLink` attribute makes it easy to set up navigation in templates, but applications will often need to initiate navigation on behalf of the user within a component or directive.

To give access to the routing system to building blocks such as directives and components, Angular provides the `Router` class, which is available as a service through dependency injection and whose most useful methods and properties are described in Table 24-9.

Table 24-9. Selected Router Methods and Properties

Name	Description
navigated	This boolean property returns true if there has been at least one navigation event and false otherwise.
url	This property returns the active URL.
isActive(url, exact)	This method returns true if the specified URL is the URL defined by the active route. The exact argument specified whether all the segments in the specified URL must match the current URL for the method to return true.
events	This property returns an Observable<Event> that can be used to monitor navigation changes. See the “Receiving Navigation Events” section for details.
navigateByUrl(url, extras)	This method navigates to the specified URL. The result of the method is a Promise, which resolves with true when the navigation is successful and false when it is not, and which is rejected when there is an error.
navigate(commands, extras)	This method navigates using an array of segments. The extras object can be used to specify whether the change of URL is relative to the current route. The result of the method is a Promise, which resolves with true when the navigation is successful and false when it is not, and which is rejected when there is an error.

The navigate and navigateByUrl methods make it easy to perform navigation inside a building block such as a component. Listing 24-18 shows the use of the Router in the form component to redirect the application back to the table after a product has been created or updated.

Listing 24-18. Navigating Programmatically in the form.component.ts File in the src/app/core Folder

```
import { Component } from "@angular/core";
import { FormControl, NgForm, Validators, FormGroup, FormArray }
  from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";
import { FilteredFormArray } from "./filteredFormArray";
import { LimitValidator } from "../validation/limit";
import { UniqueValidator } from "../validation/unique";
import { ProhibitedValidator } from "../validation/prohibited";
import { ActivatedRoute, Router } from "@angular/router";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;
```

```

// ...form structure omitted for brevity...

constructor(private model: Model, activeRoute: ActivatedRoute,
    private router: Router) {
  this.editing = activeRoute.snapshot.params["mode"] == "edit";
  let id = activeRoute.snapshot.params["id"];
  if (id != null) {
    model.getProductObservable(id).subscribe(p => {
      Object.assign(this.product, p || new Product());
      this.product.name = activeRoute.snapshot.params["name"]
        ?? this.product.name;
      this.product.category = activeRoute.snapshot.params["category"]
        ?? this.product.category;
      let price = activeRoute.snapshot.params["price"];
      if (price != null) {
        this.product.price == Number.parseFloat(price);
      }
      this.productForm.patchValue(this.product);
    });
  }
}

submitForm() {
  if (this.productForm.valid) {
    Object.assign(this.product, this.productForm.value);
    this.model.saveProduct(this.product);
    // this.product = new Product();
    // this.keywordGroup.clear();
    // this.keywordGroup.push(this.createKeywordFormControl());
    // this.productForm.reset();
    this.router.navigateByUrl("/");
  }
}

// ...methods omitted for brevity...
}

```

The component receives the Router object as a constructor argument and uses it in the `submitForm` method to navigate back to the application's root URL. The statements that have been commented out in the `submitForm` method are no longer required because the routing system will destroy the form component once it is no longer on display, which means that resetting the form's state is not required.

The result is that clicking the Save or Create button in the form will cause the application to display the product table, as shown in Figure 24-8.

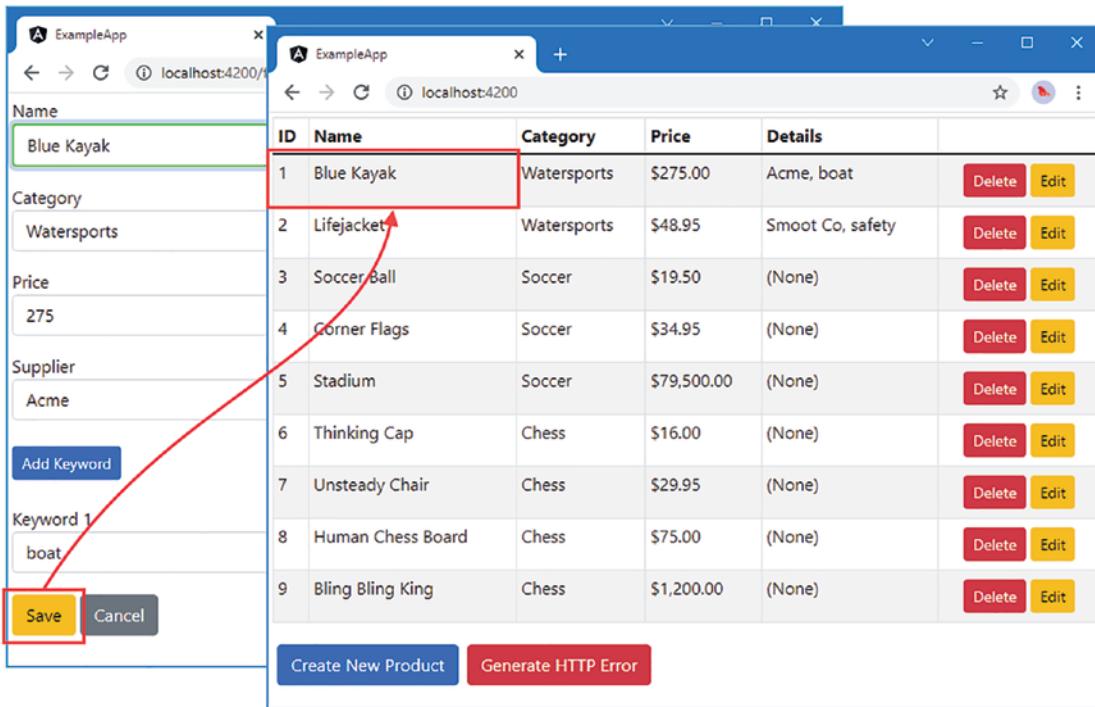


Figure 24-8. Navigating programmatically

Receiving Navigation Events

In many applications, there will be components or directives that are not directly involved in the application's navigation but that still need to know when navigation occurs. The example application contains an example in the message component, which displays notifications and errors to the user. This component always displays the most recent message, even when that information is stale and unlikely to be helpful to the user. To see the problem, click the Generate HTTP Error button and then click the Create New Product button or one of the Edit buttons; the error message remains on display even though you have navigated elsewhere in the application, as shown in Figure 24-9.

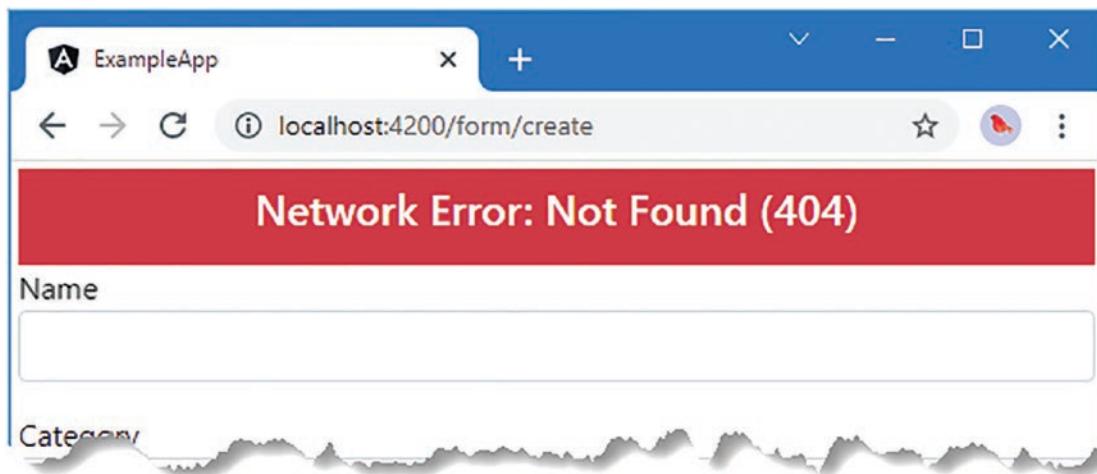


Figure 24-9. An outdated error message

The `events` property defined by the `Router` class returns an `Observable<Event>`, which emits a sequence of `Event` objects describing changes from the routing system. Table 24-10 describes the most useful events.

Table 24-10. Useful Events Provided by the Router Observer

Name	Description
<code>NavigationStart</code>	This event is sent when the navigation process starts.
<code>RoutesRecognized</code>	This event is sent when the routing system matches the URL to a route.
<code>NavigationEnd</code>	This event is sent when the navigation process completes successfully.
<code>NavigationError</code>	This event is sent when the navigation process produces an error.
<code>NavigationCancel</code>	This event is sent when the navigation process is canceled.
<code>NavigationError</code>	This event is sent when an error arises during navigation.

All the event classes define an `id` property, which returns a number that is incremented for each navigation, and a `url` property, which returns the target URL. The `RoutesRecognized` and `NavigationEnd` events also define a `urlAfterRedirects` property, which returns the URL that has been navigated to.

To address the issue with the messaging system, Listing 24-19 subscribes to the `Observer` provided by the `Router.events` property and clears the message displayed to the user when the `NavigationEnd` or `NavigationCancel` event is received.

Listing 24-19. Responding to Events in the `message.component.ts` File in the `src/app/messages` Folder

```
import { Component } from "@angular/core";
import { MessageService } from "./message.service";
import { Message } from "./message.model";
import { Router, NavigationEnd, NavigationCancel } from "@angular/router";
```

```

@Component({
  selector: "paMessages",
  templateUrl: "message.component.html",
})
export class MessageComponent {
  lastMessage?: Message;

  constructor(messageService: MessageService, router: Router) {
    messageService.messages.subscribe(msg => this.lastMessage = msg);
    router.events.subscribe(e => {
      if (e instanceof NavigationEnd || e instanceof NavigationCancel) {
        this.lastMessage = undefined;
      }
    })
  }
}

```

The result of these changes is that messages are shown to the user only until the next navigation event, as shown in Figure 24-10.

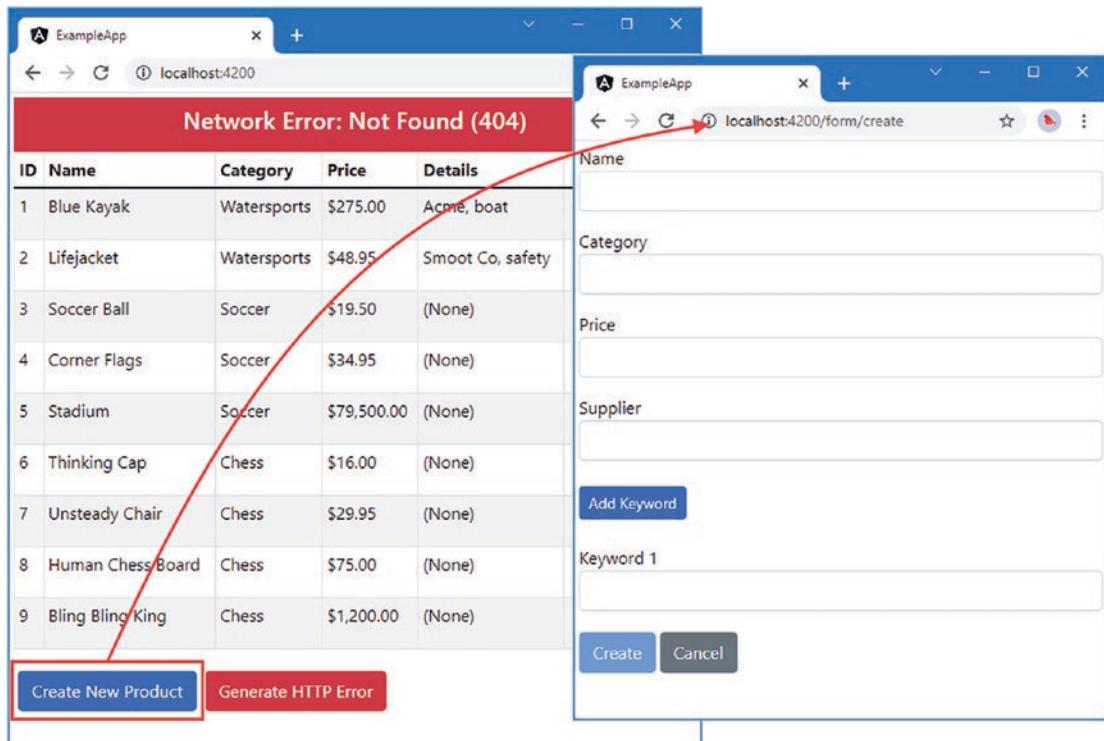


Figure 24-10. Responding to navigation events

Removing the Event Bindings and Supporting Code

One of the benefits of using the routing system is that it can simplify applications, replacing event bindings and the methods they invoke with navigation changes. The final change to complete the routing implementation is to remove the last traces of the previous mechanism that was used to coordinate between components. Listing 24-20 removes the event bindings from the table component's template, which were used to respond when the user clicked the Create New Product or Edit button. (The event binding for the Delete buttons is still required because this feature does not relate to navigation.)

Listing 24-20. Removing Event Bindings in the table.component.html File in the src/app/core Folder

```
<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr>
      <th>ID</th><th>Name</th><th>Category</th><th>Price</th>
      <th>Details</th><th></th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of getProducts()">
      <td>{{item.id}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price | currency:"USD" }}</td>
      <td>
        <ng-container *ngIf="item.details else empty">
          {{ item.details?.supplier }}, {{ item.details?.keywords }}
        </ng-container>
        <ng-template #empty>(None)</ng-template>
      </td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm m-1"
               (click)="deleteProduct(item.id)">
          Delete
        </button>
        <button class="btn btn-warning btn-sm"
               [routerLink]=[ '/form', 'edit', item.id ]">
          Edit
        </button>
      </td>
    </tr>
  </tbody>
</table>
<button class="btn btn-primary m-1" routerLink="/form/create">
  Create New Product
</button>
<button class="btn btn-danger m-1" (click)="deleteProduct(-1)">
  Generate HTTP Error
</button>
```

Listing 24-21 shows the corresponding changes in the component, which remove the methods that the event bindings invoked and remove the dependency on the service that was used to signal when a product should be edited or created.

Listing 24-21. Removing Event Handling Code in the table.component.ts File in the src/app/core Folder

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
//import { MODES, SharedState } from "./sharedState.service";

@Component({
  selector: "paTable",
  templateUrl: "table.component.html"
})
export class TableComponent {

  constructor(private model: Model) { }

  getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  getProducts(): Product[] {
    return this.model.getProducts();
  }

  deleteProduct(key?: number) {
    if (key != undefined) {
      this.model.deleteProduct(key);
    }
  }

  // editProduct(key?: number) {
  //   this.state.update(MODES.EDIT, key)
  // }

  // createProduct() {
  //   this.state.update(MODES.CREATE);
  // }
}
```

The service used for coordination by the components is no longer required, and Listing 24-22 disables it from the core module.

Listing 24-22. Removing the Shared State Service in the core.module.ts File in the src/app/core Folder

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "./table.component";
```

```

import { FormComponent } from "./form.component";
//import { SharedState } from "./sharedState.service";
import { ValidationHelper } from "./validation_helper";
import { ValidationErrorsDirective } from "./validationErrors.directive";
import { HilowValidatorDirective } from "../validation/hilow";
import { RouterModule } from "@angular/router";

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule, ReactiveFormsModule,
    RouterModule],
  declarations: [TableComponent, FormComponent, ValidationHelper,
    ValidationErrorsDirective, HilowValidatorDirective],
  exports: [ModelModule, TableComponent, FormComponent],
  //providers: [SharedState]
})
export class CoreModule { }

```

The result is that the coordination between the table and form components is handled entirely through the routing system, which is now responsible for displaying the components and managing the navigation between them.

Summary

In this chapter, I introduced the Angular routing feature and demonstrated how to navigate to a URL in an application to select the content that is displayed to the user. I showed you how to create navigation links in templates, how to perform navigation in a component or directive, and how to respond to navigation changes programmatically. In the next chapter, I continue to describe the Angular routing system.

CHAPTER 25



Routing and Navigation: Part 2

In the previous chapter, I introduced the Angular URL routing system and explained how it can be used to control the components that are displayed to the user. The routing system has a lot of features, which I continue to describe in this chapter and Chapter 26. This emphasis in this chapter is about creating more complex routes, including routes that will match any URL, routes that redirect the browser to other URLs, routes that navigate within a component, and routes that select multiple components. Table 25-1 summarizes the chapter.

Table 25-1. Chapter Summary

Problem	Solution	Listing
Matching multiple URLs with a single route	Use routing wildcards	1–8
Redirecting one URL to another	Use a redirection route	9
Navigating within a component	Use a relative URL	10
Receiving notifications when the activated URL changes	Use the <code>Observable</code> objects provided by the <code>ActivatedRoute</code> class	11
Styling an element when a specific route is active	Use the <code>routerLinkActive</code> attribute	12–15
Using the routing system to display nested components	Define child routes and use the <code>router-outlet</code> element	16–20

Preparing the Example Project

For this chapter, I will continue using the `exampleApp` project that was created in Chapter 20 and has been modified in each subsequent chapter. To prepare for this chapter, I have added two methods to the `repository` class, as shown in Listing 25-1.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 25-1. Adding Methods in the repository.model.ts File in the src/app/model Folder

```

import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { StaticDataSource } from "./static.datasource";
import { Observable, ReplaySubject } from "rxjs";
import { RestDataSource } from "./rest.datasource";

@Injectable()
export class Model {
    private products: Product[];
    private locator = (p: Product, id?: number) => p.id == id;
    private replaySubject: ReplaySubject<Product[]>;

    constructor(private dataSource: RestDataSource) {
        this.products = new Array<Product>();
        this.replaySubject = new ReplaySubject<Product[]>(1);
        this.dataSource.getData().subscribe(data => {
            this.products = data
            this.replaySubject.next(data);
            this.replaySubject.complete();
        });
    }

    getProducts(): Product[] {
        return this.products;
    }

    getProduct(id: number): Product | undefined {
        return this.products.find(p => this.locator(p, id));
    }

    getProductObservable(id: number): Observable<Product | undefined> {
        let subject = new ReplaySubject<Product | undefined>(1);
        this.replaySubject.subscribe(products => {
            subject.next(products.find(p => this.locator(p, id)));
            subject.complete();
        });
        return subject;
    }

    getNextProductId(id?: number): Observable<number> {
        let subject = new ReplaySubject<number>(1);
        this.replaySubject.subscribe(products => {
            let nextId = 0;
            let index = products.findIndex(p => this.locator(p, id));
            if (index > -1) {
                nextId = products[products.length > index + 1
                    ? index + 1 : 0].id ?? 0;
            } else {
                nextId = id || 0;
            }
        });
    }
}

```

```

        subject.next(nextId);
        subject.complete();
    });
    return subject;
}

getPreviousProductId(id?: number): Observable<number> {
    let subject = new ReplaySubject<number>(1);
    this.replaySubject.subscribe(products => {
        let nextId = 0;
        let index = products.findIndex(p => this.locator(p, id));
        if (index > -1) {
            nextId = products[index > 0
                ? index - 1 : products.length - 1].id ?? 0;
        } else {
            nextId = id || 0;
        }
        subject.next(nextId);
        subject.complete();
    });
    return subject;
}

saveProduct(product: Product) {
    if (product.id == 0 || product.id == null) {
        this.dataSource.saveProduct(product)
            .subscribe(p => this.products.push(p));
    } else {
        this.dataSource.updateProduct(product).subscribe(p => {
            let index = this.products
                .findIndex(item => this.locator(item, p.id));
            this.products.splice(index, 1, p);
        });
    }
}

deleteProduct(id: number) {
    this.dataSource.deleteProduct(id).subscribe(() => {
        let index = this.products.findIndex(p => this.locator(p, id));
        if (index > -1) {
            this.products.splice(index, 1);
        }
    });
}
}

```

The new methods accept an ID value, locate the corresponding product, and then return observables that produce the IDs of the next and previous objects in the array that the repository uses to collect the data model objects. I will use this feature later in the chapter to allow the user to page through the set of objects in the data model.

To simplify the example, Listing 25-2 removes the statements in the form component that receive the details of the product to edit using optional route parameters. I also changed the access level for the constructor parameters so I can use them directly in the component's template.

Listing 25-2. Removing Optional Parameters in the form.component.ts File in the src/app/core Folder

```
import { Component } from "@angular/core";
import { FormControl, NgForm, Validators, FormGroup, FormArray }
  from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";
import { FilteredFormArray } from "./filteredFormArray";
import { LimitValidator } from "../validation/limit";
import { UniqueValidator } from "../validation/unique";
import { ProhibitedValidator } from "../validation/prohibited";
import { ActivatedRoute, Router } from "@angular/router";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;

  keywordGroup = new FilteredFormArray([
    this.createKeywordFormControl(),
  ], {
    validators: UniqueValidator.unique()
  })

  productForm: FormGroup = new FormGroup({
    name: new FormControl("", {
      validators: [
        Validators.required,
        Validators.minLength(3),
        Validators.pattern("^[A-Za-z ]+$")
      ],
      updateOn: "change"
    }),
    category: new FormControl("", {
      validators: Validators.required,
      asyncValidators: ProhibitedValidator.prohibited()
    }),
    price: new FormControl("", {
      validators: [
        Validators.required, Validators.pattern("^[0-9\\.]+$"),
        LimitValidator.Limit(300)
      ]
    })
  })
}
```

```

        ]
    }),
details: new FormGroup({
    supplier: new FormControl("", { validators: Validators.required }),
    keywords: this.keywordGroup
})
});

constructor(public model: Model, activeRoute: ActivatedRoute,
public router: Router) {
this.editing = activeRoute.snapshot.params["mode"] == "edit";
let id = activeRoute.snapshot.params["id"];
if (id != null) {
    model.getProductObservable(id).subscribe(p => {
        Object.assign(this.product, p || new Product());
        // this.product.name = activeRoute.snapshot.params["name"]
        // ?? this.product.name;
        // this.product.category = activeRoute.snapshot.params["category"]
        // ?? this.product.category;
        // let price = activeRoute.snapshot.params["price"];
        // if (price != null) {
        //     this.product.price = Number.parseFloat(price);
        // }
        this.productForm.patchValue(this.product);
    });
}
}

submitForm() {
if (this.productForm.valid) {
    Object.assign(this.product, this.productForm.value);
    this.model.saveProduct(this.product);
    this.router.navigateByUrl("/");
}
}

resetForm() {
this.keywordGroup.clear();
this.keywordGroup.push(this.createKeywordFormControl());
this.editing = true;
this.product = new Product();
this.productForm.reset();
}

createKeywordFormControl(): FormControl {
return new FormControl("", { validators:
    Validators.pattern("^[A-Za-z ]+$" ) });
}

addKeywordControl() {
    this.keywordGroup.push(this.createKeywordFormControl());
}
}

```

```

removeKeywordControl(index: number) {
    this.keywordGroup.removeAt(index);
}
}

```

Adding Components to the Project

I need to add some components to the application to demonstrate some of the features covered in this chapter. These components are simple because I am focusing on the routing system, rather than adding useful features to the application. I created a file called `productCount.component.ts` in the `src/app/core` folder and used it to define the component shown in Listing 25-3.

Tip You can omit the `selector` attribute from the `@Component` decorator if a component is going to be displayed only through the routing system. I tend to add it anyway so that I can apply the component using an `HTML` element as well.

Listing 25-3. The Contents of the `productCount.component.ts` File in the `src/app/core` Folder

```

import {
    Component, KeyValueDiffer, KeyValueDiffers, ChangeDetectorRef
} from "@angular/core";
import { Model } from "../model/repository.model";

@Component({
    selector: "paProductCount",
    template: `<div class="bg-info text-white p-2">There are
        {{count}} products
    </div>`
})
export class ProductCountComponent {
    private differ?: KeyValueDiffer<any>;
    count: number = 0;

    constructor(private model: Model,
        private keyValueDiffers: KeyValueDiffers,
        private changeDetector: ChangeDetectorRef) { }

    ngOnInit() {
        this.differ = this.keyValueDiffers
            .find(this.model.getProducts())
            .create();
    }

    ngDoCheck() {
        if (this.differ?.diff(this.model.getProducts()) != null) {
            this.updateCount();
        }
    }
}

```

```
    private updateCount() {
        this.count = this.model.getProducts().length;
    }
}
```

This component uses an inline template to display the number of products in the data model, which is updated when the data model changes. Next, I added a file called `categoryCount.component.ts` in the `src/app/core` folder and defined the component shown in Listing 25-4.

Listing 25-4. The Contents of the categoryCount.component.ts File in the src/app/core Folder

```
import { Component, KeyValueDiffer, KeyValueDiffers, ChangeDetectorRef } from "@angular/core";
import { Model } from "../model/repository.model";

@Component({
  selector: "paCategoryCount",
  template: `<div class="bg-primary p-2 text-white">
    There are {{count}} categories
  </div>`
})
export class CategoryCountComponent {
  private differ?: KeyValueDiffer<any, any>;
  count: number = 0;

  constructor(private model: Model,
    private keyValueDiffers: KeyValueDiffers,
    private changeDetector: ChangeDetectorRef) { }

  ngOnInit() {
    this.differ = this.keyValueDiffers
      .find(this.model.getProducts())
      .create();
  }

  ngDoCheck() {
    if (this.differ?.diff(this.model.getProducts()) != null) {
      this.count = this.model.getProducts()
        .map(p => p.category)
        .filter((category, index, array) => array.indexOf(category) == index)
        .length;
    }
  }
}
```

This component uses a differ to track changes in the data model and count the number of unique categories, which is displayed using a simple inline template. For the final component, I added a file called `notFound.component.ts` in the `src/app/core` folder and used it to define the component shown in Listing 25-5.

Listing 25-5. The notFound.component.ts File in the src/app/core Folder

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

@Component({
  selector: "paNotFound",
  template: `<h3 class="bg-danger text-white p-2">Sorry, something went wrong</h3>
              <button class="btn btn-primary" routerLink="/">Start Over</button>`
})
export class NotFoundComponent {}
```

This component displays a static message that will be shown when something goes wrong with the routing system. Listing 25-6 adds the new components to the core module.

Listing 25-6. Declaring Components in the core.module.ts File in the src/app/core Folder

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "./table.component";
import { FormComponent } from "./form.component";
//import { SharedState } from "./sharedState.service";
import { ValidationHelper } from "./validation_helper";
import { ValidationErrorsDirective } from "./validationErrors.directive";
import { HilowValidatorDirective } from "../validation/hilow";
import { RouterModule } from "@angular/router";
import { ProductCountComponent } from "./productCount.component";
import { CategoryCountComponent } from "./categoryCount.component";
import { NotFoundComponent } from "./notFound.component";

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule, ReactiveFormsModule,
    RouterModule],
  declarations: [TableComponent, FormComponent, ValidationHelper,
    ValidationErrorsDirective, HilowValidatorDirective, ProductCountComponent,
    CategoryCountComponent, NotFoundComponent],
  exports: [ModelModule, TableComponent, FormComponent],
})
export class CoreModule {}
```

Open a new command prompt, navigate to the exampleApp folder, and run the following command to start the server that provides the RESTful web server:

```
npm run json
```

Open a separate command prompt, navigate to the exampleApp folder, and run the following command to start the Angular development tools:

```
ng serve
```

Open a new browser window and navigate to `http://localhost:4200` to see the content shown in Figure 25-1.

ID	Name	Category	Price	Details	
1	Kayak	Watersports	\$275.00	Acme, boat	<button>Delete</button> <button>Edit</button>
2	Lifejacket	Watersports	\$48.95	Smoot Co, safety	<button>Delete</button> <button>Edit</button>
3	Soccer Ball	Soccer	\$19.50	(None)	<button>Delete</button> <button>Edit</button>
4	Corner Flags	Soccer	\$34.95	(None)	<button>Delete</button> <button>Edit</button>
5	Stadium	Soccer	\$79,500.00	(None)	<button>Delete</button> <button>Edit</button>
6	Thinking Cap	Chess	\$16.00	(None)	<button>Delete</button> <button>Edit</button>
7	Unsteady Chair	Chess	\$29.95	(None)	<button>Delete</button> <button>Edit</button>
8	Human Chess Board	Chess	\$75.00	(None)	<button>Delete</button> <button>Edit</button>
9	Bling Bling King	Chess	\$1,200.00	(None)	<button>Delete</button> <button>Edit</button>

[Create New Product](#) [Generate HTTP Error](#)

Figure 25-1. Running the example application

Using Wildcards and Redirections

The routing configuration in an application can quickly become complex and contain redundancies and oddities to cater to the structure of an application. Angular provides two useful tools that can help simplify routes and also deal with problems when they arise, as described in the following sections.

Using Wildcards in Routes

The Angular routing system supports a special path, denoted by two asterisks (the `**` characters), that allows routes to match any URL. The basic use of the wildcard path is to deal with navigation that would otherwise create a routing error. Listing 25-7 adds a button to the table component's template that navigates to a route that hasn't been defined by the application's routing configuration.

Listing 25-7. Adding a Button in the table.component.html File in the src/app/core Folder

```
<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr>
      <th>ID</th><th>Name</th><th>Category</th><th>Price</th>
      <th>Details</th><th></th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of getProducts()">
      <td>{{item.id}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price | currency:"USD" }}</td>
      <td>
        <ng-container *ngIf="item.details else empty">
          {{ item.details?.supplier }}, {{ item.details?.keywords}}
        </ng-container>
        <ng-template #empty>(None)</ng-template>
      </td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm m-1"
               (click)="deleteProduct(item.id)">
          Delete
        </button>
        <button class="btn btn-warning btn-sm"
               [routerLink]=['/form', 'edit', item.id]">
          Edit
        </button>
      </td>
    </tr>
  </tbody>
</table>
<button class="btn btn-primary m-1" routerLink="/form/create">
  Create New Product
</button>
<button class="btn btn-danger m-1" (click)="deleteProduct(-1)">
  Generate HTTP Error
</button>
<b><button class="btn btn-danger m-1" routerLink="/does/not/exist">
  Generate Routing Error
</button></b>
```

Clicking the button will ask the application to navigate to the URL /does/not/exist, for which there is no route configured. When a URL doesn't match a URL, an error is thrown, which is then picked up and processed by the error handling class, which leads to a warning being displayed by the message component, as shown in Figure 25-2.

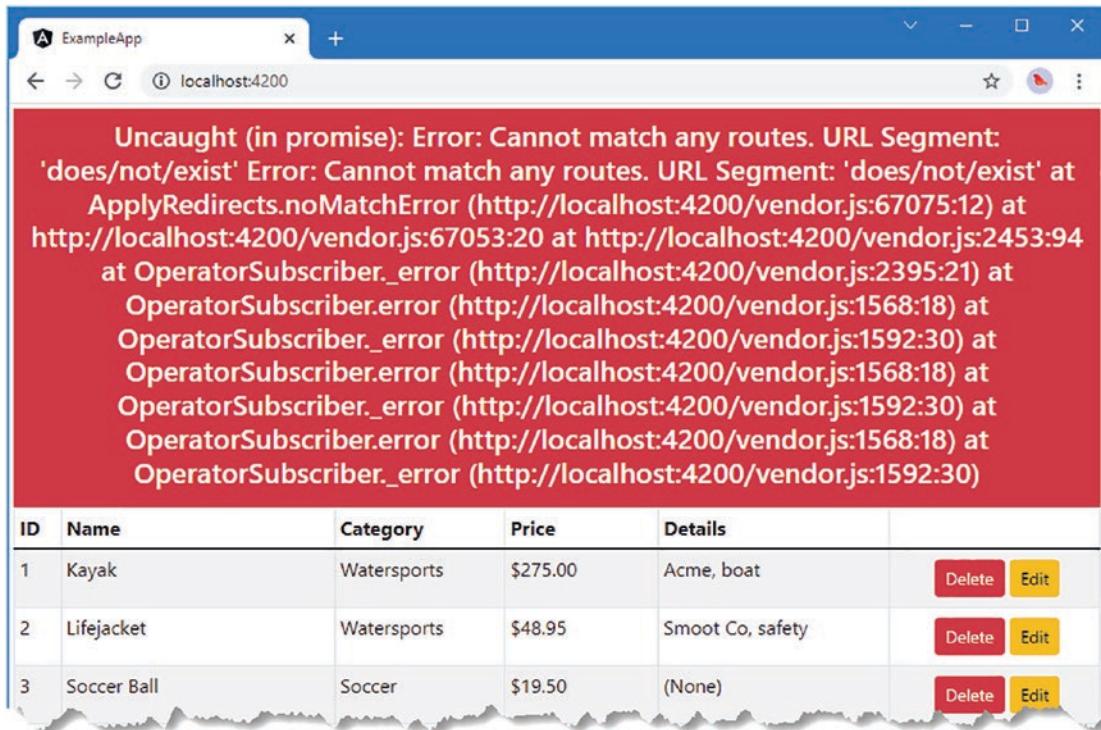


Figure 25-2. The default navigation error

This isn't a useful way to deal with an unknown route because the user won't know what routes are and may not realize that the application was trying to navigate to the problem URL.

A better approach is to use the wildcard route to handle navigation for URLs that have not been defined and select a component that will present a more useful message to the user, as illustrated in Listing 25-8.

Listing 25-8. Adding a Wildcard Route in the app.routing.ts File in the src/app Folder

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";

const routes: Routes = [
  { path: "form/:mode/:id", component: FormComponent },
  { path: "form/:mode", component: FormComponent },
  { path: "", component: TableComponent },
  { path: "**", component: NotFoundComponent }
]

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

The new route in the listing uses the wildcard to select the `NotFoundComponent`, which displays the message shown in Figure 25-3 when the Generate Routing Error button is clicked.

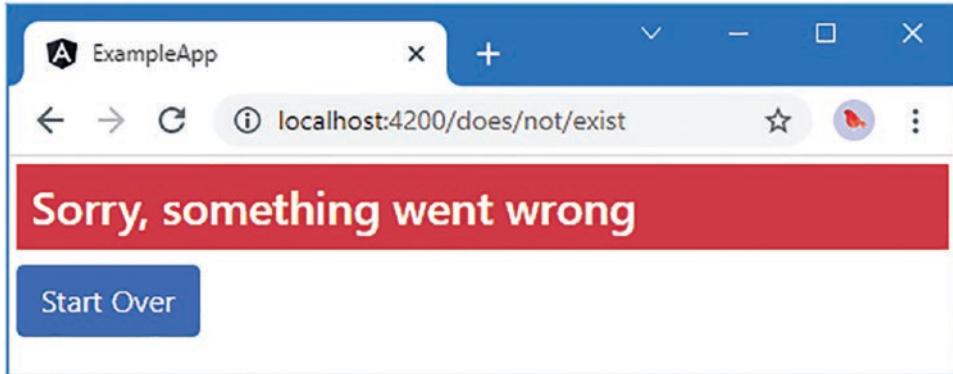


Figure 25-3. Using a wildcard route

Clicking the Start Over button navigates to the / URL, which will select the table component for display.

Using Redirections in Routes

Routes do not have to select components; they can also be used as aliases that redirect the browser to a different URL. Redirections are defined using the `redirectTo` property in a route, as shown in Listing 25-9.

Listing 25-9. Using Route Redirection in the app.routing.ts File in the src/app Folder

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";

const routes: Routes = [
  { path: "form/:mode/:id", component: FormComponent },
  { path: "form/:mode", component: FormComponent },
  { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
  { path: "table", component: TableComponent },
  { path: "", redirectTo: "/table", pathMatch: "full" },
  { path: "**", component: NotFoundComponent }
]

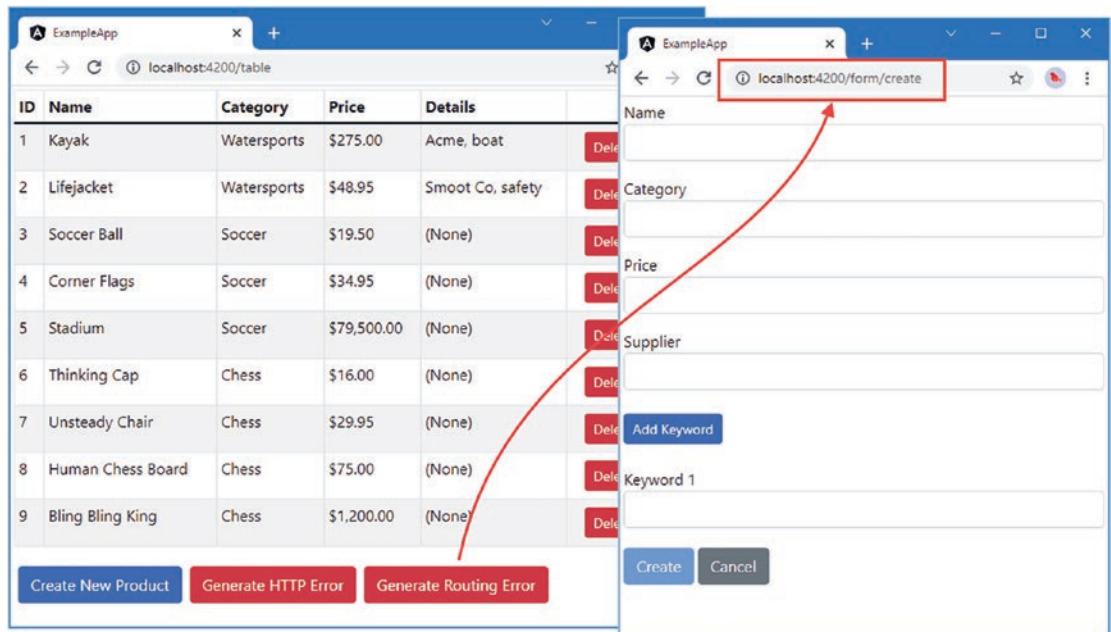
export const routing = RouterModule.forRoot(routes);
```

The `redirectTo` property is used to specify the URL that the browser will be redirected to. When defining redirections, the `pathMatch` property must also be specified, using one of the values described in Table 25-2.

Table 25-2. The pathMatch Values

Name	Description
prefix	This value configures the route so that it matches URLs that start with the specified path, ignoring any subsequent segments.
full	This value configures the route so that it matches only the URL specified by the path property.

The first route added in Listing 25-9 specifies a pathMatch value of prefix and a path of does, which means it will match any URL whose first segment is does, such as the /does/not/exist URL that is navigated to by the Generate Routing Error button. When the browser navigates to a URL that has this prefix, the routing system will redirect it to the /form/create URL, as shown in Figure 25-4.

**Figure 25-4.** Performing a route redirection

The other routes in Listing 25-9 redirect the empty path to the /table URL, which displays the table component. This is a common technique that makes the URL schema more obvious because it matches the default URL (`http://localhost:4200/`) and redirects it to something more meaningful and memorable to the user (`http://localhost:4200/table`). In this case, the pathMatch property value is full, although this has no effect since it has been applied to the empty path.

Navigating Within a Component

The examples in the previous chapter navigated between different components so that clicking a button in the table component navigates to the form component and vice versa.

This isn't the only kind of navigation that's possible; you can also navigate within a component. To demonstrate, Listing 25-10 adds buttons to the form component that allow the user to edit the previous or next data objects.

Listing 25-10. Adding Buttons to the form.component.html File in the src/app/core Folder

```
<div *ngIf="editing" class="p-2">
  <button class="btn btn-secondary m-1"
    [routerLink]=["/form", 'edit',
      model.getPreviousProductId(product.id) | async"]>
    Previous
  </button>
  <button class="btn btn-secondary"
    [routerLink]=["/form", 'edit',
      model.getNextProductId(product.id) | async"]>
    Next
  </button>
</div>

<form [formGroup]="productForm" #form="ngForm"
  (ngSubmit)="submitForm()" (reset)="resetForm()">
  <!-- ...elements omitted for brevity... -->
</form>
```

These buttons have bindings for the routerLink directive with expressions that target the previous and next objects in the data model, using the `async` pipe to get results from the observables returned added to the repository at the start of the chapter. This means that if you click the Edit button in the table for the lifejacket, for example, the Next button will navigate to the URL that edits the soccer ball, and the Previous button will navigate to the URL for the kayak.

Responding to Ongoing Routing Changes

Although the URL changes when the Previous and Next buttons are clicked, there is no change in the data displayed to the user. Angular tries to be efficient during navigation, and it knows that the URLs that the Previous and Next buttons navigate to are handled by the same component that is currently displayed to the user. Rather than create a new instance of the component, it simply tells the component that the selected route has changed.

This is a problem because the form component isn't set up to receive change notifications. Its constructor receives the `ActivatedRoute` object that Angular uses to provide details of the current route, but only its `snapshot` property is used. The component's constructor has long been executed by the time that Angular updates the values in the `ActivatedRoute` object, which means that it misses the notification. This worked when the configuration of the application meant that a new form component would be created each time the user wanted to create or edit a product, but it is no longer sufficient.

Fortunately, the `ActivatedRoute` class defines a set of properties allowing interested parties to receive notifications through Reactive Extensions `Observable` objects. These properties correspond to the ones provided by the `ActivatedRouteSnapshot` object returned by the `snapshot` property but send new events when there are any subsequent changes, as described in Table 25-3.

Table 25-3. The Observable Properties of the ActivatedRoute Class

Name	Description
url	This property returns an Observable<UrlSegment[]>, which provides the set of URL segments each time the route changes.
params	This property returns an Observable<Params>, which provides the URL parameters each time the route changes.
queryParams	This property returns an Observable<Params>, which provides the URL query parameters each time the route changes.
fragment	This property returns an Observable<string>, which provides the URL fragment each time the route changes.

These properties can be used by components that need to handle navigation changes that don't result in a different component being displayed to the user, as shown in Listing 25-11.

Tip If you need to combine different data elements from the route, such as using both segments and parameters, then subscribe to the Observer for one data element and use the snapshot property to get the rest of the data you require.

Listing 25-11. Observing Route Changes in the form.component.ts File in the src/app/core Folder

```
...
constructor(public model: Model, activeRoute: ActivatedRoute,
    public router: Router) {

    activeRoute.params.subscribe(params => {
        this.editing = params["mode"] == "edit";
        let id = params["id"];
        if (id != null) {
            model.getProductObservable(id).subscribe(p => {
                Object.assign(this.product, p || new Product());
                this.productForm.patchValue(this.product);
            });
        }
    })
}
...
}
```

The component subscribes to the `Observer<Params>` that sends a new `Params` object to subscribers each time the active route changes. The `Observer` objects returned by the `ActivatedRoute` properties send details of the most recent route change when the `subscribe` method is called, ensuring that the component's constructor doesn't miss the initial navigation that led to it being called.

The result is that the component can react to route changes that don't cause Angular to create a new component, meaning that clicking the Next or Previous button changes the product that has been selected for editing, as shown in Figure 25-5.

Tip The effect of navigation is obvious when the activated route changes the component that is displayed to the user. It may not be so obvious when just the data changes. To help emphasize changes, Angular can apply animations that draw attention to the effects of navigation. See Chapter 27 for details.

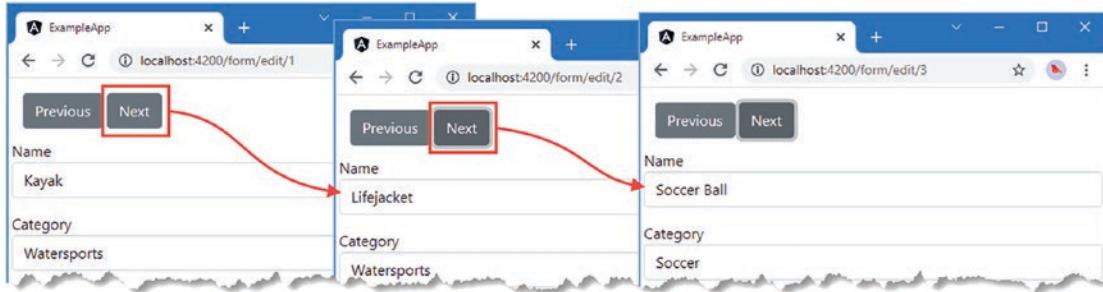


Figure 25-5. Responding to route changes

Styling Links for Active Routes

A common use for the routing system is to display multiple navigation elements alongside the content that they select. To demonstrate, Listing 25-12 adds a new route to the application that will allow the table component to be targeted with a URL that contains a category filter.

Listing 25-12. Defining a Route in the app.routing.ts File in the src/app Folder

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";

const routes: Routes = [
  { path: "form/:mode/:id", component: FormComponent },
  { path: "form/:mode", component: FormComponent },
  { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
  { path: "table/:category", component: TableComponent },
  { path: "table", component: TableComponent },
  { path: "", redirectTo: "/table", pathMatch: "full" },
  { path: "**", component: NotFoundComponent }
]

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

Listing 25-13 updates the TableComponent class so that it uses the routing system to get details of the active route and assigns the value of the category route parameter to a category property that can be accessed in the template. The category property is used in the getProducts method to filter the objects in the data model.

Listing 25-13. Adding Category Filter Support in the table.component.ts File in the src/app/core Folder

```

import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { ActivatedRoute } from "@angular/router";

@Component({
  selector: "paTable",
  templateUrl: "table.component.html"
})
export class TableComponent {
  category: string | null = null;

  constructor(public model: Model, activeRoute: ActivatedRoute) {
    activeRoute.params.subscribe(params => {
      this.category = params["category"] || null;
    })
  }

  getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  getProducts(): Product[] {
    return this.model.getProducts()
      .filter(p => this.category == null || p.category == this.category);
  }

  get categories(): (string) [] {
    return (this.model.getProducts()
      .map(p => p.category)
      .filter((c, index, array) => c != undefined
        && array.indexOf(c) == index)) as string[];
  }

  deleteProduct(key?: number) {
    if (key != undefined) {
      this.model.deleteProduct(key);
    }
  }
}

```

There is also a new `categories` property that will be used in the template to generate the set of categories for filtering. The final step is to add the HTML elements to the template that will allow the user to apply a filter, as shown in Listing 25-14.

Listing 25-14. Adding Filter Elements in the table.component.html File in the src/app/core Folder

```

<div class="container-fluid">
  <div class="row">
    <div class="col-auto">
      <div class="d-grid gap-2">
        <button class="btn btn-secondary"
               routerLink="/" routerLinkActive="bg-primary">
          All
        </button>
        <button *ngFor="let category of categories"
               class="btn btn-secondary"
               [routerLink]="/[ 'table', category ]"
               routerLinkActive="bg-primary">
          {{category}}
        </button>
      </div>
    </div>
    <div class="col">
      <table class="table table-sm table-bordered table-striped">
        <thead>
          <tr>
            <th>ID</th><th>Name</th><th>Category</th><th>Price</th>
            <th>Details</th><th></th>
          </tr>
        </thead>
        <tbody>
          <tr *ngFor="let item of getProducts()">
            <td>{{item.id}}</td>
            <td>{{item.name}}</td>
            <td>{{item.category}}</td>
            <td>{{item.price | currency:"USD" }}</td>
            <td>
              <ng-container *ngIf="item.details else empty">
                {{ item.details?.supplier }}, {{ item.details?.keywords }}
              </ng-container>
              <ng-template #empty>(None)</ng-template>
            </td>
            <td class="text-center">
              <button class="btn btn-danger btn-sm m-1"
                     (click)="deleteProduct(item.id)">
                Delete
              </button>
              <button class="btn btn-warning btn-sm"
                     [routerLink]="/[ 'form', 'edit', item.id ]">
                Edit
              </button>
            </td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>

```

```

</div>
</div>
<div class="p-2 text-center">
  <button class="btn btn-primary m-1" routerLink="/form/create">
    Create New Product
  </button>
  <button class="btn btn-danger m-1" (click)="deleteProduct(-1)">
    Generate HTTP Error
  </button>
  <button class="btn btn-danger m-1" routerLink="/does/not/exist">
    Generate Routing Error
  </button>
</div>

```

The important part of this example is the use of the `routerLinkActive` attribute, which is used to specify a CSS class that the element will be assigned to when the URL specified by the `routerLink` attribute matches the active route.

The listing specifies a class called `bg-primary`, which changes the appearance of the button and makes the selected category more obvious. When combined with the functionality added to the component in Listing 25-13, the result is a set of buttons allowing the user to view products in a single category, as shown in Figure 25-6.

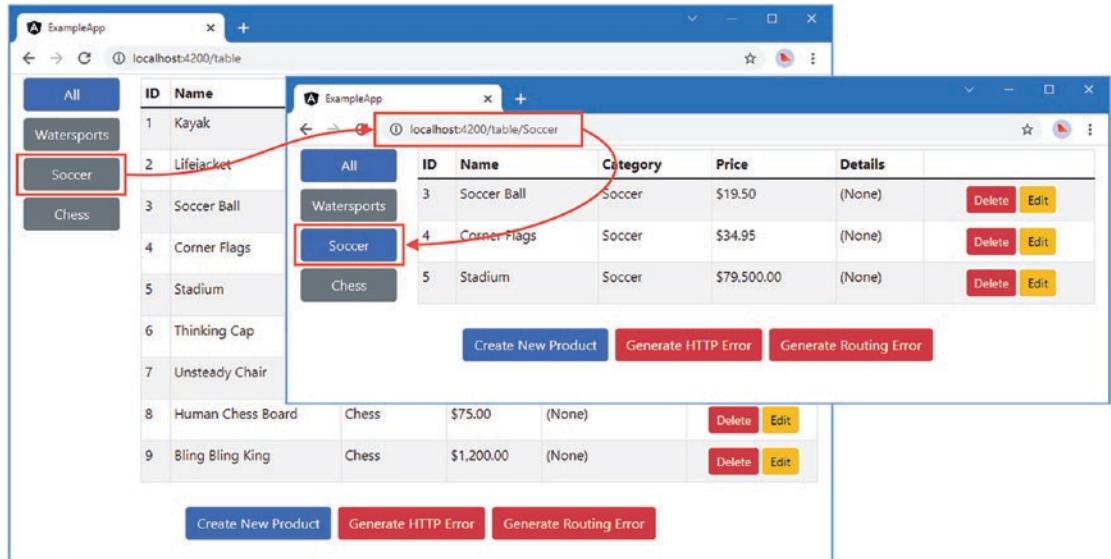


Figure 25-6. Filtering products

If you click the Soccer button, the application will navigate to the `/table/Soccer` URL, and the table will display only those products in the Soccer category. The Soccer button will also be highlighted since the `routerLinkActive` attribute means that Angular will add the `button` element to the Bootstrap `bg-primary` class.

Fixing the All Button

The navigation buttons reveal a common problem, which is that the All button is always added to the active class, even when the user has filtered the table to show a specific category.

This happens because the `routerLinkActive` attribute performs partial matches on the active URL by default. In the case of the example, the `/` URL will always cause the All button to be activated because it is at the start of all URLs. This problem can be fixed by configuring the `routerLinkActive` directive, as shown in Listing 25-15.

Listing 25-15. Configuring the Directive in the `table.component.html` File in the `src/app/core` Folder

```
...
<div class="d-grid gap-2">
  <button class="btn btn-secondary"
    routerLink="/table" routerLinkActive="bg-primary"
    [routerLinkActiveOptions]="{exact: true}">
    All
  </button>
  <button *ngFor="let category of categories"
    class="btn btn-secondary"
    [routerLink]=["/table", category]
    routerLinkActive="bg-primary">
    {{category}}
  </button>
</div>
...

```

The configuration is performed using a binding on the `routerLinkActiveOptions` attribute, which accepts a literal object. The `exact` property is the only available configuration setting and is used to control matching the active route URL. Setting this property to `true` will add the element to the class specified by the `routerLinkActive` attribute only when there is an exact match with the active route's URL, which is changed to `/table`. With this change, the All button will be highlighted only when all of the products are shown, as illustrated by Figure 25-7.

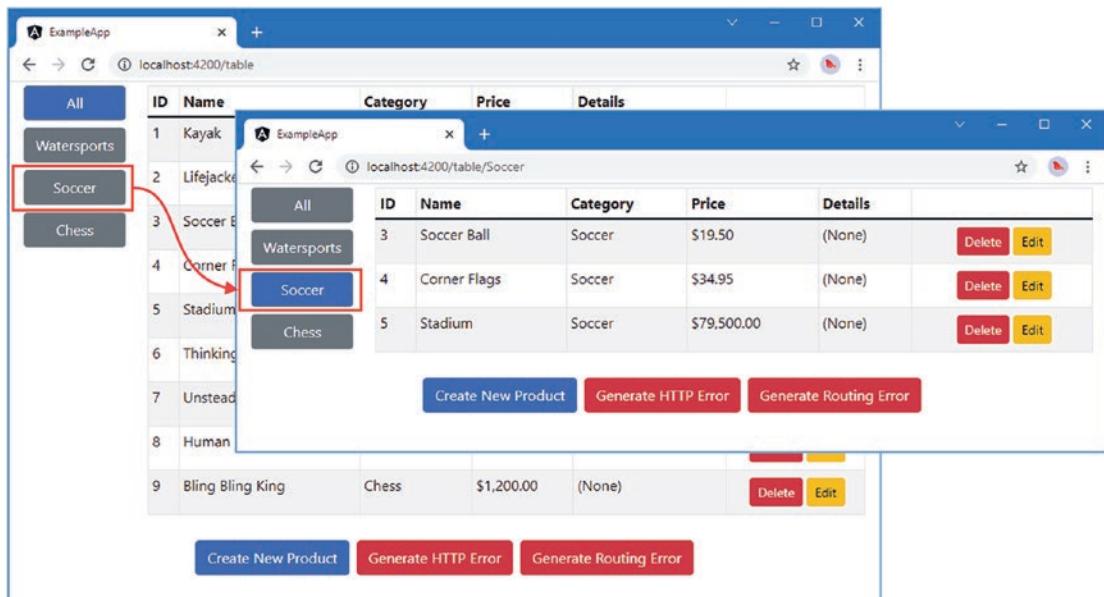


Figure 25-7. Fixing the All button problem

Creating Child Routes

Child routes allow components to respond to part of the URL by embedding router-outlet elements in their templates, creating more complex arrangements of content. I am going to use the simple components I created at the start of the chapter to demonstrate how child routes work. These components will be displayed above the product table, and the component that is shown will be specified in the URLs shown in Table 25-4.

Table 25-4. The URLs and the Components They Will Select

URL	Component
/table/products	The ProductCountComponent will be displayed.
/table/categories	The CategoryCountComponent will be displayed.
/table	Neither component will be displayed.

Listing 25-16 shows the changes to the application's routing configuration to implement the routing strategy in the table.

Listing 25-16. Configuring Routes in the app.routing.ts File in the src/app Folder

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";

const routes: Routes = [
  { path: "form/:mode/:id", component: FormComponent },
  { path: "form/:mode", component: FormComponent },
  { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
  {
    path: "table",
    component: TableComponent,
    children: [
      { path: "products", component: ProductCountComponent },
      { path: "categories", component: CategoryCountComponent }
    ]
  },
  { path: "table/:category", component: TableComponent },
  { path: "table", component: TableComponent },
  { path: "", redirectTo: "/table", pathMatch: "full" },
  { path: "**", component: NotFoundComponent }
]

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

Child routes are defined using the `children` property, which is set to an array of routes defined in the same way as the top-level routes. When Angular uses the entire URL to match a route that has children, there

will be a match only if the URL to which the browser navigates contains segments that match both the top-level segment and the segments specified by one of the child routes.

Tip Notice that I have added the new route before the one whose path is `table/:category`. Angular tries to match routes in the order in which they are defined. The `table/:category` path would match both the `/table/products` and `/table/categories` URLs and lead the `table` component to filter the products for nonexistent categories. By placing the more specific route first, the `/table/products` and `/table/categories` URLs will be matched before the `table/:category` path is considered.

Creating the Child Route Outlet

The components selected by child routes are displayed in a `router-outlet` element defined in the template of the component selected by the parent route. In the case of the example, this means the child routes will target an element in the `table` component's template, as shown in Listing 25-17, which also adds elements that will navigate to the new routes.

Listing 25-17. Adding an Outlet in the `table.component.html` File in the `src/app/core` Folder

```
<div class="container-fluid">
  <div class="row">
    <div class="col-auto">
      <div class="d-grid gap-2">
        <button class="btn btn-secondary"
          routerLink="/table" routerLinkActive="bg-primary"
          [routerLinkActiveOptions]="{exact: true}">
          All
        </button>
        <button *ngFor="let category of categories"
          class="btn btn-secondary"
          [routerLink]=["'/table', category]"
          routerLinkActive="bg-primary">
          {{category}}
        </button>
      </div>
    </div>
    <div class="col">

      <button class="btn btn-info mx-1" routerLink="/table/products">
        Count Products
      </button>
      <button class="btn btn-primary mx-1" routerLink="/table/categories">
        Count Categories
      </button>
      <button class="btn btn-secondary mx-1" routerLink="/table">
        Count Neither
      </button>

    </div>
  </div>
</div>
```

```

<div class="my-2">
  <router-outlet></router-outlet>
</div>

<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr>
      <th>ID</th><th>Name</th><th>Category</th><th>Price</th>
      <th>Details</th><th></th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of getProducts()">
      <td>{{item.id}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price | currency:"USD" }}</td>
      <td>
        <ng-container *ngIf="item.details else empty">
          {{ item.details?.supplier }}, {{ item.details?.keywords }}
        </ng-container>
        <ng-template #empty>(None)</ng-template>
      </td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm m-1"
               (click)="deleteProduct(item.id)">
          Delete
        </button>
        <button class="btn btn-warning btn-sm"
               [routerLink]=["/form", 'edit', item.id]">
          Edit
        </button>
      </td>
    </tr>
  </tbody>
</table>
</div>
</div>
<div class="p-2 text-center">
  <button class="btn btn-primary m-1" routerLink="/form/create">
    Create New Product
  </button>
  <button class="btn btn-danger m-1" (click)="deleteProduct(-1)">
    Generate HTTP Error
  </button>
  <button class="btn btn-danger m-1" routerLink="/does/not/exist">
    Generate Routing Error
  </button>
</div>

```

The button elements have `routerLink` attributes that specify the URLs listed in Table 25-4, and there is also a `router-outlet` element, which will be used to display the selected component, as shown in Figure 25-8, or no component if the browser navigates to the `/table` URL.

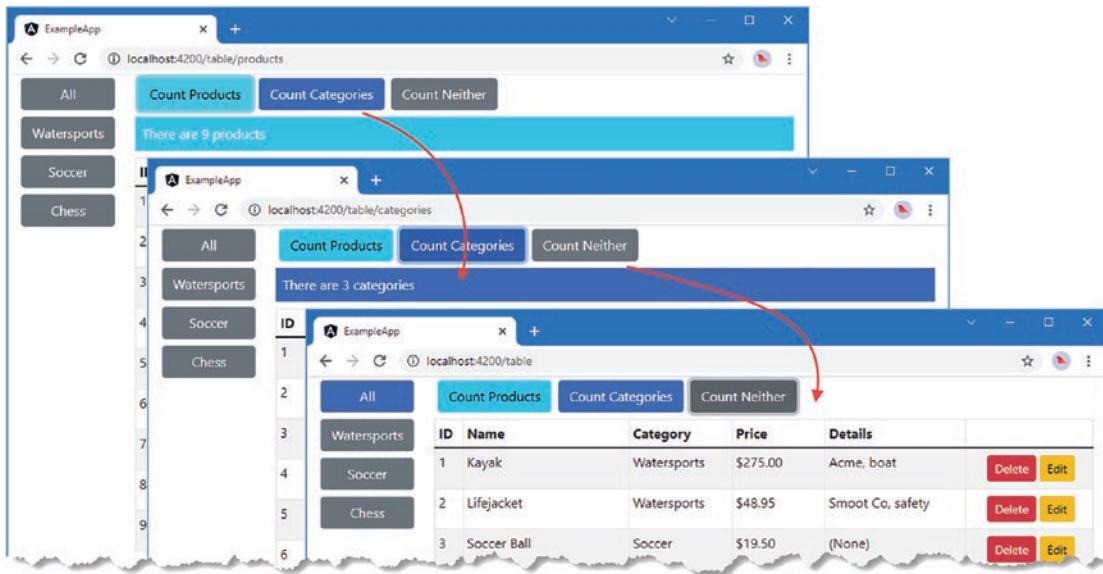


Figure 25-8. Using child routes

Accessing Parameters from Child Routes

Child routes can use all the features available to the top-level routes, including defining route parameters and even having their own child routes. Route parameters are worth special attention in child routes because of the way that Angular isolates children from their parents. For this section, I am going to add support for the URLs described in Table 25-5.

Table 25-5. The New URLs Supported by the Example Application

Name	Description
<code>/table/:category/products</code>	This route will filter the contents of the table and select the <code>ProductCountComponent</code> .
<code>/table/:category/categories</code>	This route will filter the contents of the table and select the <code>CategoryCountComponent</code> .

Listing 25-18 defines the routes that support the URLs shown in the table.

Listing 25-18. Adding Routes in the `app.routing.ts` File in the `src/app` Folder

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
```

```

import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";

const childRoutes: Routes = [
  { path: "products", component: ProductCountComponent },
  { path: "categories", component: CategoryCountComponent },
  { path: "", component: ProductCountComponent }
];

const routes: Routes = [
  { path: "form/:mode/:id", component: FormComponent },
  { path: "form/:mode", component: FormComponent },
  { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
  // {
  //   path: "table",
  //   component: TableComponent,
  //   children: [
  //     { path: "products", component: ProductCountComponent },
  //     { path: "categories", component: CategoryCountComponent }
  //   ]
  // },
  // { path: "table/:category", component: TableComponent },
  // { path: "table", component: TableComponent },
  { path: "table", component: TableComponent, children: childRoutes },
  { path: "table/:category", component: TableComponent, children: childRoutes },
  { path: "", redirectTo: "/table", pathMatch: "full" },
  { path: "**", component: NotFoundComponent }
]

export const routing = RouterModule.forRoot(routes);

```

The type of the `children` property is a `Routes` object, which makes it easy to minimize duplication in the route configuration when you need to apply the same set of child routes in different parts of the URL schema. In the listing, I have defined the child routes in a `Routes` object called `childRoutes` and used it as the value for the `children` property in two different top-level routes.

To make it possible to target these new routes, Listing 25-19 changes the targets of the buttons that appear above the table so they navigate relative to the current URL. I have removed the Count Neither button since the `ProductCountComponent` will be shown when the empty path child route matches the URL.

Listing 25-19. Using Relative URLs in the `table.component.html` File in the `src/app/core` Folder

```

...
<div class="col">
  <button class="btn btn-info mx-1" routerLink="products">
    Count Products
  </button>
  <button class="btn btn-primary mx-1" routerLink="categories">
    Count Categories
  </button>
</div>

```

```

</button>
<button class="btn btn-secondary mx-1" routerLink="/table">
  Count Neither
</button>
<div class="my-2">
  <router-outlet></router-outlet>
</div>

<table class="table table-sm table-bordered table-striped">
...

```

When Angular matches routes, the information it provides to the components that are selected through the `ActivatedRoute` object is segregated so that each component receives details of only the part of the route that selected it.

In the case of the routes added in Listing 25-19, this means the `ProductCountComponent` and `CategoryCountComponent` receive an `ActivatedRoute` object that describes only the child route that selected them, with the single segment of `/products` or `/categories`. Equally, the `TableComponent` component receives an `ActivatedRoute` object that doesn't contain the segment that was used to match the child route.

Fortunately, the `ActivatedRoute` class provides some properties that offer access to the rest of the route, allowing parents and children to access the rest of the routing information, as described in Table 25-6.

Table 25-6. The `ActivatedRoute` Properties for Child-Parent Route Information

Name	Description
<code>pathFromRoot</code>	This property returns an array of <code>ActivatedRoute</code> objects representing all the routes used to match the current URL.
<code>parent</code>	This property returns an <code>ActivatedRoute</code> representing the parent of the route that selected the component.
<code>firstChild</code>	This property returns an <code>ActivatedRoute</code> representing the first child route used to match the current URL.
<code>children</code>	This property returns an array of <code>ActivatedRoute</code> objects representing all the child routes used to match the current URL.

Listing 25-20 shows how the `ProductCountComponent` component can access the wider set of routes used to match the current URL to get a value for the category route parameter and adapt its output when the contents of the table are filtered for a single category.

Listing 25-20. Ancestor Routes in the `productCount.component.ts` File in the `src/app/core` Folder

```

import {
  Component, KeyValueDiffer, KeyValueDiffers, ChangeDetectorRef
} from "@angular/core";
import { Model } from "../model/repository.model";
import { ActivatedRoute } from "@angular/router";

@Component({
  selector: "paProductCount",

```

```

template: `<div class="bg-info text-white p-2">There are
            {{count}} products
        </div>`
})
export class ProductCountComponent {
    private differ?: KeyValueDiffer<any, any>;
    count: number = 0;
    private category?: string;

    constructor(private model: Model,
                private keyValueDiffers: KeyValueDiffers,
                private changeDetector: ChangeDetectorRef,
                activeRoute: ActivatedRoute) {
        activeRoute.pathFromRoot.forEach(route => route.params.subscribe(params => {
            if (params["category"] != null) {
                this.category = params["category"];
                this.updateCount();
            }
        }))
    }

    ngOnInit() {
        this.differ = this.keyValueDiffers
            .find(this.model.getProducts())
            .create();
    }

    ngDoCheck() {
        if (this.differ?.diff(this.model.getProducts()) != null) {
            this.updateCount();
        }
    }

    private updateCount() {
        this.count = this.model.getProducts()
        .filter(p => this.category == null || p.category == this.category)
        .length;
    }
}

```

The `pathFromRoot` property is especially useful because it allows a component to inspect all the routes that have been used to match the URL. Angular minimizes the routing updates required to handle navigation, which means that a component that has been selected by a child route won't receive a change notification through its `ActivatedRoute` object if only its parent has changed. It is for this reason that I have subscribed to updates from all the `ActivatedRoute` objects returned by the `pathFromRoot` property, ensuring that the component will always detect changes in the value of the `category` route parameter.

To see the result, save the changes, click the Watersports button to filter the contents of the table, and then click the Count Products button, which selects the `ProductCountComponent`. This number of products reported by the component will correspond to the number of rows in the table, as shown in Figure 25-9.

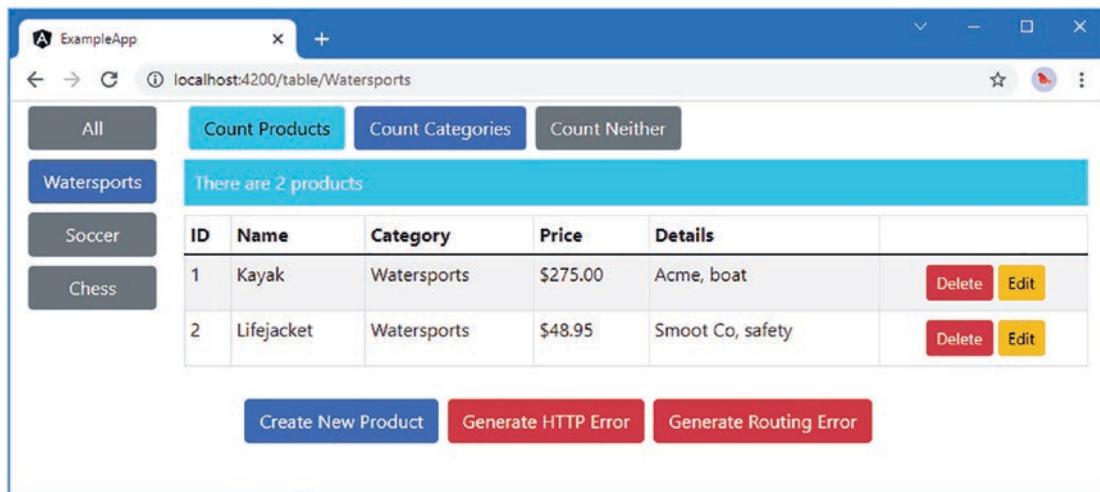


Figure 25-9. Accessing the other routes used to match a URL

Summary

In this chapter, I continued to describe the features provided by the Angular URL routing system, going beyond the basic features described in the previous chapter. I explained how to create wildcard and redirection routes, how to create routes that navigate relative to the current URL, and how to create child routes to display nested components. In the next chapter, I finish describing the URL routing system, focusing on the most advanced features.

CHAPTER 26



Routing and Navigation: Part 3

In this chapter, I continue to describe the Angular URL routing system, focusing on the most advanced features. I explain how to control route activation, how to load feature modules dynamically, and how to use multiple outlet elements in a template. Table 26-1 summarizes the chapter.

Table 26-1. Chapter Summary

Problem	Solution	Listing
Delaying navigation until a task is complete	Use a route resolver	1–6
Preventing route activation	Use an activation guard	7–13
Preventing the user from navigating away from the current content	Use a deactivation guard	14–18
Deferring loading a feature module until it is required	Create a dynamically loaded module	19–24
Controlling when a dynamically loaded module is used	Use a loading guard	25–27
Using routing to manage multiple router outlets	Use named outlets in the same template	28–33

Preparing the Example Project

For this chapter, I will continue using the exampleApp project that was created in Chapter 20 and has been modified in each subsequent chapter. No changes are required for this chapter.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Open a new command prompt, navigate to the exampleApp folder, and run the following command to start the server that provides the RESTful web server:

```
npm run json
```

Open a separate command prompt, navigate to the exampleApp folder, and run the following command to start the Angular development tools:

```
ng serve
```

Open a new browser window and navigate to `http://localhost:4200/table` to see the content shown in Figure 26-1.

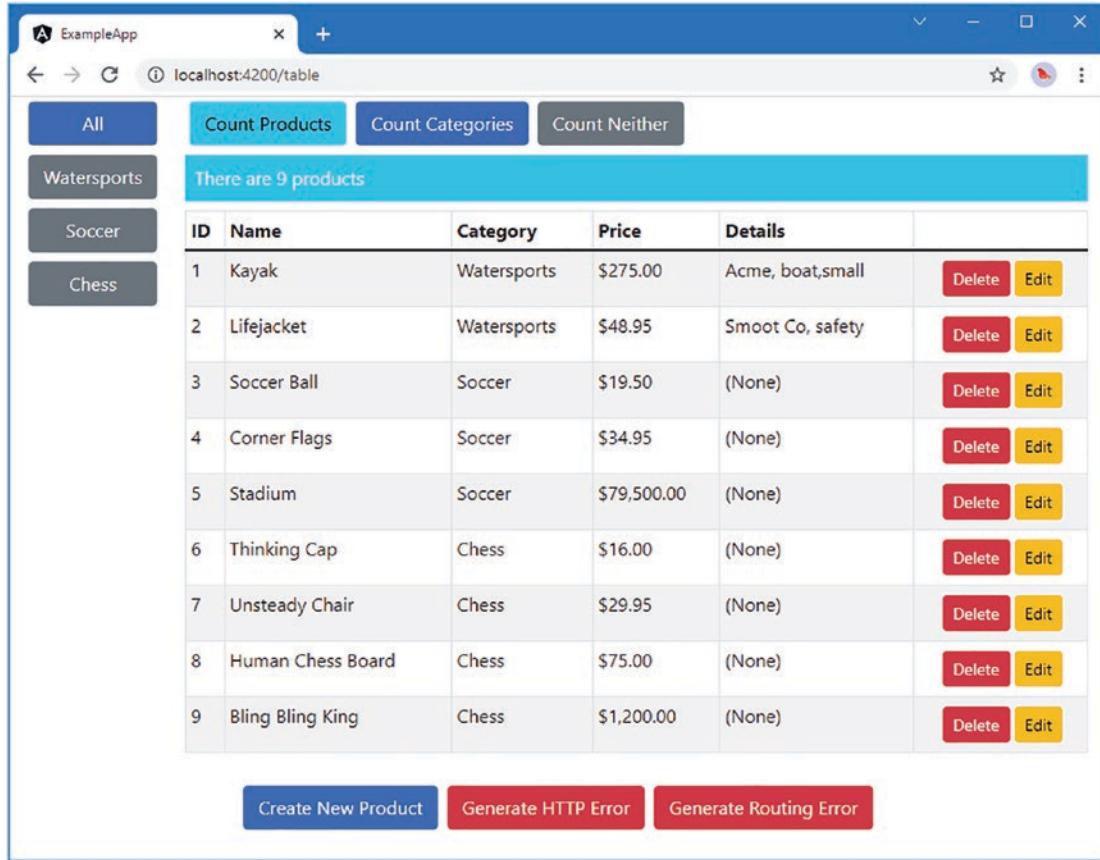


Figure 26-1. Running the example application

Guarding Routes

At the moment, the user can navigate anywhere in the application at any time. This isn't always a good idea, either because some parts of the application may not always be ready or because some parts of the application are restricted until specific actions are performed. To control the use of navigation, Angular supports *guards*, which are specified as part of the route configuration using the properties defined by the `Routes` class, described in Table 26-2.

Table 26-2. The Routes Properties for Guards

Name	Description
resolve	This property is used to specify guards that will delay route activation until some operation has been completed, such as loading data from a server.
canActivate	This property is used to specify the guards that will be used to determine whether a route can be activated.
canActivateChild	This property is used to specify the guards that will be used to determine whether a child route can be activated.
canDeactivate	This property is used to specify the guards that will be used to determine whether a route can be deactivated.
canLoad	This property is used to guard routes that load feature modules dynamically, as described in the “Loading Feature Modules Dynamically” section.

Delaying Navigation with a Resolver

A common reason for guarding routes is to ensure that the application has received the data that it requires before a route is activated. The example application loads data from the RESTful web service asynchronously, which means there can be a delay between the moment at which the browser is asked to send the HTTP request and the moment at which the response is received and the data is processed. You may not have noticed this delay as you followed the examples because the browser and the web service are running on the same machine. In a deployed application, there is a much greater prospect of there being a delay, caused by network congestion, a high server load, or a dozen other factors.

To simulate network congestion, Listing 26-1 modifies the RESTful data source class to introduce a delay after the response is received from the web service.

Listing 26-1. Adding a Delay in the rest.datasource.ts File in the src/app/model Folder

```
import { Injectable, Inject, InjectionToken } from "@angular/core";
import { HttpClient, HttpHeaders } from "@angular/common/http";
import { catchError, delay, Observable } from "rxjs";
import { Product } from "./product.model";

export const REST_URL = new InjectionToken("rest_url");

@Injectable()
export class RestDataSource {
    constructor(private http: HttpClient,
        @Inject(REST_URL) private url: string) { }

    getData(): Observable<Product[]> {
        return this.sendRequest<Product[]>("GET", this.url).pipe(delay(5000));
    }

    saveProduct(product: Product): Observable<Product> {
        return this.sendRequest<Product>("POST", this.url, product);
    }
}
```

```

updateProduct(product: Product): Observable<Product> {
    return this.sendRequest<Product>("PUT",
        `${this.url}/${product.id}`, product);
}

deleteProduct(id: number): Observable<Product> {
    return this.sendRequest<Product>("DELETE", `${this.url}/${id}`);
}

private sendRequest<T>(verb: string, url: string, body?: Product)
    : Observable<T> {

    let myHeaders = new HttpHeaders();
    myHeaders = myHeaders.set("Access-Key", "<secret>");
    myHeaders = myHeaders.set("Application-Names", ["exampleApp", "proAngular"]);

    return this.http.request<T>(verb, url, {
        body: body,
        headers: myHeaders
    }).pipe(catchError((error: Response) => {
        throw(`Network Error: ${error.statusText} (${error.status})`)
    }));
}
}

```

The delay is added using the Reactive Extensions `delay` method and is applied to create a five-second delay, which is long enough to create a noticeable pause without being too painful to wait for every time the application is reloaded. To change the delay, increase or decrease the argument for the `delay` method, which is expressed in milliseconds.

The effect of the delay is that the user is presented with an incomplete and confusing layout while the application is waiting for the data to load, as shown in Figure 26-2.

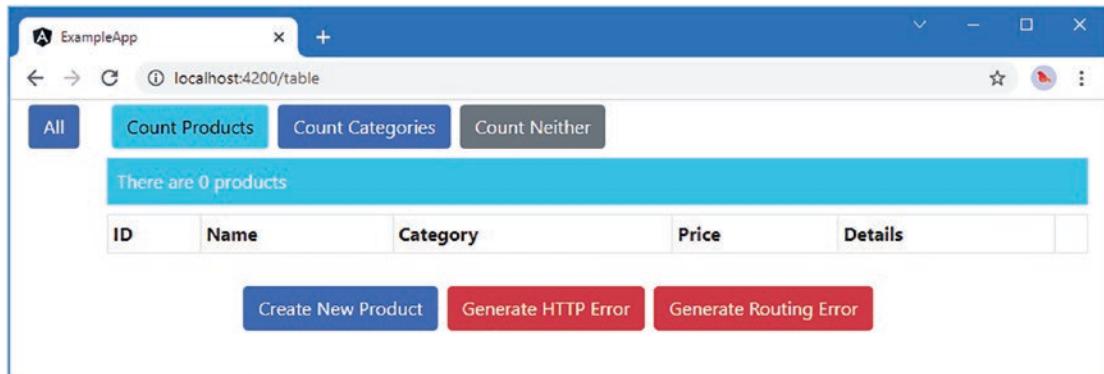


Figure 26-2. Waiting for data

Creating a Resolver Service

A *resolver* is used to ensure that a task is performed before a route can be activated. To create a resolver, I added a file called `model.resolver.ts` in the `src/app/model` folder and defined the class shown in Listing 26-2.

Listing 26-2. The Contents of the `model.resolver.ts` File in the `src/app/model` Folder

```
import { Injectable } from "@angular/core";
import { ActivatedRouteSnapshot, RouterStateSnapshot } from "@angular/router";
import { Observable } from "rxjs";
import { Model } from "./repository.model"
import { Product } from "./product.model";

@Injectable()
export class ModelResolver {

    constructor(private model: Model) { }

    resolve(route: ActivatedRouteSnapshot,
        state: RouterStateSnapshot): Observable<Product | undefined> {

        return this.model.getProductObservable(1);
    }
}
```

Resolvers are classes that define a `resolve` method that accepts two arguments. The first argument is an `ActivatedRouteSnapshot` object, which describes the route that is being navigated to using the properties described in Chapter 24. The second argument is a `RouterStateSnapshot` object, which describes the current route through a single property called `url`. These arguments can be used to adapt the resolver to the navigation that is about to be performed, although neither is required by the resolver in the listing, which uses the same behavior regardless of the routes that are being navigated to and from.

Note All of the guards described in this chapter can implement interfaces defined in the `@angular/router` module. For example, resolvers can implement an interface called `Resolve`. These interfaces are optional, and I have not used them in this chapter.

The `resolve` method can return three different types of result, as described in Table 26-3.

Table 26-3. The Result Types Allowed by the `resolve` Method

Result Type	Description
<code>Observable<any></code>	The browser will activate the new route when the <code>Observer</code> emits an event.
<code>Promise<any></code>	The browser will activate the new route when the <code>Promise</code> resolves.
Any other result	The browser will activate the new route as soon as the method produces a result.

The Observable and Promise results are useful when dealing with asynchronous operations, such as requesting data using an HTTP request. Angular waits until the asynchronous operation is complete before activating the new route. Any other result is interpreted as the result of a synchronous operation, and Angular will activate the new route immediately.

The resolver in Listing 26-2 uses its constructor to receive a Model object via dependency injection. When the resolve method is called, it calls the getProductObservable method, which returns an observable that will emit a result only once data has been received. Angular will subscribe to the Observable and delay activating the new route until it emits an event.

The observable returned by the getProductObservable method will emit an event immediately once data has been received, which is important because Angular will call the guard's resolve method every time that the application tries to navigate to a route to which the resolver has been applied.

Notice that I don't care about the data produced by the repository through the observable. All that matters from the perspective of the guard is that the observable returned by the getProductObservable method will emit an event that indicates data has been received.

Registering the Resolver Service

The next step is to register the resolver as a service in its feature module, as shown in Listing 26-3.

Listing 26-3. Registering the Resolver in the model.module.ts File in the src/app/model Folder

```
import { NgModule } from "@angular/core";
import { StaticDataSource } from "./static.datasource";
import { Model } from "./repository.model";
import { HttpClientJsonpModule, HttpClientModule } from "@angular/common/http";
import { RestDataSource, REST_URL } from "./rest.datasource";
import { ModelResolver } from "./model.resolver";

@NgModule({
  imports: [HttpClientModule, HttpClientJsonpModule],
  providers: [Model, RestDataSource,
    { provide: REST_URL, useValue: `http://${location.hostname}:3500/products` },
    ModelResolver]
})
export class ModelModule { }
```

Applying the Resolver

The resolver is applied to routes using the resolve property, as shown in Listing 26-4.

Listing 26-4. Applying a Resolver in the app.routing.ts File in the src/app Folder

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";
import { ModelResolver } from "./model/model.resolver";
```

```

const childRoutes: Routes = [
  {
    path: "",
    children: [{ path: "products", component: ProductCountComponent },
               { path: "categories", component: CategoryCountComponent },
               { path: "", component: ProductCountComponent }],
    resolve: { model: ModelResolver }
  }
];

const routes: Routes = [
  { path: "form/:mode/:id", component: FormComponent },
  { path: "form/:mode", component: FormComponent },
  { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
  { path: "table", component: TableComponent, children: childRoutes },
  { path: "table/:category", component: TableComponent, children: childRoutes },
  { path: "", redirectTo: "/table", pathMatch: "full" },
  { path: "**", component: NotFoundComponent }
]

export const routing = RouterModule.forRoot(routes);

```

The `resolve` property accepts a map object whose property values are the resolver classes that will be applied to the route. (The property names do not matter.) I want to apply the resolver to all the views that display the product table, so to avoid duplication, I created a route with the `resolve` property and used it as the parent for the existing child routes.

The effect is that the user will see no content until the data has been received from the web service and processed by the application, as shown in Figure 26-3.

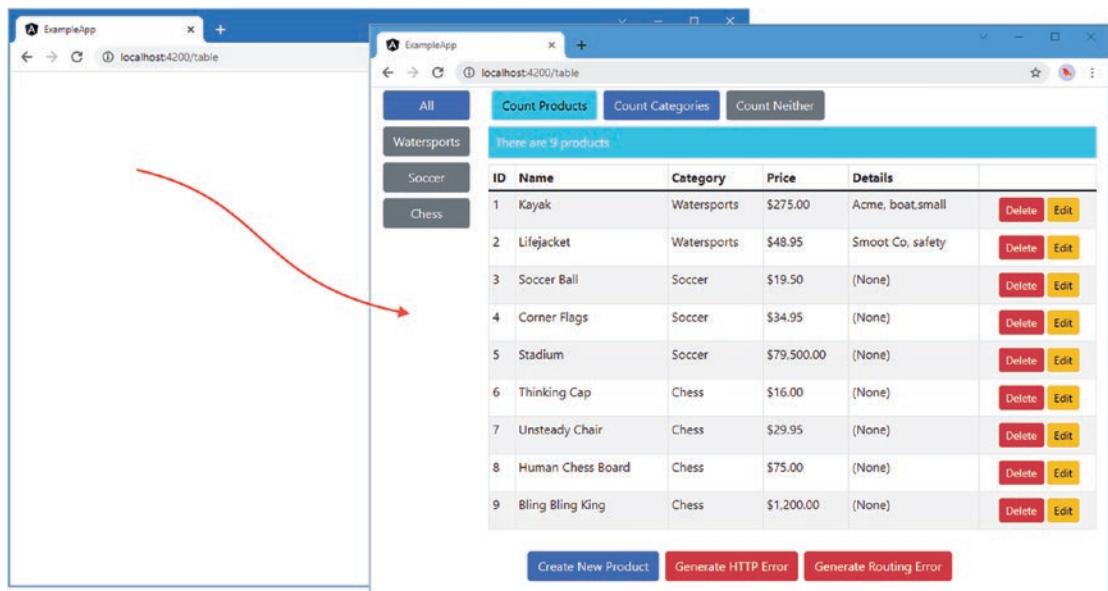


Figure 26-3. Using a route guard

Displaying Placeholder Content

Angular uses the resolver before activating any of the routes to which it has been applied, which prevents the user from seeing the product table until the model has been populated with the data from the RESTful web service. Sadly, that just means the user sees an empty window while the browser is waiting for the server to respond. To address this, Listing 26-5 enhances the resolver to use the message service to tell the user what is happening when the data is being loaded.

Listing 26-5. Displaying a Message in the model.resolver.ts File in the src/app/model Folder

```
import { Injectable } from "@angular/core";
import { ActivatedRouteSnapshot, RouterStateSnapshot } from "@angular/router";
import { Observable } from "rxjs";
import { Model } from "./repository.model"
import { Product } from "./product.model";
import { MessageService } from "../messages/message.service";
import { Message } from "../messages/message.model";

@Injectable()
export class ModelResolver {

  constructor(private model: Model,
    private messages: MessageService) { }

  resolve(route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<Product | undefined> {
    this.messages.reportMessage(new Message("Loading data..."));
    return this.model.getProductObservable(1);
  }
}
```

The guard uses the message service to give the user an indication that something is happening and relies on the way that the service removes messages when navigation events are received. The result is that the user sees a loading message until data is received, as shown in Figure 26-4.

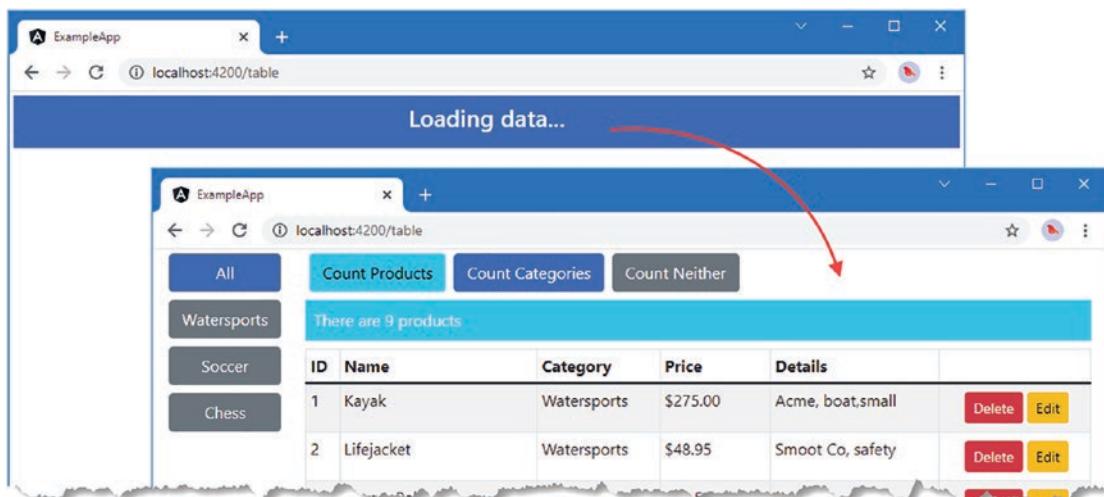


Figure 26-4. Displaying a loading message

Using a Resolver to Prevent URL Entry Problems

A resolver can be applied more broadly so that it protects multiple routes, which extends the loading message when the user navigates directly to a URL for a specific product, as shown in Listing 26-6.

Listing 26-6. Applying the Resolver to Other Routes in the app.routing.ts File in the src/app Folder

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";
import { ModelResolver } from "./model/model.resolver";

const childRoutes: Routes = [
  {
    path: "",
    children: [{ path: "products", component: ProductCountComponent },
               { path: "categories", component: CategoryCountComponent },
               { path: "", component: ProductCountComponent }],
    resolve: { model: ModelResolver }
  }
];

const routes: Routes = [
  {
    path: "form/:mode/:id", component: FormComponent,
    resolve: { model: ModelResolver }
  },
  {
    path: "form/:mode", component: FormComponent,
    resolve: { model: ModelResolver }
  },
  { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
  { path: "table", component: TableComponent, children: childRoutes },
  { path: "table/:category", component: TableComponent, children: childRoutes },
  { path: "", redirectTo: "/table", pathMatch: "full" },
  { path: "**", component: NotFoundComponent }
]

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

Applying the ModelResolver class to the routes that target FormComponent ensures that the user is shown the placeholder message while the data is loaded, as shown in Figure 26-5.

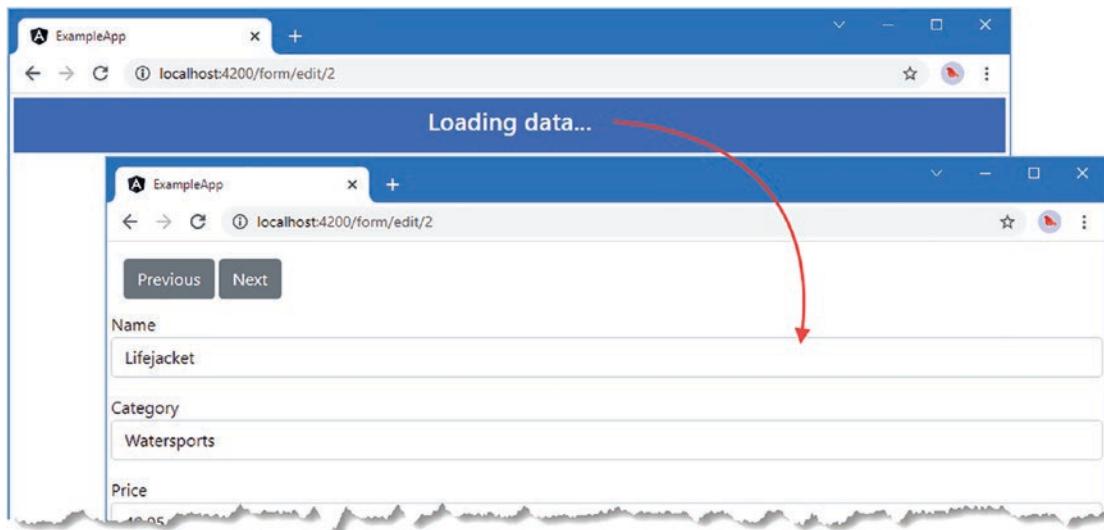


Figure 26-5. Expanding the use of a resolver

Preventing Navigation with Guards

Resolvers are used to delay navigation while the application performs some prerequisite work, such as loading data. The other guards that Angular provides are used to control whether navigation can occur at all, which can be useful when you want to alert the user to prevent potentially unwanted operations (such as abandoning data edits) or limit access to parts of the application unless the application is in a specific state, such as when a user has been authenticated.

Many uses for route guards introduce an additional interaction with the user, either to gain explicit approval to perform an operation or to obtain additional data, such as authentication credentials. For this chapter, I am going to handle this kind of interaction by extending the message service so that messages can require user input. In Listing 26-7, I have added an optional responses constructor argument/property to the Message model class, which will allow messages to contain prompts to the user and callbacks that will be invoked when they are selected. The responses property is an array of TypeScript tuples, where the first value is the name of the response, which will be presented to the user, and the second value is the callback function, which will be passed the name as its argument.

Listing 26-7. Adding Responses in the message.model.ts File in the src/app/messages Folder

```
export class Message {

    constructor(public text: string,
        public error: boolean = false,
        public responses?: [string, (x: string) => void][] ) { }

}
```

The only other change required to implement this feature is to present the response options to the user. Listing 26-8 adds button elements below the message text for each response. Clicking the buttons will invoke the callback function.

Listing 26-8. Presenting Responses in the message.component.html File in the src/app/core Folder

```
<div *ngIf="lastMessage"
      class="bg-primary text-white p-2 text-center"
      [class.bg-danger]="lastMessage.error">
    <h4>{{lastMessage.text}}</h4>
</div>
<div class="text-center my-2">
  <button *ngFor="let resp of lastMessage?.responses; let i = index"
          (click)="resp[1](resp[0])"
          class="btn btn-primary m-2" [class.btn-secondary]="i > 0">
    {{resp[0]}}
  </button>
</div>
```

Preventing Route Activation

Guards can be used to prevent a route from being activated, helping to protect the application from entering an unwanted state or warning the user about the impact of performing an operation. To demonstrate, I am going to guard the /form/create URL to prevent the user from starting the process of creating a new product unless the user agrees to the application's terms and conditions.

Guards for route activation are classes that define a method called `canActivate`, which receives the same `ActivatedRouteSnapshot` and `RouterStateSnapshot` arguments as resolvers. The `canActivate` method can be implemented to return three different result types, as described in Table 26-4.

Table 26-4. The Result Types Allowed by the `canActivate` Method

Result Type	Description
<code>boolean</code>	This type of result is useful when performing synchronous checks to see whether the route can be activated. A <code>true</code> result will activate the route, and a result of <code>false</code> will not, effectively ignoring the navigation request.
<code>Observable<boolean></code>	This type of result is useful when performing asynchronous checks to see whether the route can be activated. Angular will wait until the <code>Observable</code> emits a value, which will be used to determine whether the route is activated. When using this kind of result, it is important to terminate the <code>Observable</code> by calling the <code>complete</code> method; otherwise, Angular will just keep waiting.
<code>Promise<boolean></code>	This type of result is useful when performing asynchronous checks to see whether the route can be activated. Angular will wait until the <code>Promise</code> is resolved and activate the route if it yields <code>true</code> . If the <code>Promise</code> yields <code>false</code> , then the route will not be activated, effectively ignoring the navigation request.

To get started, I added a file called `terms.guard.ts` to the `src/app` folder and defined the class shown in Listing 26-9.

Listing 26-9. The Contents of the terms.guard.ts File in the src/app Folder

```

import { Injectable } from "@angular/core";
import {
    ActivatedRouteSnapshot, RouterStateSnapshot,
    Router
} from "@angular/router";
import { MessageService } from "./messages/message.service";
import { Message } from "./messages/message.model";

@Injectable()
export class TermsGuard {

    constructor(private messages: MessageService,
                private router: Router) { }

    canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Promise<boolean> | boolean {

        if (route.params["mode"] == "create") {

            return new Promise<boolean>((resolve) => {
                let responses: [string, () => void][] =
                    [["Yes", () => resolve(true)], ["No", () => resolve(false)]];
                this.messages.reportMessage(
                    new Message("Do you accept the terms & conditions?",
                               false, responses));
            });
        } else {
            return true;
        }
    }
}

```

The `canActivate` method can return two different types of results. The first type is a boolean, which allows the guard to respond immediately for routes that it doesn't need to protect, which in this case is any that lacks a parameter called `mode` whose value is `create`. If the URL matched by the route doesn't contain this parameter, the `canActivate` method returns `true`, which tells Angular to activate the route. This is important because the edit and create features both rely on the same routes, and the guard should not interfere with edit operations.

The other type of result is a `Promise<boolean>`, which I have used instead of `Observable<true>` for variety. The `Promise` uses the modifications to the message service to solicit a response from the user, confirming they accept the (unspecified) terms and conditions. There are two possible responses from the user. If the user clicks the Yes button, then the `Promise` will resolve and yield `true`, which tells Angular to activate the route, displaying the form that is used to create a new product. The `Promise` will resolve and yield `false` if the user clicks the No button, which tells Angular to ignore the navigation request.

[Listing 26-10](#) registers the `TermsGuard` as a service so that it can be used in the application's routing configuration.

Listing 26-10. Registering the Guard as a Service in the app.module.ts File in the src/app Folder

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { ModelModule } from './model/model.module';
import { CoreModule } from './core/core.module';
import { TableComponent } from './core/table.component';
import { FormComponent } from './core/form.component';
import { MessageModule } from './messages/message.module';
import { MessageComponent } from './messages/message.component';
import { AppComponent } from './app.component';
import { routing } from './app.routing';
import { TermsGuard } from './terms.guard"

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, ModelModule, CoreModule, MessageModule, routing],
  providers: [TermsGuard],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Listing 26-11 applies the guard to the routing configuration. Activation guards are applied to a route using the canActivate property, which is assigned an array of guard services. The canActivate method of all the guards must return true (or return an Observable or Promise that eventually yields true) before Angular will activate the route.

Listing 26-11. Applying the Guard to a Route in the app.routing.ts File in the src/app Folder

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";
import { ModelResolver } from "./model/model.resolver";
import { TermsGuard } from './terms.guard";

const childRoutes: Routes = [
  {
    path: "",
    children: [{ path: "products", component: ProductCountComponent },
               { path: "categories", component: CategoryCountComponent },
               { path: "", component: ProductCountComponent }],
    resolve: { model: ModelResolver }
  }
];
const routes: Routes = [
  {
    path: "form/:mode/:id", component: FormComponent,
    resolve: { model: ModelResolver }
  },
  
```

```
{
  path: "form/:mode", component: FormComponent,
  resolve: { model: ModelResolver },
  canActivate: [TermsGuard]
},
{ path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
{ path: "table", component: TableComponent, children: childRoutes },
{ path: "table/:category", component: TableComponent, children: childRoutes },
{ path: "", redirectTo: "/table", pathMatch: "full" },
{ path: "**", component: NotFoundComponent }]
}

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

The effect of creating and applying the activation guard is that the user is prompted when clicking the Create New Product button, as shown in Figure 26-6. If they respond by clicking the Yes button, then the navigation request will be completed, and Angular will activate the route that selects the form component, which will allow a new product to be created. If the user clicks the No button, then the navigation request will be canceled. In both cases, the routing system emits an event that is received by the component that displays the messages to the user, which clears its display and ensures that the user doesn't see stale messages.

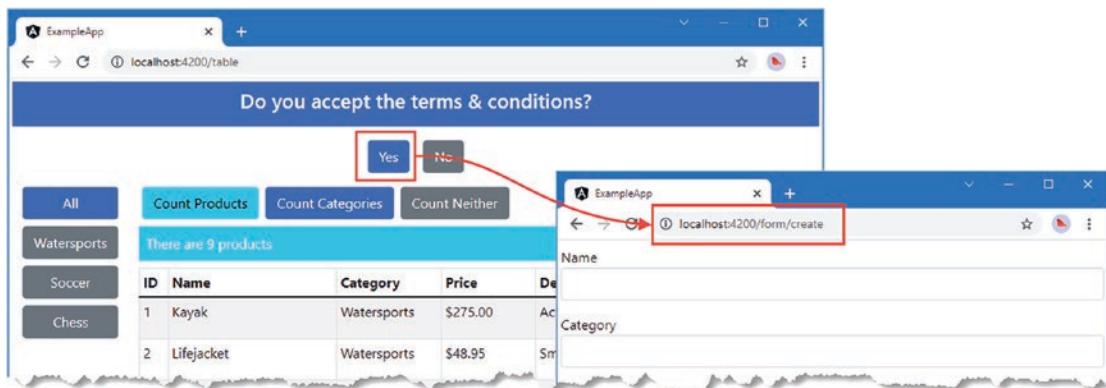


Figure 26-6. Guarding route activation

Consolidating Child Route Guards

If you have a set of child routes, you can guard against their activation using a child route guard, which is a class that defines a method called `canActivateChild`. The guard is applied to the parent route in the application's configuration, and the `canActivateChild` method is called whenever any of the child routes are about to be activated. The method receives the same `ActivatedRouteSnapshot` and `RouterStateSnapshot` objects as the other guards and can return the set of result types described in Table 26-4.

This guard in this example is more readily dealt with by changing the configuration before implementing the `canActivateChild` method, as shown in Listing 26-12.

Listing 26-12. Guarding Child Routes in the app.routing.ts File in the src/app Folder

```

import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";
import { ModelResolver } from "./model/model.resolver";
import { TermsGuard } from "./terms.guard";

const childRoutes: Routes = [
  {
    path: "",
    canActivateChild: [TermsGuard],
    children: [{ path: "products", component: ProductCountComponent },
               { path: "categories", component: CategoryCountComponent },
               { path: "", component: ProductCountComponent }],
    resolve: { model: ModelResolver }
  }
];

const routes: Routes = [
  {
    path: "form/:mode/:id", component: FormComponent,
    resolve: { model: ModelResolver }
  },
  {
    path: "form/:mode", component: FormComponent,
    resolve: { model: ModelResolver },
    canActivate: [TermsGuard]
  },
  { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
  { path: "table", component: TableComponent, children: childRoutes },
  { path: "table/:category", component: TableComponent, children: childRoutes },
  { path: "", redirectTo: "/table", pathMatch: "full" },
  { path: "**", component: NotFoundComponent }
]

export const routing = RouterModule.forRoot(routes);

```

Child route guards are applied to a route using the `canActivateChild` property, which is set to an array of service types implementing the `canActivateChild` method. This method will be called before Angular activates any of the route's children. Listing 26-13 adds the `canActivateChild` method to the guard class from the previous section.

Listing 26-13. Implementing Child Route Guards in the terms.guard.ts File in the src/app Folder

```

import { Injectable } from "@angular/core";
import {
  ActivatedRouteSnapshot, RouterStateSnapshot,
  Router
} from "@angular/router";
import { MessageService } from "./messages/message.service";
import { Message } from "./messages/message.model";

```

```

@Injectable()
export class TermsGuard {

  constructor(private messages: MessageService,
    private router: Router) { }

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
    Promise<boolean> | boolean {

    if (route.params["mode"] == "create") {

      return new Promise<boolean>((resolve) => {
        let responses: [string, () => void][] =
          [["Yes", () => resolve(true)], ["No", () => resolve(false)]];
        this.messages.reportMessage(
          new Message("Do you accept the terms & conditions?", false, responses));
      });
    } else {
      return true;
    }
  }

  canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
    Promise<boolean> | boolean {

    if (route.url.length > 0
      && route.url[route.url.length - 1].path == "categories") {

      return new Promise<boolean>((resolve, reject) => {
        let responses: [string, (arg: string) => void][] = [
          ["Yes", () => resolve(true)],
          ["No ", () => resolve(false)]
        ];
        this.messages.reportMessage(
          new Message("Do you want to see the categories component?", false, responses));
      });
    } else {
      return true;
    }
  }
}

```

The guard only protects the categories child route and will return true immediately for any other route. The guard prompts the user using the message service but does something different if the user clicks the No button. In addition to rejecting the active route, the guard navigates to a different URL using the Router service, which is received as a constructor argument. This is a common pattern for authentication when the user is redirected to a component that will solicit security credentials if a restricted operation is attempted. The example is simpler in this case, and the guard navigates to a sibling route that shows a different component. (You can see an example of using route guards for navigation in the SportsStore application.)

To see the effect of the guard, click the Count Categories button, as shown in Figure 26-7. Responding to the prompt by clicking the Yes button will show the CategoryCountComponent, which displays the number of categories in the table. Clicking No will reject the active route and navigate to a route that displays the ProductCountComponent instead.

Note Guards are applied only when the active route changes. So, for example, if you click the Count Categories button when the /table URL is active, then you will see the prompt, and clicking Yes will change the active route. But nothing will happen if you click the Count Categories button again because Angular doesn't trigger a route change when the target route and the active route are the same.

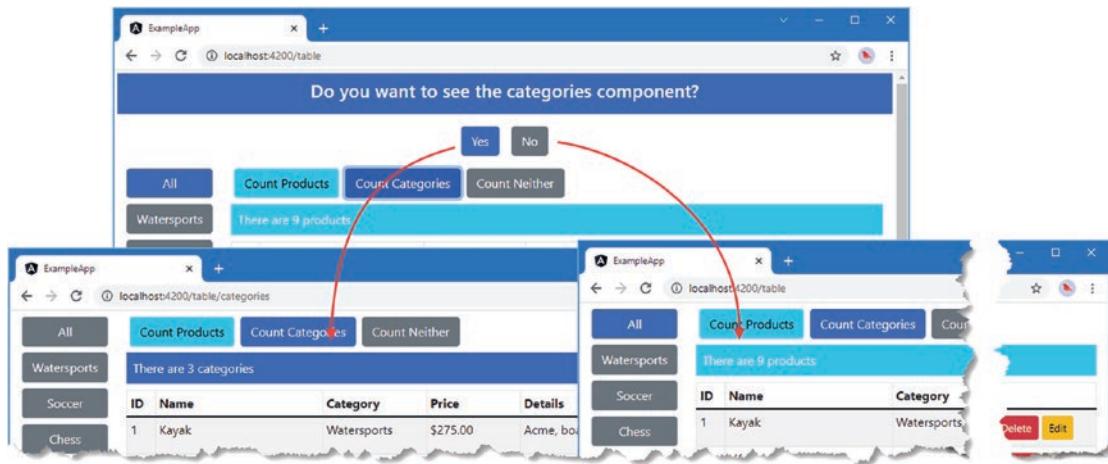


Figure 26-7. Guarding child routes

Preventing Route Deactivation

When you start working with routes, you will tend to focus on the way that routes are activated to respond to navigation and present new content to the user. But equally important is route *deactivation*, which occurs when the application navigates away from a route.

The most common use for deactivation guards is to prevent the user from navigating when there are unsaved edits to data. In this section, I will create a guard that warns the user when they are about to abandon unsaved changes when editing a product. In preparation for this, Listing 26-14 changes the FormComponent class to simplify the work of the guard.

Listing 26-14. Preparing for the Guard in the form.component.ts File in the src/app/core Folder

```
import { Component } from "@angular/core";
import { FormControl, NgForm, Validators, FormGroup, FormArray }
    from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
```

```

import { MODES, SharedState, StateUpdate } from "./sharedState.service";
import { FilteredFormArray } from "./filteredFormArray";
import { LimitValidator } from "../validation/limit";
import { UniqueValidator } from "../validation/unique";
import { ProhibitedValidator } from "../validation/prohibited";
import { ActivatedRoute, Router } from "@angular/router";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;

  keywordGroup = new FilteredFormArray([
    this.createKeywordFormControl(),
  ], {
    validators: UniqueValidator.unique()
  })

  // ...form structure and methods omitted for brevity...

  // resetForm() {
  //   this.keywordGroup.clear();
  //   this.keywordGroup.push(this.createKeywordFormControl());
  //   this.editing = true;
  //   this.product = new Product();
  //   this.productForm.reset();
  // }

  unsavedChanges(): boolean {
    return this.productForm.dirty;
  }

  createKeywordFormControl(): FormControl {
    return new FormControl("", { validators:
      Validators.pattern("^[A-Za-z ]+$/) });
  }

  addKeywordControl() {
    this.keywordGroup.push(this.createKeywordFormControl());
  }

  removeKeywordControl(index: number) {
    this.keywordGroup.removeAt(index);
  }
}

```

The `unsavedChanges` method will be used to indicate whether the user has made changes since editing began, which is determined by checking the state of the top-level `FormGroup`.

A corresponding change is required in the template so that the Cancel button doesn't invoke the form's reset event handler, as shown in Listing 26-15.

Listing 26-15. Disabling Form Reset in the `form.component.html` File in the `src/app/core` Folder

```
<div *ngIf="editing" class="p-2">
  <button class="btn btn-secondary m-1"
    [routerLink]=["/form", 'edit',
      model.getPreviousProductId(product.id) | async]">
    Previous
  </button>
  <button class="btn btn-secondary"
    [routerLink]=["/form", 'edit',
      model.getNextProductId(product.id) | async]">
    Next
  </button>
</div>
<form [formGroup]="productForm" #form="ngForm" (ngSubmit)="submitForm()">

  <!-- ...elements omitted for brevity... -->

  <div class="mt-2">
    <button type="submit" class="btn btn-primary"
      [class.btn-warning]="editing"
      [disabled]="form.invalid">
      {{editing ? "Save" : "Create"}}
    </button>
    <button type="button" class="btn btn-secondary m-1" routerLink="/">
      Cancel
    </button>
  </div>
</form>
```

To create the guard, I added a file called `unsaved.guard.ts` in the `src/app/core` folder and defined the class shown in Listing 26-16.

Listing 26-16. The Contents of the `unsaved.guard.ts` File in the `src/app/core` Folder

```
import { Injectable } from "@angular/core";
import {
  ActivatedRouteSnapshot, RouterStateSnapshot,
  Router
} from "@angular/router";
import { Observable, Subject } from "rxjs";
import { MessageService } from "../messages/message.service";
import { Message } from "../messages/message.model";
import { FormComponent } from "./form.component";
```

```

@Injectable()
export class UnsavedGuard {

  constructor(private messages: MessageService,
              private router: Router) { }

  canDeactivate(component: FormComponent, route: ActivatedRouteSnapshot,
                state: RouterStateSnapshot): Observable<boolean> | boolean {
    if (component.editing && component.unsavedChanges()) {
      let subject = new Subject<boolean>();

      let responses: [string, (r: string) => void][] = [
        ["Yes", () => {
          subject.next(true);
          subject.complete();
        }],
        ["No", () => {
          this.router.navigateByUrl(this.router.url);
          subject.next(false);
          subject.complete();
        }]
      ];
      this.messages.reportMessage(new Message("Discard Changes?",
                                              true, responses));
      return subject;
    }
    return true;
  }
}

```

Deactivation guards define a class called `canDeactivate` that receives three arguments: the component that is about to be deactivated and the `ActivatedRouteSnapshot` and `RouterStateSnapshot` objects. This guard checks to see whether there are unsaved edits in the component and prompts the user if there are. For variety, this guard uses an `Observable<true>`, implemented as a `Subject<true>` instead of a `Promise<true>`, to tell Angular whether it should activate the route, based on the response selected by the user.

Tip Notice that I call the `complete` method on the `Subject` after calling the `next` method. Angular will wait indefinitely for the `complete` method to be called, effectively freezing the application.

The next step is to register the guard as a service in the module that contains it, as shown in Listing 26-17.

Listing 26-17. Registering the Guard as a Service in the core.module.ts File in the src/app/core Folder

```

import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";

```

```

import { TableComponent } from "./table.component";
import { FormComponent } from "./form.component";
//import { SharedState } from "./sharedState.service";
import { ValidationHelper } from "./validation_helper";
import { ValidationErrorsDirective } from "./validationErrors.directive";
import { HiLowValidatorDirective } from "../validation/hilow";
import { RouterModule } from "@angular/router";
import { ProductCountComponent } from "./productCount.component";
import { CategoryCountComponent } from "./categoryCount.component";
import { NotFoundComponent } from "./notFound.component";
import { UnsavedGuard } from "./unsaved.guard";

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule, ReactiveFormsModule,
    RouterModule],
  declarations: [TableComponent, FormComponent, ValidationHelper,
    ValidationErrorsDirective, HiLowValidatorDirective, ProductCountComponent,
    CategoryCountComponent, NotFoundComponent],
  exports: [ModelModule, TableComponent, FormComponent],
  providers: [UnsavedGuard]
})
export class CoreModule { }

```

Finally, Listing 26-18 applies the guard to the application's routing configuration. Deactivation guards are applied to routes using the canDeactivate property, which is set to an array of guard services.

Listing 26-18. Applying the Guard in the app.routing.ts File in the src/app Folder

```

import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";
import { ModelResolver } from "./model/model.resolver";
import { TermsGuard } from "./terms.guard";
import { UnsavedGuard } from "./core/unsaved.guard";

const childRoutes: Routes = [
  {
    path: "",
    canActivateChild: [TermsGuard],
    children: [{ path: "products", component: ProductCountComponent },
      { path: "categories", component: CategoryCountComponent },
      { path: "", component: ProductCountComponent }],
    resolve: { model: ModelResolver }
  }
];

```

```
const routes: Routes = [
  {
    path: "form/:mode/:id", component: FormComponent,
    resolve: { model: ModelResolver },
    canDeactivate: [UnsavedGuard]
  },
  {
    path: "form/:mode", component: FormComponent,
    resolve: { model: ModelResolver },
    canActivate: [TermsGuard]
  },
  { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
  { path: "table", component: TableComponent, children: childRoutes },
  { path: "table/:category", component: TableComponent, children: childRoutes },
  { path: "", redirectTo: "/table", pathMatch: "full" },
  { path: "**", component: NotFoundComponent }
]

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

To see the effect of the guard, click one of the Edit buttons in the table; edit the data in one of the text fields; then click the Cancel, Next, or Previous button. The guard will prompt you before allowing Angular to activate the route you selected, as shown in Figure 26-8.

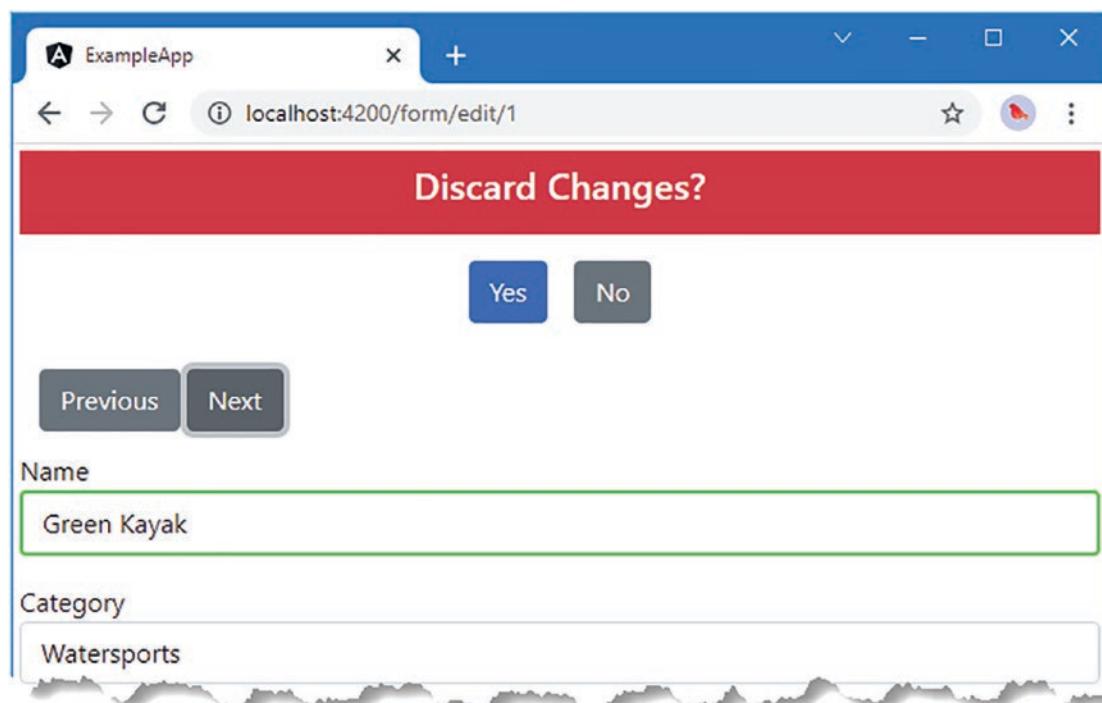


Figure 26-8. Guarding route deactivation

Loading Feature Modules Dynamically

Angular supports loading feature modules only when they are required, known as *dynamic loading* or *lazy loading*. This can be useful for functionality that is unlikely to be required by all users. In the sections that follow, I create a simple feature module and demonstrate how to configure the application so that Angular will load the module only when the application navigates to a specific URL.

Note Loading modules dynamically is a trade-off. The application will be smaller and faster to download for most users, improving their overall experience. But users who require the dynamically loaded features will have to wait while Angular gets the module and its dependencies. The effect can be jarring because the user has no idea that some features have been loaded and others have not. When you create dynamically loaded modules, you are balancing improving the experience for some users against making it worse for others. Consider how your users fall into these groups and take care not to degrade the experience of your most valuable and important customers.

Creating a Simple Feature Module

Dynamically loaded modules must contain only functionality that not all users require. I can't use the existing modules because they provide the core functionality for the application, which means that I need a new module for this part of the chapter. I started by creating a folder called `ondemand` in the `src/app` folder. To give the new module a component, I added a file called `ondemand.component.ts` in the `example/app/ondemand` folder and added the code shown in Listing 26-19.

Caution It is important not to create dependencies between other parts of the application and the classes in the dynamically loaded module so that the JavaScript module loader doesn't try to load the module before it is required.

Listing 26-19. The Contents of the `ondemand.component.ts` File in the `src/app/ondemand` Folder

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

@Component({
  selector: "ondemand",
  templateUrl: "ondemand.component.html"
})
export class OndemandComponent {}
```

To provide the component with a template, I added a file called `ondemand.component.html` and added the markup shown in Listing 26-20.

Listing 26-20. The `ondemand.component.html` File in the `src/app/ondemand` Folder

```
<div class="bg-primary text-white p-2">This is the ondemand component</div>
<button class="btn btn-primary m-2" routerLink="/" >Back</button>
```

The template contains a message that will make it obvious when the component is selected and that contains a button element that will navigate back to the application's root URL when clicked.

To define the module, I added a file called `ondemand.module.ts` and added the code shown in Listing 26-21.

Listing 26-21. The Contents of the `ondemand.module.ts` File in the `src/app/ondemand` Folder

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { OndemandComponent } from "./ondemand.component";

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

The module imports the `CommonModule` functionality, which is used instead of the browser-specific `BrowserModule` to access the built-in directives in feature modules that are loaded on-demand.

Loading the Module Dynamically

There are two steps to set up dynamically loading a module. The first is to set up a routing configuration inside the feature module to provide the rules that will allow Angular to select a component when the module is loaded. Listing 26-22 adds a single route to the feature module.

Listing 26-22. Defining Routes in the `ondemand.module.ts` File in the `src/app/ondemand` Folder

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { OndemandComponent } from "./ondemand.component";
import { RouterModule } from "@angular/router";

let routing = RouterModule.forChild([
  { path: "", component: OndemandComponent }
]);

@NgModule({
  imports: [CommonModule, routing],
  declarations: [OndemandComponent],
  exports: [OndemandComponent]
})
export class OndemandModule { }
```

Routes in dynamically loaded modules are defined using the same properties as in the main part of the application and can use all the same features, including child components, guards, and redirections. The route defined in the listing matches the empty path and selects the `OndemandComponent` for display.

One important difference is the method used to generate the module that contains the routing information, as follows:

```
...
let routing = RouterModule.forChild([
  { path: "", component: OndemandComponent }
]);
...
...
```

When I created the application-wide routing configuration, I used the `RouterModule.forRoot` method. This is the method that is used to set up the routes in the root module of the application. When creating dynamically loaded modules, the `RouterModule.forChild` method must be used; this method creates a routing configuration that is merged into the overall routing system when the module is loaded.

Creating a Route to Dynamically Load a Module

The second step to set up a dynamically loaded module is to create a route in the main part of the application that provides Angular with the module's location, as shown in Listing 26-23.

Listing 26-23. Creating an On-Demand Route in the `app.routing.ts` File in the `src/app` Folder

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";
import { ModelResolver } from "./model/model.resolver";
import { TermsGuard } from "./terms.guard";
import { UnsavedGuard } from "./core/unsaved.guard";

const childRoutes: Routes = [
  {
    path: "",
    canActivateChild: [TermsGuard],
    children: [{ path: "products", component: ProductCountComponent },
               { path: "categories", component: CategoryCountComponent },
               { path: "", component: ProductCountComponent }],
    resolve: { model: ModelResolver }
  }
];

const routes: Routes = [
  {
    path: "ondemand",
    loadChildren: () => import("./ondemand/ondemand.module")
      .then(m => m.OndemandModule)
  },
  {
    path: "form/:mode/:id", component: FormComponent,
    resolve: { model: ModelResolver },
    canDeactivate: [UnsavedGuard]
  },
]
```

```
{
  path: "form/:mode", component: FormComponent,
  resolve: { model: ModelResolver },
  canActivate: [TermsGuard]
},
{ path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
{ path: "table", component: TableComponent, children: childRoutes },
{ path: "table/:category", component: TableComponent, children: childRoutes },
{ path: "", redirectTo: "/table", pathMatch: "full" },
{ path: "**", component: NotFoundComponent }]
}

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

The `loadChildren` property is used to provide Angular with details of how the module should be loaded. The property is assigned a function that invokes `import`, passing in the path to the module. The result is a `Promise` whose `then` method is used to select the module after it has been imported. The function in the listing tells Angular to load the `OndemandModule` class from the `ondemand/ondemand.module` file.

Using a Dynamically Loaded Module

All that remains is to add support for navigating to the URL that will activate the route for the on-demand module, as shown in Listing 26-24, which adds a button to the template for the table component.

Listing 26-24. Adding Navigation in the `table.component.html` File in the `src/app/core` Folder

```
<div class="container-fluid">
  <!-- ...elements omitted for brevity... -->
</div>
<div class="p-2 text-center">
  <button class="btn btn-primary m-1" routerLink="/form/create">
    Create New Product
  </button>
  <button class="btn btn-danger m-1" (click)="deleteProduct(-1)">
    Generate HTTP Error
  </button>
  <button class="btn btn-danger m-1" routerLink="/does/not/exist">
    Generate Routing Error
  </button>
  <button class="btn btn-danger" routerLink="/ondemand">
    Load Module
  </button>
</div>
```

No special measures are required to target a route that loads a module, and the Load Module button in the listing uses the standard `routerLink` attribute to navigate to the URL specified by the route added in Listing 26-23.

Click the Load Module button, and you will see an HTTP request in the browser's F12 developer tools window for the new module. When the button is clicked, Angular uses the routing configuration to load the module, inspect its routing configuration, and select the component that will be displayed to the user, as shown in Figure 26-9.

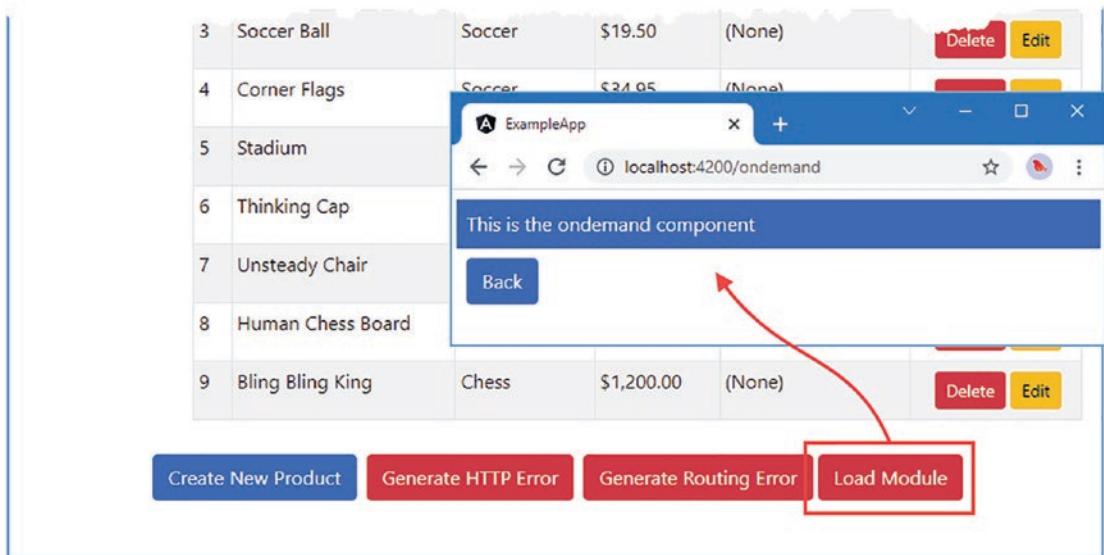


Figure 26-9. Loading a module dynamically

Guarding Dynamic Modules

You can guard against dynamically loading modules to ensure that they are loaded only when the application is in a specific state or when the user has explicitly agreed to wait while Angular does the loading (this latter option is typically used only for administration functions, where the user can be expected to have some understanding of how the application is structured).

The guard for the module must be defined in the main part of the application, so I added a file called `load.guard.ts` in the `src/app` folder and defined the class shown in Listing 26-25.

Listing 26-25. The Contents of the `load.guard.ts` File in the `src/app` Folder

```
import { Injectable } from "@angular/core";
import { Route, Router } from "@angular/router";
import { MessageService } from "./messages/message.service";
import { Message } from "./messages/message.model";

@Injectable()
export class LoadGuard {
    private loaded: boolean = false;

    constructor(private messages: MessageService,
        private router: Router) { }

    canLoad(route: Route): Promise<boolean> | boolean {

        return this.loaded || new Promise<boolean>((resolve, reject) => {
            let responses: [string, (r: string) => void][] = [
                ["Yes", () => {

```

```

        this.loaded = true;
        resolve(true);
    ],
    ["No", () => {
        this.router.navigateByUrl(this.router.url);
        resolve(false);
    }
];
}

this.messages.reportMessage(
    new Message("Do you want to load the module?",
        false, responses));
});
}
}

```

Dynamic loading guards are classes that implement a method called `canLoad`, which is invoked when Angular needs to activate the route to which it is applied, and is provided with a `Route` object that describes the route.

The guard is required only when the URL that loads the module is first activated, so it defines a `loaded` property that is set to `true` when the module has been loaded so that subsequent requests are immediately approved. Otherwise, this guard follows the same pattern as earlier examples and returns a `Promise` that will be resolved when the user clicks one of the buttons displayed by the message service. Listing 26-26 registers the guard as a service in the root module.

Listing 26-26. Registering the Guard as a Service in the `app.module.ts` File in the `src/app` Folder

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { ModelModule } from './model/model.module';
import { CoreModule } from './core/core.module';
import { TableComponent } from './core/table.component';
import { FormComponent } from './core/form.component';
import { MessageModule } from './messages/message.module';
import { MessageComponent } from './messages/message.component';
import { AppComponent } from './app.component';
import { routing } from './app.routing';
import { TermsGuard } from './terms.guard'
import { LoadGuard } from './load.guard';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, ModelModule, CoreModule, MessageModule, routing],
  providers: [TermsGuard, LoadGuard],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Applying a Dynamic Loading Guard

Guards for dynamic loading are applied to routes using the `canLoad` property, which accepts an array of guard types. Listing 26-27 applies the `LoadGuard` class, which was defined in Listing 26-25, to the route that dynamically loads the module.

Listing 26-27. Guarding the Route in the `app.routing.ts` File in the `src/app` Folder

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { ProductCountComponent } from "./core/productCount.component";
import { CategoryCountComponent } from "./core/categoryCount.component";
import { ModelResolver } from "./model/model.resolver";
import { TermsGuard } from "./terms.guard";
import { UnsavedGuard } from "./core/unsaved.guard";
import { LoadGuard } from "./load.guard";

const childRoutes: Routes = [
  {
    path: "",
    canActivateChild: [TermsGuard],
    children: [{ path: "products", component: ProductCountComponent },
               { path: "categories", component: CategoryCountComponent },
               { path: "", component: ProductCountComponent }],
    resolve: { model: ModelResolver }
  }
];
const routes: Routes = [
  {
    path: "ondemand",
    loadChildren: () => import("./ondemand/ondemand.module")
      .then(m => m.OndemandModule),
    canLoad: [LoadGuard]
  },
  {
    path: "form/:mode/:id", component: FormComponent,
    resolve: { model: ModelResolver },
    canDeactivate: [UnsavedGuard]
  },
  {
    path: "form/:mode", component: FormComponent,
    resolve: { model: ModelResolver },
    canActivate: [TermsGuard]
  },
  { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
  { path: "table", component: TableComponent, children: childRoutes },
  { path: "table/:category", component: TableComponent, children: childRoutes },
  { path: "", redirectTo: "/table", pathMatch: "full" },
  { path: "**", component: NotFoundComponent }
]

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

The result is that the user is prompted to determine whether they want to load the module the first time that Angular tries to activate the route, as shown in Figure 26-10.

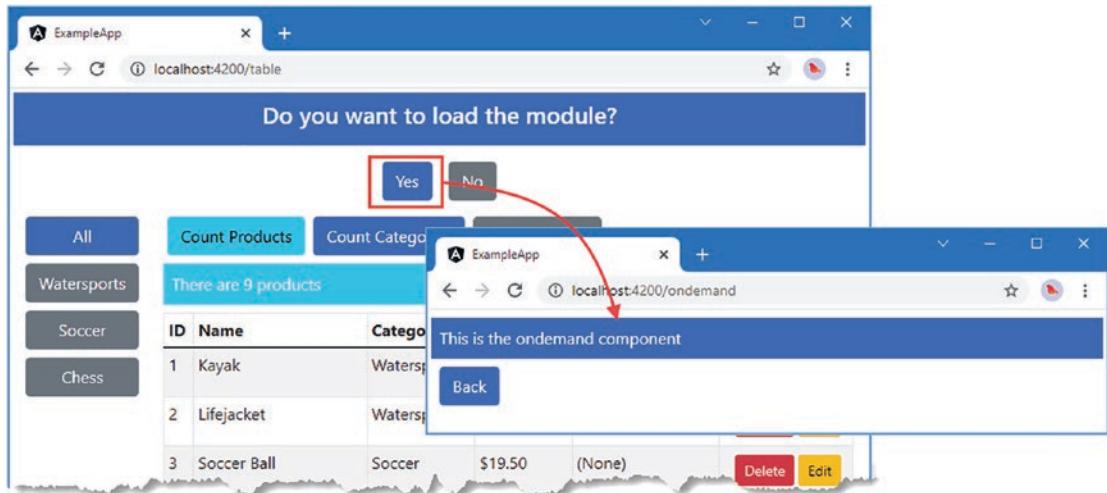


Figure 26-10. Guarding dynamic loading

Targeting Named Outlets

A template can contain more than one `router-outlet` element, which allows a single URL to select multiple components to be displayed to the user.

To demonstrate this feature, I need to add two new components to the `ondemand` module. I started by creating a file called `first.component.ts` in the `src/app/ondemand` folder and using it to define the component shown in Listing 26-28.

Listing 26-28. The Contents of the `first.component.ts` File in the `src/app/ondemand` Folder

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

@Component({
  selector: "first",
  template: `<div class="bg-primary text-white p-2">First Component</div>`
})
export class FirstComponent { }
```

This component uses an inline template to display a message whose purpose is simply to make it clear which component has been selected by the routing system. Next, I created a file called `second.component.ts` in the `src/app/ondemand` folder and created the component shown in Listing 26-29.

Listing 26-29. The Contents of the second.component.ts File in the src/app/ondemand Folder

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

@Component({
  selector: "second",
  template: `<div class="bg-info text-white p-2">Second Component</div>`
})
export class SecondComponent { }
```

This component is almost identical to the one in Listing 26-28, differing only in the message that it displays through its inline template.

Creating Additional Outlet Elements

When you are using multiple outlet elements in the same template, Angular needs some way to tell them apart. This is done using the name attribute, which allows an outlet to be uniquely identified, as shown in Listing 26-30.

Listing 26-30. Adding Outlets in the ondemand.component.html File in the src/app/ondemand Folder

```
<div class="bg-primary text-white p-2">This is the ondemand component</div>
<div class="container-fluid">
  <div class="row">
    <div class="col-12 p-2">
      <router-outlet></router-outlet>
    </div>
  </div>
  <div class="row">
    <div class="col-6 p-2">
      <router-outlet name="left"></router-outlet>
    </div>
    <div class="col-6 p-2">
      <router-outlet name="right"></router-outlet>
    </div>
  </div>
</div>
<button class="btn btn-primary m-2" routerLink="/">Back</button>
```

The new elements create three new outlets. There can be at most one `router-outlet` element without a `name` element, which is known as the *primary outlet*. This is because omitting the `name` attribute has the same effect as applying it with a value of `primary`. All the routing examples so far in this book have relied on the primary outlet to display components to the user.

All other `router-outlet` elements must have a `name` element with a unique name. The names I have used in the listing are `left` and `right` because the classes applied to the `div` elements that contain the outlets use CSS to position these two outlets side by side.

The next step is to create a route that includes details of which component should be displayed in each outlet element, as shown in Listing 26-31. If Angular can't find a route that matches a specific outlet, then no content will be shown in that element.

Listing 26-31. Targeting Outlets in the ondemand.module.ts File in the src/app/ondemand Folder

```

import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { OndemandComponent } from "./ondemand.component";
import { RouterModule } from "@angular/router";
import { FirstComponent } from "./first.component";
import { SecondComponent } from "./second.component";

let routing = RouterModule.forChild([
  {
    path: "",
    component: OndemandComponent,
    children: [
      { path: "", children: [
        { outlet: "primary", path: "", component: FirstComponent, },
        { outlet: "left", path: "", component: SecondComponent, },
        { outlet: "right", path: "", component: SecondComponent, },
      ]},
    ],
  },
]);

```

```

@NgModule({
  imports: [CommonModule, routing],
  declarations: [OndemandComponent, FirstComponent, SecondComponent],
  exports: [OndemandComponent]
})
export class OndemandModule { }

```

The outlet property is used to specify the outlet element that the route applies to. The routing configuration in the listing matches the empty path for all three outlets and selects the newly created components for them: the primary outlet will display FirstComponent, and the left and right outlets will display SecondComponent, as shown in Figure 26-11. To see the effect yourself, click the Load Module button and click the Yes button when prompted. (If you don't see the expected content, reload the browser and try again.)

Tip If you omit the outlet property, then Angular assumes that the route targets the primary outlet. I tend to include the outlet property on all routes to emphasize which routes match an outlet element.

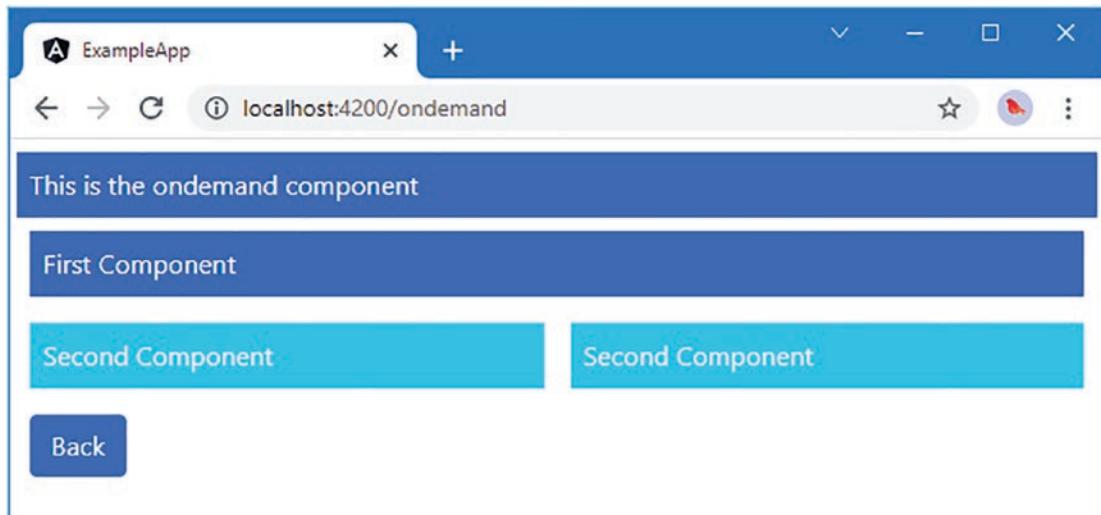


Figure 26-11. Using multiple router outlets

When Angular activates the route, it looks for matches for each outlet. All three of the new outlets have routes that match the empty path, which allows Angular to present the components shown in the figure.

Navigating When Using Multiple Outlets

Changing the components that are displayed by each outlet means creating a new set of routes and then navigating to the URL that contains them. Listing 26-32 sets up a route that will match the path /ondemand/swap and that will switch the components displayed by the three outlets.

Listing 26-32. Setting Routes for Outlets in the ondemand.module.ts File in the src/app/ondemand Folder

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { OndemandComponent } from "./ondemand.component";
import { RouterModule } from "@angular/router";
import { FirstComponent } from "./first.component";
import { SecondComponent } from "./second.component";
let routing = RouterModule.forChild([
  {
    path: "",
    component: OndemandComponent,
    children: [
      {
        path: "",
        children: [
          { outlet: "primary", path: "", component: FirstComponent, },
          { outlet: "left", path: "", component: SecondComponent, },
          { outlet: "right", path: "", component: SecondComponent, },
        ]
      },
    ],
  },
]);
```

```

    {
      path: "swap",
      children: [
        { outlet: "primary", path: "", component: SecondComponent, },
        { outlet: "left", path: "", component: FirstComponent, },
        { outlet: "right", path: "", component: FirstComponent, },
      ],
    },
  ],
),
@NgModule({
  imports: [CommonModule, routing],
  declarations: [OndemandComponent, FirstComponent, SecondComponent],
  exports: [OndemandComponent]
})
export class OndemandModule { }

```

Listing 26-33 adds button elements to the component's template that will navigate to the two sets of routes in Listing 26-32, alternating the set of components displayed to the user.

Listing 26-33. Navigating to Outlets in the ondemand.component.html File in the src/app/ondemand Folder

```

<div class="bg-primary text-white p-2">This is the ondemand component</div>
<div class="container-fluid">
  <div class="row">
    <div class="col-12 p-2">
      <router-outlet></router-outlet>
    </div>
  </div>
  <div class="row">
    <div class="col-6 p-2">
      <router-outlet name="left"></router-outlet>
    </div>
    <div class="col-6 p-2">
      <router-outlet name="right"></router-outlet>
    </div>
  </div>
</div>
<button class="btn btn-secondary m-2" routerLink="/ondemand">Normal</button>
<button class="btn btn-secondary m-2" routerLink="/ondemand/swap">Swap</button>
<button class="btn btn-primary m-2" routerLink="/">Back</button>

```

The result is that clicking the Swap and Normal buttons will navigate to routes whose children tell Angular which components should be displayed by each of the outlet elements, as illustrated by Figure 26-12.

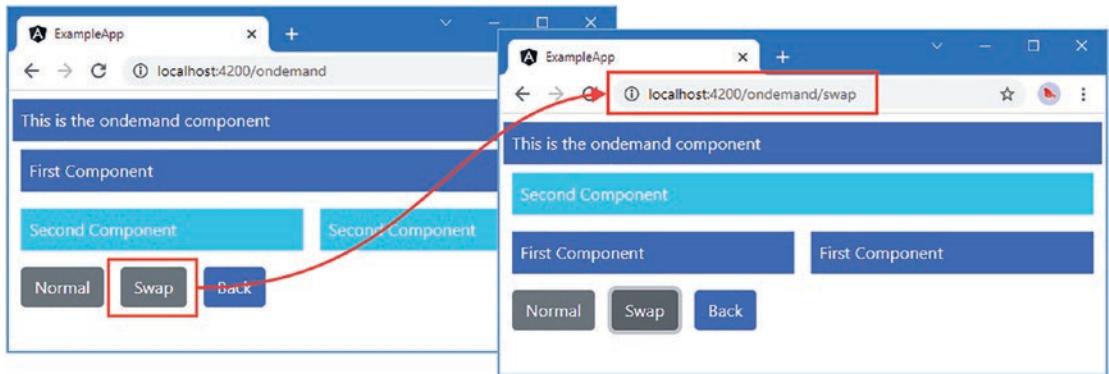


Figure 26-12. Using navigation to target multiple outlet elements

Summary

In this chapter, I finished describing the Angular URL routing features and explaining how to guard routes to control when a route is activated, how to load modules only when they are needed, and how to use multiple outlet elements to display components to the user. In the next chapter, I show you how to apply animations to Angular applications.

CHAPTER 27



Using Animations

In this chapter, I describe the Angular animation system, which uses data bindings to animate HTML elements to reflect changes in the state of the application. In broad terms, animations have two roles in an Angular application: to emphasize changes in content and to smooth them out.

Emphasizing changes is important when the content changes in a way that may not be obvious to the user. In the example application, using the Previous and Next buttons when editing a product changes the data fields but doesn't create any other visual change, which results in a transition that the user may not notice. Animations can be used to draw the eye to this kind of change, helping the user notice the results of an action.

Smoothing out changes can make an application more pleasant to use. When the user clicks the Edit button to start editing a product, the content displayed by the example application switches in a way that can be jarring. Using animations to slow down the transition can help provide a sense of context for the content change and make it less abrupt. In this chapter, I explain how the animation system works and how it can be used to draw the user's eye or take the edge off of sudden transitions. Table 27-1 puts Angular animations in context.

Table 27-1. Putting Angular Animations in Context

Question	Answer
What are they?	The animation system can change the appearance of HTML elements to reflect changes in the application state.
Why are they useful?	Used judiciously, animations can make applications more pleasant to use.
How are they used?	Animations are defined using functions defined in a platform-specific module, registered using the <code>animations</code> property in the <code>@Component</code> decorator and applied using a data binding.
Are there any pitfalls or limitations?	The main limitation is that Angular animations are fully supported by few browsers and, as a consequence, cannot be relied on to work properly on all the browsers that Angular supports for its other features.
Are there any alternatives?	The only alternative is not to animate the application.

Table 27-2 summarizes the chapter.

Table 27-2. Chapter Summary

Problem	Solution	Listing
Drawing the user's attention to a transition in the state of an element	Apply an animation	1-9
Animating the change from one element state to another	Use an element transition	9-14
Performing animations in parallel	Use animation groups	15
Using the same styles in multiple animations	Use common styles	16
Animating the position or size of elements	Use element transformations	17
Using animations to apply CSS framework styles	Use the DOM and CSS APIs	18-21

Preparing the Example Project

In this chapter, I continue using the exampleApp project that was first created in Chapter 20 and has been the focus of every chapter since. The changes in the following sections prepare the example application for the features described in this chapter.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Disabling the HTTP Delay

The first preparatory step for this chapter is to disable the delay added to asynchronous HTTP requests, as shown in Listing 27-1.

Listing 27-1. Disabling the Delay in the rest.datasource.ts File in the src/app/model Folder

```
import { Injectable, Inject, InjectionToken } from "@angular/core";
import { HttpClient, HttpHeaders } from "@angular/common/http";
import { catchError, delay, Observable } from "rxjs";
import { Product } from "./product.model";

export const REST_URL = new InjectionToken("rest_url");

@Injectable()
export class RestDataSource {
  constructor(private http: HttpClient,
    @Inject(REST_URL) private url: string) { }

  getData(): Observable<Product[]> {
    return this.sendRequest<Product[]>("GET", this.url); // .pipe(delay(5000));
  }
}
```

```

saveProduct(product: Product): Observable<Product> {
    return this.sendRequest<Product>("POST", this.url, product);
}

updateProduct(product: Product): Observable<Product> {
    return this.sendRequest<Product>("PUT",
        `${this.url}/${product.id}`, product);
}

deleteProduct(id: number): Observable<Product> {
    return this.sendRequest<Product>("DELETE", `${this.url}/${id}`);
}

private sendRequest<T>(verb: string, url: string, body?: Product)
: Observable<T> {

    let myHeaders = new HttpHeaders();
    myHeaders = myHeaders.set("Access-Key", "<secret>");
    myHeaders = myHeaders.set("Application-Names", ["exampleApp", "proAngular"]);

    return this.http.request<T>(verb, url, {
        body: body,
        headers: myHeaders
    }).pipe(catchError((error: Response) => {
        throw(`Network Error: ${error.statusText} (${error.status})`)
    }));
}
}
}

```

Simplifying the Table Template and Routing Configuration

Many of the examples in this chapter are applied to the elements in the table of products. The final preparation for this chapter is to simplify the template for the table component so that I can focus on a smaller amount of content in the listings.

Listing 27-2 shows the simplified template, which removes the buttons that generated HTTP and routing errors and the button and outlet element that counted the categories or products. The listing also removes the buttons that allow the table to be filtered by category.

Listing 27-2. Simplifying the Template in the table.component.html File in the src/app/core Folder

```

<table class="table table-sm table-bordered table-striped">
    <thead>
        <tr>
            <th>ID</th><th>Name</th><th>Category</th><th>Price</th>
            <th>Details</th><th></th>
        </tr>
    </thead>
    <tbody>
        <tr *ngFor="let item of getProducts()">
            <td>{{item.id}}</td>
            <td>{{item.name}}</td>

```

```

<td>{{item.category}}</td>
<td>{{item.price | currency:"USD" }}</td>
<td>
    <ng-container *ngIf="item.details else empty">
        {{ item.details?.supplier }}, {{ item.details?.keywords}}
    </ng-container>
    <ng-template #empty>(None)</ng-template>
</td>
<td class="text-center">
    <button class="btn btn-danger btn-sm m-1"
           (click)="deleteProduct(item.id)">
        Delete
    </button>
    <button class="btn btn-warning btn-sm"
           [routerLink]="/form", 'edit', item.id]">
        Edit
    </button>
</td>
</tr>
</tbody>
</table>

<div class="p-2 text-center">
    <button class="btn btn-primary m-1" routerLink="/form/create">
        Create New Product
    </button>
</div>

```

Listing 27-3 updates the URL routing configuration for the application so that the routes don't target the outlet element that has been removed from the table component's template.

Listing 27-3. Updating the Routing Configuration in the app.routing.ts File in the src/app Folder

```

import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { UnsavedGuard } from "./core/unsaved.guard";

const routes: Routes = [
{
    path: "form/:mode/:id", component: FormComponent,
    canDeactivate: [UnsavedGuard]
},
{ path: "form/:mode", component: FormComponent },
{ path: "table", component: TableComponent },
{ path: "table/:category", component: TableComponent },
{ path: "", redirectTo: "/table", pathMatch: "full" },
{ path: "**", component: NotFoundComponent }
]

export const routing = RouterModule.forRoot(routes);

```

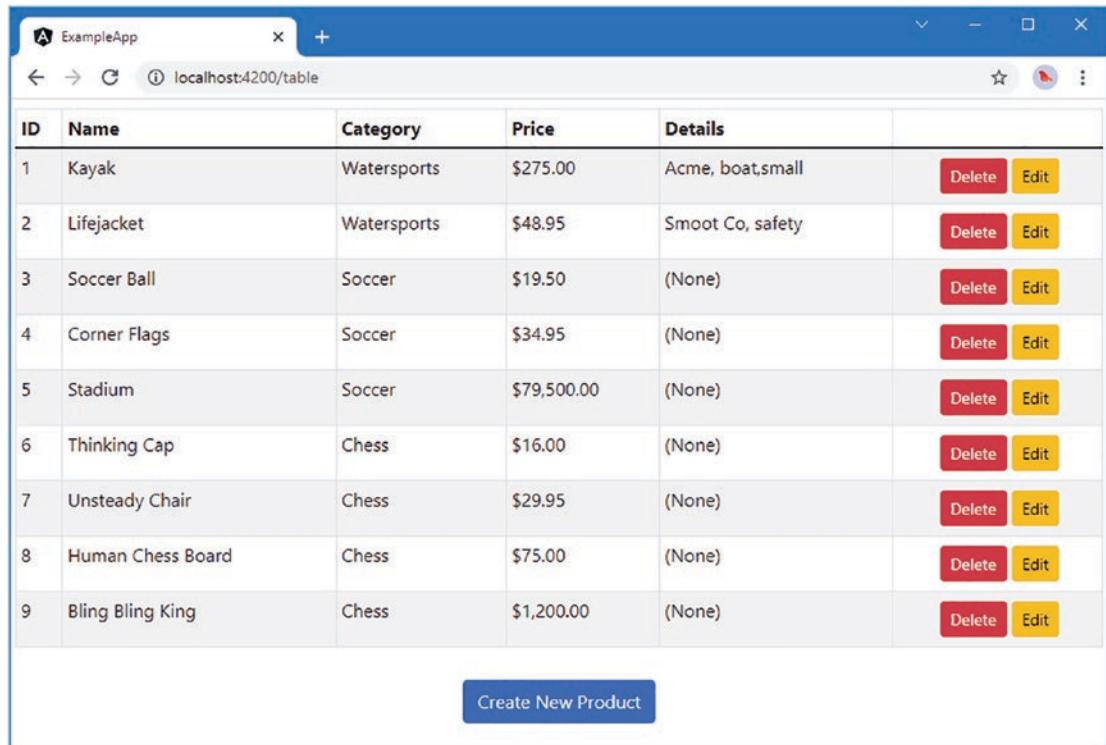
Open a new command prompt, navigate to the `exampleApp` folder, and run the following command to start the server that provides the RESTful web server:

```
npm run json
```

Open a separate command prompt, navigate to the `exampleApp` folder, and run the following command to start the Angular development tools:

```
ng serve
```

Open a new browser window and navigate to `http://localhost:4200` to see the content shown in Figure 27-1.



The screenshot shows a web browser window titled "ExampleApp" displaying a table of products at the URL "localhost:4200/table". The table has columns for ID, Name, Category, Price, and Details. Each row contains a "Delete" and "Edit" button. A "Create New Product" button is located at the bottom of the table.

ID	Name	Category	Price	Details	
1	Kayak	Watersports	\$275.00	Acme, boat, small	<button>Delete</button> <button>Edit</button>
2	Lifejacket	Watersports	\$48.95	Smoot Co, safety	<button>Delete</button> <button>Edit</button>
3	Soccer Ball	Soccer	\$19.50	(None)	<button>Delete</button> <button>Edit</button>
4	Corner Flags	Soccer	\$34.95	(None)	<button>Delete</button> <button>Edit</button>
5	Stadium	Soccer	\$79,500.00	(None)	<button>Delete</button> <button>Edit</button>
6	Thinking Cap	Chess	\$16.00	(None)	<button>Delete</button> <button>Edit</button>
7	Unsteady Chair	Chess	\$29.95	(None)	<button>Delete</button> <button>Edit</button>
8	Human Chess Board	Chess	\$75.00	(None)	<button>Delete</button> <button>Edit</button>
9	Bling Bling King	Chess	\$1,200.00	(None)	<button>Delete</button> <button>Edit</button>

[Create New Product](#)

Figure 27-1. Running the example application

Getting Started with Angular Animation

As with most Angular features, the best place to start is with an example, which will let me introduce how animation works and how it fits into the rest of the Angular functionality. In the sections that follow, I create a basic animation that will affect the rows in the table of products. Once you have seen how the basic features work, I will dive into the details of each of the different configuration options and explain how they work in depth.

But to get started, I am going to add a select element to the application that allows the user to select a category. When a category is selected, the table rows for products in that category will be shown in one of two styles, as described in Table 27-3.

Table 27-3. The Styles for the Animation Example

Description	Styles
The product is in the selected category.	The table row will have a green background and larger text.
The product is not in the selected category.	The table row will have a red background and smaller text.

Enabling the Animation Module

The animation features are contained in their own module that must be imported in the application's root module, as shown in Listing 27-4.

Listing 27-4. Importing the Animation Module in the app.module.ts File in the src/app Folder

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { ModelModule } from './model/model.module';
import { CoreModule } from './core/core.module';
import { TableComponent } from './core/table.component';
import { FormComponent } from './core/form.component';
import { MessageModule } from './messages/message.module';
import { MessageComponent } from './messages/message.component';
import { AppComponent } from './app.component';
import { routing } from './app.routing';
import { TermsGuard } from './terms.guard'
import { LoadGuard } from './load.guard';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations";

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, ModelModule, CoreModule, MessageModule, routing,
    BrowserAnimationsModule],
  providers: [TermsGuard, LoadGuard],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Creating the Animation

To get started with the animation, I created a file called table.animations.ts in the src/app/core folder and added the code shown in Listing 27-5.

[Listing 27-5.](#) The Contents of the table.animations.ts File in the src/app/core Folder

```
import { trigger, style, state, transition, animate } from "@angular/animations";

export const HighlightTrigger = trigger("rowHighlight", [
  state("selected", style({
    backgroundColor: "lightgreen",
    fontSize: "20px"
  })),
  state("notselected", style({
    backgroundColor: "lightsalmon",
    fontSize: "12px"
  })),
  transition("selected => notselected", animate("200ms")),
  transition("notselected => selected", animate("400ms"))
]);
```

The syntax used to define animations can be dense and relies on a set of functions defined in the @angular/animations module. In the following sections, I start at the top and work my way down through the details to explain each of the animation building blocks used in the listing.

Tip Don't worry if all the building blocks described in the following sections don't make immediate sense. This is an area of functionality that starts to make more sense only when you see how all the parts fit together.

Defining Style Groups

The heart of the animation system is the style group, which is a set of CSS style properties and values that will be applied to an HTML element. Style groups are defined using the `style` function, which accepts a JavaScript object literal that provides a map between property names and values, like this:

```
...
style({
  backgroundColor: "lightgreen",
  fontSize: "20px"
})
...
```

This style group tells Angular to set the background color to lightgreen and to set the font size to 20 pixels.

CSS PROPERTY NAME CONVENTIONS

There are two ways to specify CSS properties when using the `style` function. You can use the JavaScript property naming convention, such that the property to set the background color of an element is specified as `backgroundColor` (all one word, no hyphens, and subsequent words capitalized). This is the convention I used in Listing 27-5:

```
...
style({
  backgroundColor: "lightgreen",
  fontSize: "20px"
}),
...

```

Alternatively, you can use the CSS convention, where the same property is expressed as `background-color` (all lowercase with hyphens between words). If you use the CSS format, then you must enclose the property names in quotes to stop JavaScript from trying to interpret the hyphens as arithmetic operators, like this:

```
...
state("green", style({
  "background-colorfont-size

```

It doesn't matter which name convention you use, just as long as you are consistent. At the time of writing, Angular does not correctly apply styles if you mix and match property name conventions. To get consistent results, pick a naming convention and use it for all the style properties you set throughout your application.

Defining Element States

Angular needs to know when it needs to apply a set of styles to an element. This is done by defining an element state, which provides a name by which the set of styles can be referred. Element states are created using the `state` function, which accepts the name and the style set that should be associated with it. This is one of the two element states that are defined in Listing 27-5:

```
...
state("selected", style({
  backgroundColor: "lightgreen",
  fontSize: "20px"
}),
...

```

There are two states in the listing, called `selected` and `notselected`, which will correspond to whether the product described by a table row is in the category selected by the user.

Defining State Transitions

When an HTML element is in one of the states created using the `state` function, Angular will apply the CSS properties in the state's style group. The `transition` function is used to tell Angular how the new CSS properties should be applied. There are two transitions in Listing 27-5.

```
...
transition("selected => notselected", animate("200ms")),
transition("notselected => selected", animate("400ms"))
...

```

The first argument passed to the transition function tells Angular which states this instruction applies to. The argument is a string that specifies two states and an arrow that expresses the relationship between them. Two kinds of arrow are available, as described in Table 27-4.

Table 27-4. The Animation Transition Arrow Types

Arrow	Example	Description
=>	selected => notselected	This arrow specifies a one-way transition between two states, such as when the element moves from the selected state to the notselected state.
<=>	selected <=> notselected	This array specifies a two-way transition between two states, such as when the element moves from the selected state to the notselected state and from the notselected state to the selected state.

The transitions defined in Listing 27-5 use one-way arrows to tell Angular how it should respond when an element moves from the selected state to the notselected state and from the notselected state to the selected state.

The second argument to the transition function tells Angular what action it should take when the state change occurs. The animate function tells Angular to gradually transition between the properties defined in the CSS style set defined by two element states. The arguments passed to the animate function in Listing 27-5 specify the period of time that this gradual transition should take, either 200 milliseconds or 400 milliseconds.

GUIDANCE FOR APPLYING ANIMATIONS

Developers often get carried away when applying animations, resulting in applications that users find frustrating. Animations should be applied sparingly, they should be simple, and they should be quick. Use animations to help the user make sense of your application and not as a vehicle to demonstrate your artistic skills. Users, especially for corporate line-of-business applications, have to perform the same task repeatedly, and excessive and long animations just get in the way.

I suffer from this tendency, and, unchecked, my applications behave like Las Vegas slot machines. I have two rules that I follow to keep the problem under control. The first is that I perform the major tasks or workflows in the application 20 times in a row. In the case of the example application, that might mean creating 20 products and then editing 20 products. I remove or shorten any animation that I find myself having to wait to complete before I can move on to the next step in the process.

The second rule is that I don't disable animations during development. It can be tempting to comment out an animation when I am working on a feature because I will be performing a series of quick tests as I write the code. But any animation that gets in my way will also get in the user's way, so I leave the animations in place and adjust them—generally reducing their duration—until they become less obtrusive and annoying.

You don't have to follow my rules, of course, but it is important to make sure that the animations are helpful to the user and not a barrier to working quickly or a distracting annoyance.

Defining the Trigger

The final piece of plumbing is the animation trigger, which packages up the element states and transitions and assigns a name that can be used to apply the animation in a component. Triggers are created using the `trigger` function, like this:

```
...
export const HighlightTrigger = trigger("rowHighlight", [...])
...
```

The first argument is the name by which the trigger will be known, which is `rowHighlight` in this example, and the second argument is the array of states and transitions that will be available when the trigger is applied.

Applying the Animation

Once you have defined an animation, you can apply it to one or more components by using the `animations` property of the `@Component` decorator. Listing 27-6 applies the animation defined in Listing 27-5 to the table component and adds some additional features that are needed to support the animation.

Listing 27-6. Applying an Animation in the table.component.ts File in the src/app/core Folder

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { ActivatedRoute } from "@angular/router";
import { HighlightTrigger } from "./table.animations";

@Component({
  selector: "paTable",
  templateUrl: "table.component.html",
  animations: [HighlightTrigger]
})
export class TableComponent {
  category: string | null = null;

  constructor(public model: Model, activeRoute: ActivatedRoute) {
    activeRoute.params.subscribe(params => {
      this.category = params["category"] || null;
    })
  }

  getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  getProducts(): Product[] {
    return this.model.getProducts()
      .filter(p => this.category == null || p.category == this.category);
  }
}
```

```

get categories(): (string) [] {
    return (this.model.getProducts()
        .map(p => p.category)
        .filter((c, index, array) => c != undefined
            && array.indexOf(c) == index)) as string[];
}

deleteProduct(key?: number) {
    if (key != undefined) {
        this.model.deleteProduct(key);
    }
}

highlightCategory: string = "";

getRowState(category: string | undefined): string {
    return this.highlightCategory == "" ? "" :
        this.highlightCategory == category ? "selected" : "notselected";
}
}

```

The `animations` property is set to an array of triggers. You can define animations inline, but they can quickly become complex and make the entire component hard to read, which is why I used a separate file and exported a constant value from it, which I then assign to the `animations` property.

The other changes are to provide a mapping between the category selected by the user and the animation state that will be assigned to elements. The value of the `highlightCategory` property will be set using a `select` element and is used in the `getRowState` method to tell Angular which of the animation states defined in Listing 27-7 should be assigned based on a product category. If a product is in the selected category, then the method returns `selected`; otherwise, it returns `notselected`. If the user has not selected a category, then the empty string is returned.

The final step is to apply the animation to the component's template, telling Angular which elements are going to be animated, as shown in Listing 27-7. This listing also adds a `select` element that sets the value of the component's `highlightCategory` property using the `ngModel` binding.

Listing 27-7. Applying an Animation in the `table.component.html` File in the `src/app/core` Folder

```

<div class="form-group bg-info text-white p-2">
    <label>Category</label>
    <select [(ngModel)]="highlightCategory" class="form-control">
        <option value="">None</option>
        <option *ngFor="let category of categories">
            {{category}}
        </option>
    </select>
</div>

<table class="table table-sm table-bordered table-striped">
    <thead>
        <tr>
            <th>ID</th><th>Name</th><th>Category</th><th>Price</th>
            <th>Details</th><th></th>
        </tr>
    </thead>

```

```

<tbody>
  <tr *ngFor="let item of getProducts()"
      [@rowHighlight]="getRowState(item.category)">
    <td>{{item.id}}</td>
    <td>{{item.name}}</td>
    <td>{{item.category}}</td>
    <td>{{item.price | currency:"USD" }}</td>
    <td>
      <ng-container *ngIf="item.details else empty">
        {{ item.details?.supplier }}, {{ item.details?.keywords}}
      </ng-container>
      <ng-template #empty>(None)</ng-template>
    </td>
    <td class="text-center">
      <button class="btn btn-danger btn-sm m-1"
             (click)="deleteProduct(item.id)">
        Delete
      </button>
      <button class="btn btn-warning btn-sm"
             [routerLink]=["/form", 'edit', item.id]">
        Edit
      </button>
    </td>
  </tr>
</tbody>
</table>

<div class="p-2 text-center">
  <button class="btn btn-primary m-1" routerLink="/form/create">
    Create New Product
  </button>
</div>

```

Animations are applied to templates using special data bindings, which associate an animation trigger with an HTML element. The binding's target tells Angular which animation trigger to apply, and the binding's expression tells Angular how to work out which state an element should be assigned to, like this:

```

...
<tr *ngFor="let item of getProducts()" [@rowHighlight]="getRowState(item.category)">
...

```

The target of the binding is the name of the animation trigger, prefixed with the @ character, which denotes an animation binding. This binding tells Angular that it should apply the rowHighlight trigger to the tr element. The expression tells Angular that it should invoke the component's getRowState method to work out which state the element should be assigned to, using the item.category value as an argument. Figure 27-2 illustrates the anatomy of an animation data binding for quick reference.

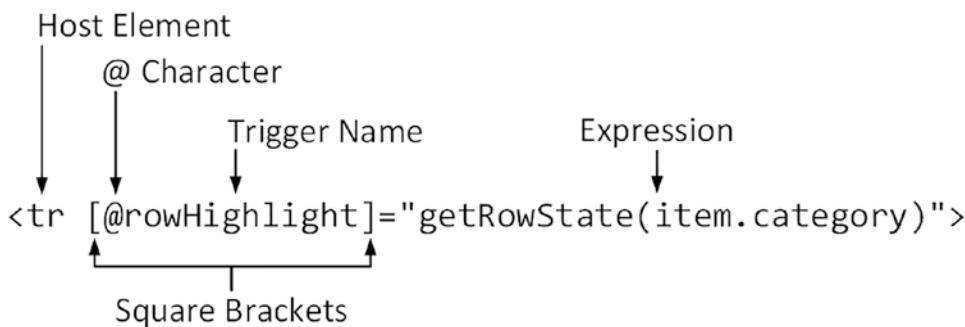


Figure 27-2. The anatomy of an animation data binding

Testing the Animation Effect

The changes in the previous section add a select element above the product table. To see the effect of the animation, restart the Angular development tools, request `http://localhost:4200`, and then select Soccer from the list at the top of the window. Angular will use the trigger to figure out which of the animation states each element should be applied to. Table rows for products in the Soccer category will be assigned to the selected state, while the other rows will be assigned to the notselected state, creating the effect shown in Figure 27-3.

Category						
ID	Name	Category	Price	Details	Delete	Edit
1	Kayak	Watersports	\$275.00	Acme, boats, small	<button>Delete</button>	<button>Edit</button>
2	Lifejacket	Watersports	\$48.95	Smooth Co, safety	<button>Delete</button>	<button>Edit</button>
3	Soccer Ball	Soccer	\$19.50	(None)	<button>Delete</button>	<button>Edit</button>
4	Corner Flags	Soccer	\$34.95	(None)	<button>Delete</button>	<button>Edit</button>
5	Stadium	Soccer	\$79,500.00	(None)	<button>Delete</button>	<button>Edit</button>
6	Thinking Cap	Chess	\$16.00	(None)	<button>Delete</button>	<button>Edit</button>
7	Unsteady Chair	Chess	\$29.95	(None)	<button>Delete</button>	<button>Edit</button>
8	Human Chess Board	Chess	\$75.00	(None)	<button>Delete</button>	<button>Edit</button>
9	Bling Bling King	Chess	\$1,200.00	(None)	<button>Delete</button>	<button>Edit</button>

[Create New Product](#)

Figure 27-3. Selecting a product category

The new styles are applied suddenly. To see a smoother transition, select the Chess category from the list, and you will see a gradual animation as the Chess rows are assigned to the selected state and the other rows are assigned to the notselected state. This happens because the animation trigger contains transitions between these states that tell Angular to animate the change in CSS styles, as illustrated in Figure 27-4. There is no transition for the earlier change, so Angular defaults to applying the new styles immediately.

Tip It is impossible to capture the effect of animations in a series of screenshots, and the best I can do is present some of the intermediate states. This is a feature that requires firsthand experimentation to understand. I encourage you to download the project for this chapter from GitHub and create your own animations.

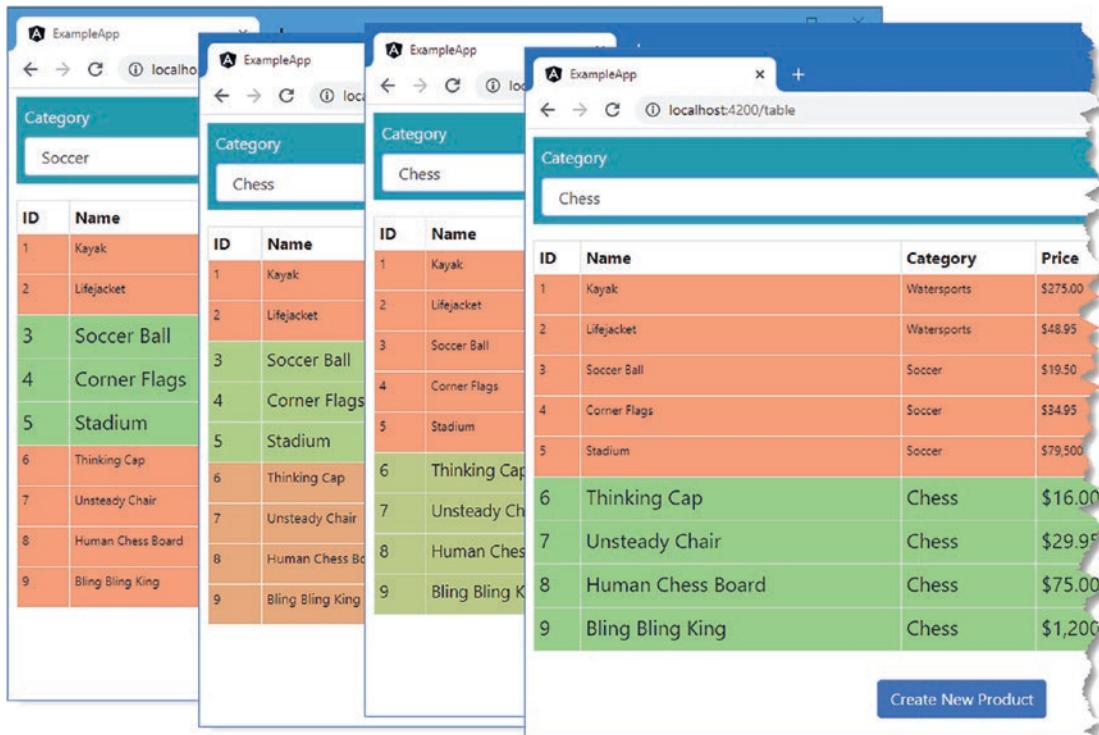


Figure 27-4. A gradual transition between animation states

To understand the Angular animation system, you need to understand the relationship between the different building blocks used to define and apply an animation, which can be described like this:

1. Evaluating the data binding expression tells Angular which animation state the host element is assigned to.
2. The data binding target tells Angular which animation target defines CSS styles for the element's state.

3. The state tells Angular which CSS styles should be applied to the element.
4. The transition tells Angular how it should apply CSS styles when evaluating the data binding expression results in a change to the element's state.

Keep these four points in mind as you read through the rest of the chapter, and you will find the animation system easier to understand.

Understanding the Built-in Animation States

Animation states are used to define the end result of an animation, grouping together the styles that should be applied to an element with a name that can be selected by an animation trigger. There are two built-in states that Angular provides that make it easier to manage the appearance of elements, as described in Table 27-5.

Table 27-5. The Built-in Animation States

State	Description
*	This is a fallback state that will be applied if the element isn't in any of the other states defined by the animation trigger.
void	Elements are in the void state when they are not part of the template. When the expression for an <code>ngIf</code> directive evaluates as <code>false</code> , for example, the host element is in the void state. This state is used to animate the addition and removal of elements, as described in the next section.

An asterisk (the * character) is used to denote a special state that Angular should apply to elements that are not in any of the other states defined by an animation trigger. Listing 27-8 adds the fallback state to the animations in the example application.

Listing 27-8. Using the Fallback State in the table.animations.ts File in the src/app/core Folder

```
import { trigger, style, state, transition, animate } from "@angular/animations";

export const HighlightTrigger = trigger("rowHighlight", [
  state("selected", style({
    backgroundColor: "lightgreen",
    fontSize: "20px"
  })),
  state("notselected", style({
    backgroundColor: "lightsalmon",
    fontSize: "12px"
  })),
  state("*", style({
    border: "solid black 2px"
  })),
  transition("selected => notselected", animate("200ms")),
  transition("notselected => selected", animate("400ms"))
]);
```

In the example application, elements are assigned only to the selected or notselected state once the user has picked a value with the select element. The fallback state defines a style group that will be applied to elements until they are entered into one of the other states, as shown in Figure 27-5.

ID	Name	Category	Price	Details	
1	Kayak	Watersports	\$275.00	Acme, boat, small	<button>Delete</button> <button>Edit</button>
2	Lifejacket	Watersports	\$48.95	Smooth Co, safety	<button>Delete</button> <button>Edit</button>
3	Soccer Ball	Soccer	\$19.50	(None)	<button>Delete</button> <button>Edit</button>

Figure 27-5. Using the fallback state

Understanding Element Transitions

The transitions are the real power of the animation system; they tell Angular how it should manage the change from one state to another. In the sections that follow, I describe different ways in which transitions can be created and used.

Creating Transitions for the Built-in States

The built-in states described in Table 27-5 can be used in transitions. The fallback state can be used to simplify the animation configuration by representing any state, as shown in Listing 27-9.

Listing 27-9. Using the Fallback State in the table.animations.ts File in the src/app/core Folder

```
import { trigger, style, state, transition, animate } from "@angular/animations";

export const HighlightTrigger = trigger("rowHighlight", [
  state("selected", style({
    backgroundColor: "lightgreen",
    fontSize: "20px"
  })),
  state("notselected", style({
    backgroundColor: "lightsalmon",
    fontSize: "12px"
  })),
  state("*", style({
    border: "solid black 2px"
  })),
]);
```

```
transition("* => notselected", animate("200ms")),
transition("* => selected", animate("400ms"))
]);
```

The transitions in the listing tell Angular how to deal with the change from any state into the notselected and selected states.

Animating Element Addition and Removal

The void state is used to define transitions for when an element is added to or removed from the template, as shown in Listing 27-10.

Listing 27-10. Using the Void State in the table.animations.ts File in the src/app/core Folder

```
import { trigger, style, state, transition, animate } from "@angular/animations";

export const HighlightTrigger = trigger("rowHighlight", [
  state("selected", style({
    backgroundColor: "lightgreen",
    fontSize: "20px"
  })),
  state("notselected", style({
    backgroundColor: "lightsalmon",
    fontSize: "12px"
  })),
  state("void", style({
    opacity: 0
  })),
  transition("* => notselected", animate("200ms")),
  transition("* => selected", animate("400ms")),
  transition("void => *", animate("500ms"))
]);
```

This listing includes a definition for the void state that sets the opacity property to zero, which makes the element transparent and, as a consequence, invisible. There is also a transition that tells Angular to animate the change from the void state to any other state. The effect is that the rows in the table fade into view as the browser gradually increases the opacity value until the fill opacity is reached, as shown in Figure 27-6.

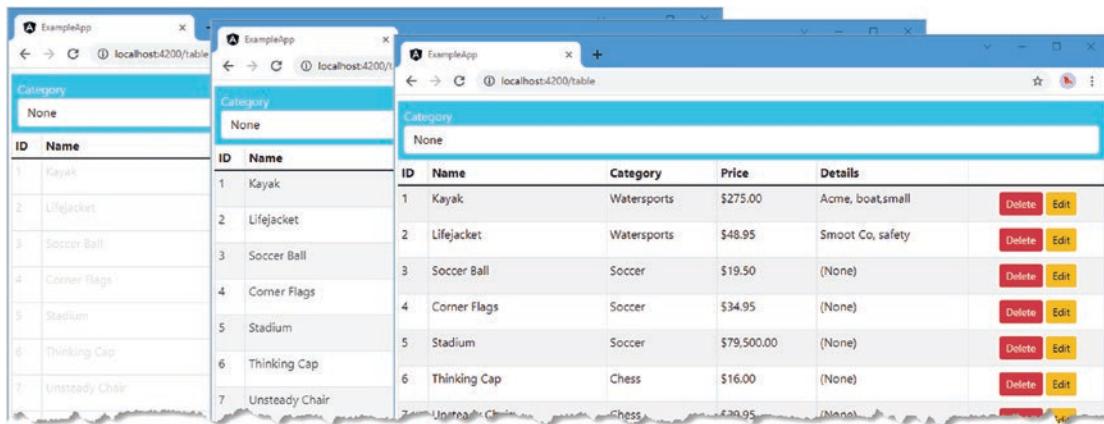


Figure 27-6. Animating element addition

Controlling Transition Animations

All the examples so far in this chapter have used the `animate` function in its simplest form, which is to specify how long a transition between two states should take, like this:

```
...
transition("void => *", animate("500ms"))
...
```

The string argument passed to the `animate` method can be used to exercise finer-grained control over the way that transitions are animated by providing an initial delay and specifying how intermediate values for the style properties are calculated.

EXPRESSING ANIMATION DURATIONS

Durations for animations are expressed using CSS time values, which are string values containing one or more numbers followed by either `s` for seconds or `ms` for milliseconds. This value, for example, specifies a duration of 500 milliseconds:

```
...
transition("void => *", animate("500ms"))
...
```

Durations are expressed flexibly, and the same value could be expressed as a fraction of a second, like this:

```
...
transition("void => *", animate("0.5s"))
...
```

My advice is to stick to one set of units throughout a project to avoid confusion, although it doesn't matter which one you use.

Specifying a Timing Function

The timing function is responsible for calculating the intermediate values for CSS properties during the transition. The timing functions, which are defined as part of the Web Animations specification, are described in Table 27-6.

Table 27-6. The Animation Timing Functions

Name	Description
linear	This function changes the value in equal amounts. This is the default.
ease-in	This function starts with small changes that increase over time, resulting in an animation that starts slowly and speeds up.
ease-out	This function starts with large changes that decrease over time, resulting in an animation that starts quickly and then slows down.
ease-in-out	This function starts with large changes that become smaller until the midway point, after which they become larger again. The result is an animation that starts quickly, slows down in the middle, and then speeds up again at the end.
cubic-bezier	This function is used to create intermediate values using a Bezier curve. See http://w3c.github.io/web-animations/#time-transformations for details.

Listing 27-11 applies a timing function to one of the transitions in the example application. The timing function is specified after the duration in the argument to the `animate` function.

Listing 27-11. Applying a Timing Function in the `table.animations.ts` File in the `src/app/core` Folder

```
import { trigger, style, state, transition, animate } from "@angular/animations";

export const HighlightTrigger = trigger("rowHighlight", [
  state("selected", style({
    backgroundColor: "lightgreen",
    fontSize: "20px"
  })),
  state("notselected", style({
    backgroundColor: "lightsalmon",
    fontSize: "12px"
  })),
  state("void", style({
    opacity: 0
  })),
  transition("* => notselected", animate("200ms")),
  transition("* => selected", animate("400ms ease-in")),
  transition("void => *", animate("500ms"))
]);
```

Specifying an Initial Delay

An initial delay can be provided to the `animate` method, which can be used to stagger animations when there are multiple transitions being performed simultaneously. The delay is specified as the second value in the argument passed to the `animate` function, as shown in Listing 27-12.

Listing 27-12. Adding an Initial Delay in the `table.animations.ts` File in the `src/app/core` Folder

```
import { trigger, style, state, transition, animate } from "@angular/animations";

export const HighlightTrigger = trigger("rowHighlight", [
  state("selected", style({
    backgroundColor: "lightgreen",
    fontSize: "20px"
  })),
  state("notselected", style({
    backgroundColor: "lightsalmon",
    fontSize: "12px"
  })),
  state("void", style({
    opacity: 0
  })),
  transition("* => notselected", animate("200ms")),
  transition("* => selected", animate("400ms 200ms ease-in")),
  transition("void => *", animate("500ms"))
]);
```

The 200-millisecond delay in this example corresponds to the duration of the animation used when an element transitions to the `notselected` state. The effect is that changing the `selected` category will show elements returning to the `notselected` state before the `selected` elements are changed.

Using Additional Styles During Transition

The `animate` function can accept a style group as its second argument, as shown in Listing 27-13. These styles are applied to the host element gradually, over the duration of the animation.

Listing 27-13. Defining Transition Styles in the `table.animations.ts` File in the `src/app/core` Folder

```
import { trigger, style, state, transition, animate } from "@angular/animations";

export const HighlightTrigger = trigger("rowHighlight", [
  state("selected", style({
    backgroundColor: "lightgreen",
    fontSize: "20px"
  })),
  state("notselected", style({
    backgroundColor: "lightsalmon",
    fontSize: "12px"
  })),
  state("void", style({
    opacity: 0
  })),
]);
```

```

transition("* => notselected", animate("200ms")),
transition("* => selected",
  animate("400ms 200ms ease-in",
    style({
      backgroundColor: "lightblue",
      fontSize: "25px"
    })
  ),
  transition("void => *", animate("500ms"))
]);

```

The effect of this change is that when an element is transitioning into the selected state, its appearance will be animated so that the background color will be lightblue and its font size will be 25 pixels. At the end of the animation, the styles defined by the selected state will be applied all at once, creating a snap effect.

The sudden change in appearance at the end of the animation can be jarring. An alternative approach is to change the second argument of the transition function to an array of animations. This defines multiple animations that will be applied to the element in sequence, and as long as it doesn't define a style group, the final animation will be used to transition to the styles defined by the state. Listing 27-14 uses this feature to add two animations to the transition, the last of which will apply the styles defined by the selected state.

Listing 27-14. Using Multiple Animations in the table.animations.ts File in the src/app/core Folder

```

import { trigger, style, state, transition, animate } from "@angular/animations";

export const HighlightTrigger = trigger("rowHighlight", [
  state("selected", style({
    backgroundColor: "lightgreen",
    fontSize: "20px"
  })),
  state("notselected", style({
    backgroundColor: "lightsalmon",
    fontSize: "12px"
  })),
  state("void", style({
    opacity: 0
  })),
  transition("* => notselected", animate("200ms")),
  transition("* => selected",
    [animate("400ms 200ms ease-in",
      style({
        backgroundColor: "lightblue",
        fontSize: "25px"
      }),
      animate("250ms", style({
        backgroundColor: "lightcoral",
        fontSize: "30px"
      })),
      animate("200ms")]
  ),
  transition("void => *", animate("500ms"))
]);

```

There are three animations in this transition, and the last one will apply the styles defined by the selected state. Table 27-7 describes the sequence of animations.

Table 27-7. The Sequence of Animations in the Transition to the selected State

Duration	Style Properties and Values
400 milliseconds	backgroundColor: lightblue; fontSize: 25px
250 milliseconds	backgroundColor: lightcoral; fontSize: 30px
200 milliseconds	backgroundColor: lightgreen; fontSize: 20px

Pick a category using the select element to see the sequence of animations. Figure 27-7 shows one frame from each animation.

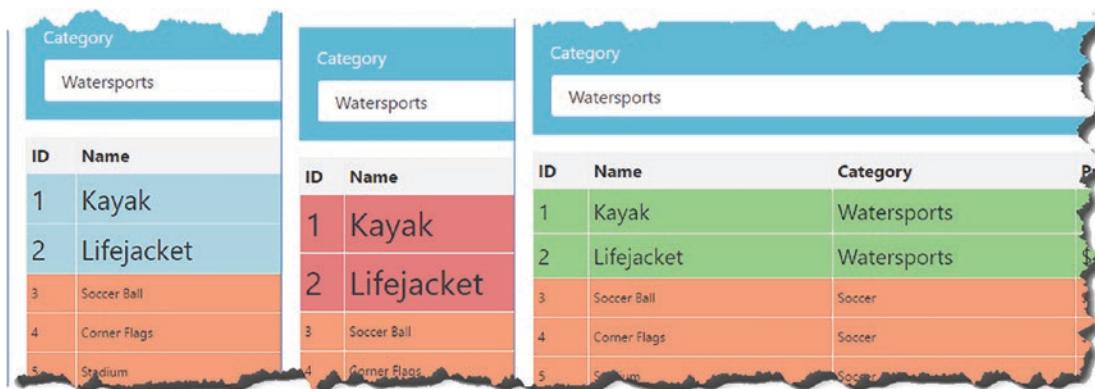


Figure 27-7. Using multiple animations in a transition

Performing Parallel Animations

Angular can perform animations at the same time, which means you can have different CSS properties change over different time periods. Parallel animations are passed to the group function, as shown in Listing 27-15.

Listing 27-15. Performing Parallel Animations in the table.animations.ts File in the src/app/core Folder

```
import { trigger, style, state, transition, animate, group }
  from "@angular/animations";

export const HighlightTrigger = trigger("rowHighlight", [
  state("selected", style({
    backgroundColor: "lightgreen",
    fontSize: "20px"
  })),
  group([
    transition(":enter", [animate("0.2s ease-in")]),
    transition(":leave", [animate("0.2s ease-out")])
  ])
]);
```

```

state("notselected", style({
  backgroundColor: "lightsalmon",
  fontSize: "12px"
))),
state("void", style({
  opacity: 0
))),
transition("* => notselected", animate("200ms")),
transition("* => selected",
  [animate("400ms 200ms ease-in",
    style({
      backgroundColor: "lightblue",
      fontSize: "25px"
    })),
  group([
    animate("250ms", style({
      backgroundColor: "lightcoral",
    })),
    animate("450ms", style({
      fontSize: "30px"
    })),
  ]),
  animate("200ms")]
),
transition("void => *", animate("500ms"))
]);

```

The listing replaces one of the animations in sequence with a pair of parallel animations. The animations for the `backgroundColor` and `fontSize` properties will be started at the same time but last for differing durations. When both of the animations in the group have completed, Angular will move on to the final animation, which will target the styles defined in the state.

Understanding Animation Style Groups

The outcome of an Angular animation is that an element is put into a new state and styled using the properties and values in the associated style group. In this section, I explain some different ways in which style groups can be used.

Tip Not all CSS properties can be animated, and of those that can be animated, some are handled better by the browser than others. As a rule of thumb, the best results are achieved with properties whose values can be easily interpolated, which allows the browser to provide a smooth transition between element states. This means you will usually get good results using properties whose values are colors or numerical values, such as background, text and font colors, opacity, element sizes, and borders. See <https://www.w3.org/TR/css3-transitions/#animatable-properties> for a complete list of properties that can be used with the animation system.

Defining Common Styles in Reusable Groups

As you create more complex animations and apply them throughout your application, you will inevitably find that you need to apply some common CSS property values in multiple places. The `style` function can accept an array of objects, all of which are combined to create the overall set of styles in the group. This means you can reduce duplication by defining objects that contain common styles and use them in multiple style groups, as shown in Listing 27-16. (To keep the example simple, I have also removed the sequence of styles defined in the previous section.)

Listing 27-16. Defining Common Styles in the `table.animations.ts` File in the `src/app/core` Folder

```
import { trigger, style, state, transition, animate, group }
from "@angular/animations";

const commonStyles = {
  border: "black solid 4px",
  color: "white"
};

export const HighlightTrigger = trigger("rowHighlight", [
  state("selected", style([commonStyles, {
    backgroundColor: "lightgreen",
    fontSize: "20px"
}])),
  state("notselected", style([commonStyles, {
    backgroundColor: "lightsalmon",
    fontSize: "12px",
    color: "black"
}])),
  state("void", style({
    opacity: 0
})),
  transition("* => notselected", animate("200ms")),
  transition("* => selected", animate("400ms 200ms ease-in")),
  transition("void => *", animate("500ms"))
]);
```

The `commonStyles` object defines values for the `border` and `color` properties and is passed to the `style` function in an array along with the regular style objects. Angular processes the style objects in order, which means you can override a style value by redefining it in a later object. As an example, the second style object for the `notselected` state overrides the common value for the `color` property with a custom value. The result is that the styles for both animation states incorporate the common value for the `border` property, and the styles for the `selected` state also use the common value for the `color` property, as shown in Figure 27-8.

ID	Name	Category	Price	Details	
1	Kayak	Watersports	\$275.00	Acme, boat,small	<button>Delete</button> <button>Edit</button>
2	Lifejacket	Watersports	\$48.95	Smoot Co, safety	<button>Delete</button> <button>Edit</button>
3	Soccer Ball	Soccer	\$19.50	(None)	<button>Delete</button> <button>Edit</button>
4	Corner Flags	Soccer	\$34.95	(None)	<button>Delete</button> <button>Edit</button>
5	Stamps	Sports	\$70.00	(None)	<button>Delete</button> <button>Edit</button>

Figure 27-8. Defining common properties

Using Element Transformations

All the examples so far in this chapter have animated properties that have affected an aspect of an element's appearance, such as background color, font size, or opacity. Animations can also be used to apply CSS element transformation effects, which are used to move, resize, rotate, or skew an element. These effects are applied by defining a `transform` property in a style group, as shown in Listing 27-17.

Listing 27-17. Using an Element Transformation in the `table.animations.ts` File in the `src/app/core` Folder

```
import { trigger, style, state, transition, animate, group } from "@angular/animations";

const commonStyles = {
  border: "black solid 4px",
  color: "white"
};

export const HighlightTrigger = trigger("rowHighlight", [
  state("selected", style([commonStyles, {
    backgroundColor: "lightgreen",
    fontSize: "20px"
  }])),
  state("notselected", style([commonStyles, {
    backgroundColor: "lightsalmon",
    fontSize: "12px",
    color: "black"
  }])),
  state("void", style({
    transform: "translateX(-50%)"
  })),
]);
```

```
transition("* => notselected", animate("200ms")),
transition("* => selected", animate("400ms 200ms ease-in")),
transition("void => *", animate("500ms"))
]);
```

The value of the `transform` property is `translateX(50%)`, which tells Angular to move the element 50 percent of its length along the x-axis. The `transform` property has been applied to the `void` state, which means that it will be used on elements as they are being added to the template. The animation contains a transition from the `void` state to any other state and tells Angular to animate the changes over 500 milliseconds. The result is that new elements will be shifted to the left initially and then slid back into their default position over a period of half a second, as illustrated in Figure 27-9.

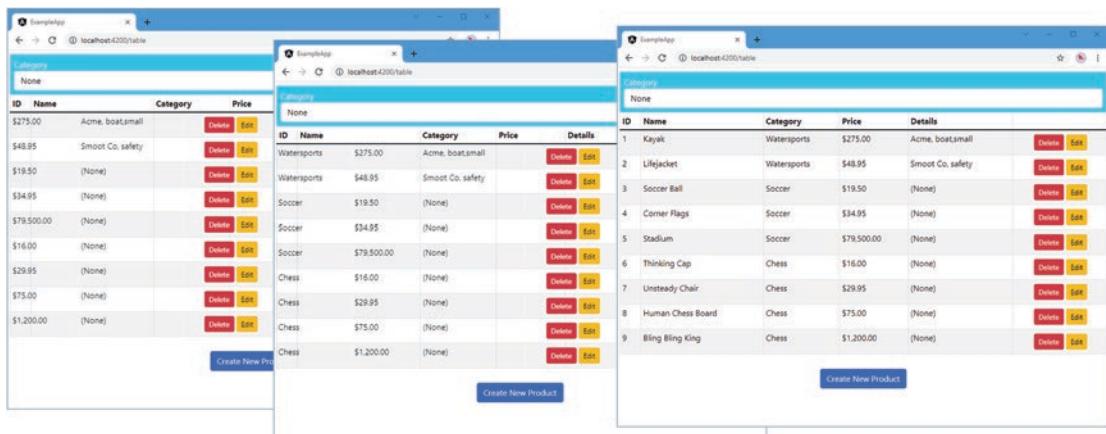


Figure 27-9. Transforming an element

Table 27-8 describes the set of transformations that can be applied to elements.

Tip Multiple transformations can be applied in a single `transform` property by separating them with spaces, like this: `transform: "scale(1.1, 1.1) rotate(10deg)"`.

Table 27-8. The CSS Transformation Functions

Function	Description
<code>translateX(offset)</code>	This function moves the element along the x-axis. The amount of movement can be specified as a percentage or as a length (expressed in pixels or one of the other CSS length units). Positive values translate the element to the right, negative values to the left.
<code>translateY(offset)</code>	This function moves the element along the y-axis.
<code>translate(xOffset, yOffset)</code>	This function moves the element along both axes.

(continued)

Table 27-8. (continued)

Function	Description
scaleX(amount)	This function scales the element along the x-axis. The scaling size is expressed as a fraction of the element's regular size, such that 0.5 reduces the element to 50 percent of the original width and 2.0 will double the width.
scaleY(amount)	This function scales the element along the y-axis.
scale(xAmount, yAmount)	This function scales the element along both axes.
rotate(angle)	This function rotates the element clockwise. The amount of rotation is expressed as an angle, such as 90deg or 3.14rad.
skewX(angle)	This function skews the element along the x-axis by a specified angle, expressed in the same way as for the rotate function.
skewY(angle)	This function skews the element along the y-axis by a specified angle, expressed in the same way as for the rotate function.
skew(xAngle, yAngle)	This function skews the element along both axes.

Applying CSS Framework Styles

If you are using a CSS framework like Bootstrap, you may want to apply classes to elements, rather than having to define groups of properties. There is no built-in support for working directly with CSS classes, but the Document Object Model (DOM) and the CSS Object Model (CSSOM) provide API access to inspect the CSS stylesheets that have been loaded and to see whether they apply to an HTML element. To get the set of styles defined by classes, I created a file called `animationUtils.ts` to the `src/app/core` folder and added the code shown in Listing 27-18.

Caution This technique can require substantial processing in an application that uses a lot of complex stylesheets, and you may need to adjust the code to work with different browsers and different CSS frameworks.

Listing 27-18. The Contents of the `animationUtils.ts` File in the `src/app/core` Folder

```
export const stateClassMap : {[key: string]: string[] | string} = {
    selected: ["table-success", "h2"],
    notselected: ["table-info"]
};

export function getStylesFromClasses(names: string | string[],
    elementType: string = "div") : { [key: string]: string | number } {
    return findStylesOrProps(names, elementType, (name) => !name.startsWith("--"))
}

export function setPropertiesFromClasses(state: string, target: HTMLElement) {
    let props = findStylesOrProps(stateClassMap[state], "div",
        (name) => name.startsWith("--"));
}
```

```

Object.keys(props).forEach(k => {
    target.style.setProperty(k, props[k]);
})
}

function findStylesOrProps(names: string | string[], elementType: string,
    selector: (name: string) => boolean) : { [key: string]: string } {

let elem = document.createElement(elementType);
(typeof names == "string" ? [names] : names)
    .forEach(c => elem.classList.add(c));

let result : { [key: string]: string } = {};

for (let i = 0; i < document.styleSheets.length; i++) {
    let sheet = document.styleSheets[i] as CSSStyleSheet;
    let rules = sheet.cssRules || sheet.cssRules;
    for (let j = 0; j < rules.length; j++) {
        if (rules[j] instanceof CSSStyleRule) {
            let styleRule = rules[j] as CSSStyleRule;
            if (elem.matches(styleRule.selectorText)) {
                for (let k = 0; k < styleRule.style.length; k++) {
                    let name = styleRule.style[k];
                    if (selector(name)) {
                        result[name] = styleRule.style.getPropertyValue(name);
                    }
                }
            }
        }
    }
}
return result;
}

```

The `getStylesFromClass` method accepts a single class name or an array of class names and the element type to which they should be applied, which defaults to a `div` element. An element is created and assigned to the classes and then inspected to see which of the CSS rules defined in the CSS stylesheets apply to it. One complication of applying the Bootstrap styles in an animation is they rely on custom CSS properties, which are not supported for animation. To work around this issue, the `getStylesFromClasses` method skips styles whose name begins with `--`, and the `setPropertiesFromClasses` is used to set these properties on the element to be animated. This is an inefficient and error-prone approach, but it is the best that I have been able to find. I hope future releases of Angular will address this problem directly.

The style properties for each matching style are added to an object that can be used to create Angular animation style groups, as shown in Listing 27-19.

Listing 27-19. Using Bootstrap Classes in the `table.animations.ts` File in the `src/app/core` Folder

```

import { trigger, style, state, transition, animate, group }
    from "@angular/animations";
import { getStylesFromClasses, stateClassMap } from "./animationUtils";

```

```
// const commonStyles = {
//   border: "black solid 4px",
//   color: "white"
// };

export const HighlightTrigger = trigger("rowHighlight", [
  state("selected", style(getStylesFromClasses(stateClassMap["selected"]))),
  state("notselected", style(getStylesFromClasses(stateClassMap["notselected"]))),
  state("void", style({
    transform: "translateX(-50%)"
  })),
  transition("* => notselected", animate("200ms")),
  transition("* => selected", animate("400ms 200ms ease-in")),
  transition("void => *", animate("500ms"))
]);
```

The selected state and unselected states use the styles defined in Listing 27-18. To ensure that the custom properties are set correctly, Listing 27-20 updates the table component.

Listing 27-20. Setting Custom Properties in the table.component.ts File in the src/app/core Folder

```
import { Component, ElementRef } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { ActivatedRoute } from "@angular/router";
import { HighlightTrigger } from "./table.animations";
import { setPropertiesFromClasses, stateClassMap } from "./animationUtils";

@Component({
  selector: "paTable",
  templateUrl: "table.component.html",
  animations: [HighlightTrigger]
})
export class TableComponent {
  category: string | null = null;

  constructor(public model: Model, activeRoute: ActivatedRoute) {
    activeRoute.params.subscribe(params => {
      this.category = params["category"] || null;
    })
  }

  getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  getProducts(): Product[] {
    return this.model.getProducts()
      .filter(p => this.category == null || p.category == this.category);
  }
}
```

```

get categories(): string[] {
    return (this.model.getProducts()
        .map(p => p.category)
        .filter((c, index, array) => c != undefined
            && array.indexOf(c) == index)) as string[];
}

deleteProduct(key?: number) {
    if (key != undefined) {
        this.model.deleteProduct(key);
    }
}

highlightCategory: string = "";

getRowState(category: string | undefined, elem: HTMLTableRowElement): string {
    let state = this.highlightCategory == "" ? "" :
        this.highlightCategory == category ? "selected" : "notselected"
    if (state != "") {
        setPropertiesFromClasses(state, elem);
    }
    return state;
}
}

```

The `getRowState` method has been modified to receive the element that is being modified. Listing 27-21 updates the template to provide the additional argument using a template variable.

Listing 27-21. Adding a Template Variable in the `table.component.html` File in the `src/app/core` Folder

```

...
<tbody>
    <tr *ngFor="let item of getProducts()" #elem
        [@rowHighlight]="getRowState(item.category, elem)">
        <td>{{item.id}}</td>
        <td>{{item.name}}</td>
...

```

The effect is that the Bootstrap styles are applied to the rows in the table, based on the selected category, as shown in Figure 27-10.

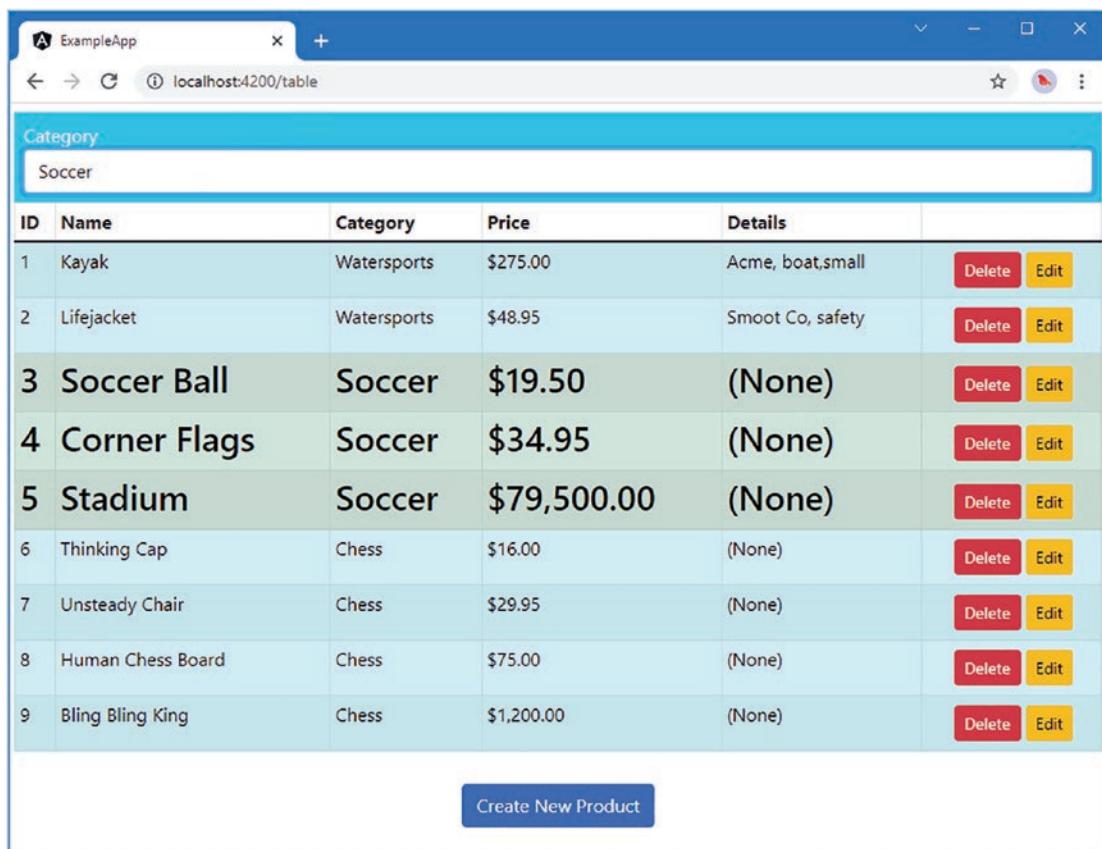


Figure 27-10. Using CSS framework styles in Angular animations

Summary

I described the Angular animation system in this chapter and explained how it uses data bindings to animate changes in the application's state. In the next chapter, I describe the features that Angular provides to support unit testing.

CHAPTER 28



Working with Component Libraries

Component libraries are packages that contain Angular components and directives, such as buttons, tables, and layouts. Throughout this book, I have been creating custom components and directives to demonstrate Angular features, but component libraries use these same features to provide building blocks that you can use to simplify the development process.

One of the recurring themes in this book is that nothing in Angular is magic, and this extends to component libraries, which are written using the same features that you used in earlier chapters. Component libraries are useful because they mean you don't have to write code and templates for basic tasks, such as creating a button, for example, and can focus on dealing with what happens when the user clicks the button.

In this chapter, I use the Angular Material component library to add components to the project and explain how to use CSS to give a custom component an appearance that is consistent with the library components. Table 28-1 puts the use of component libraries in context.

Note This chapter is not a detailed description of Angular Material or any other component library. There are several good component libraries available for Angular and each has its own set of features and its own API.

Table 28-1. Putting Component Libraries in Context

Question	Answer
What are they?	Component libraries are packages containing commonly required user interface features for Angular applications.
Why are they useful?	Component libraries can speed up project development and ensure a consistent appearance in the finished application.
How are they used?	Features are presented as Angular components or directives, which are applied in the same way as custom components and directives.
Are there any pitfalls or limitations?	Component libraries can require data to be presented in a specific way or for the application to be structured using a specific pattern. These restrictions may not suit all projects.
Are there any alternatives?	Component libraries are entirely optional and are not required for Angular development.

Table 28-2 summarizes the chapter.

Table 28-2. Chapter Summary

Problem	Solution	Listing
Applying the features provided by a component library	Use the components or directives provided contained in the library package	4-11
Using the advanced features provided by a component library	Adopt the structure or API that the component library provides for integration	12-15
Styling custom components to match the theme used by the component library	Use the CSS styles provided by the component library, which are typically provided for use with Sass	16-24

Preparing for This Chapter

In this chapter, I continue using the exampleApp project that was first created in Chapter 20 and has been the focus of every chapter since. To prepare for this chapter, Listing 28-1 removes the animations added in Chapter 27 and simplifies the table component to remove features that are no longer required.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 28-1. Simplifying the Component in the table.component.ts File in the src/app/core Folder

```
import { Component, ElementRef } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { ActivatedRoute } from "@angular/router";
// import { HighlightTrigger } from "./table.animations";
// import { setPropertiesFromClasses, stateClassMap } from "./animationUtils";

@Component({
  selector: "paTable",
  templateUrl: "table.component.html",
  // animations: [HighlightTrigger]
})
export class TableComponent {
  category: string | null = null;

  constructor(public model: Model, activeRoute: ActivatedRoute) {
    activeRoute.params.subscribe(params => {
      this.category = params["category"] || null;
    })
  }
}
```

```

getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
}

getProducts(): Product[] {
    return this.model.getProducts()
        // .filter(p => this.category == null || p.category == this.category);
}

// get categories(): (string) [] {
//     return (this.model.getProducts()
//         .map(p => p.category)
//         .filter((c, index, array) => c != undefined
//             && array.indexOf(c) == index)) as string[];
// }

deleteProduct(key?: number) {
    if (key != undefined) {
        this.model.deleteProduct(key);
    }
}

// highlightCategory: string = "";

// getRowState(category: string | undefined, elem: HTMLTableRowElement): string {

//     let state = this.highlightCategory == "" ? "" :
//         this.highlightCategory == category ? "selected" : "notselected"
//     if (state != "") {
//         setPropertiesFromClasses(state, elem);
//     }
//     return state
// }
}

```

Listing 28-2 makes the corresponding changes to the template.

Listing 28-2. Simplifying the Template in the table.component.html File in the src/app/core Folder

```

<!-- <div class="form-group bg-info text-white p-2">
    <label>Category</label>
    <select [(ngModel)]="highlightCategory" class="form-control">
        <option value="">None</option>
        <option *ngFor="let category of categories">
            {{category}}
        </option>
    </select>
</div> -->

<table class="table table-sm table-bordered table-striped">
    <thead>
        <tr>

```

```

<th>ID</th><th>Name</th><th>Category</th><th>Price</th>
<th>Details</th><th></th>
```

ID	Name	Category	Price	Details	
1	Smartphone	Electronics	\$599.99	High-end smartphone with advanced features.	Delete Edit
2	Laptop	Electronics	\$1,299.99	Powerful laptop with a large screen and fast processor.	Delete Edit

[Create New Product](#)

Installing the Component Library

Open a new command prompt, navigate to the exampleApp folder, and run the following command to download and install the Angular Material package:

```
ng add @angular/material@13.0.2
```

The Angular Material package uses the schematics API to configure the project. The installation process presents several queries, the first of which is to confirm the package installation:

```
Using package manager: npm
Package information loaded.
The package @angular/material@13.0.2 will be installed and executed.
Would you like to proceed? (Y/n)
```

Press Y to confirm the installation. Select the default option for the remaining questions to complete the installation.

CHOOSING A COMPONENT LIBRARY

I have used Angular Material because it is the most popular Angular component library. There are several other packages available. Teradata Covalent (<https://teradata.github.io/covalent>) is an open-source library that follows the same Material Design standard as Angular Material, but with the addition of good charting components. Some packages present the features of the Bootstrap CSS package using Angular features, such as ng-bootstrap (<https://ng-bootstrap.github.io>) and ngx-bootstrap (<https://valor-software.com/ngx-bootstrap>), and each provides a different approach to developing components. There are also commercial packages, such as Kendo UI (<https://www.telerik.com/kendo-angular-ui>), which can be useful for development teams that require support.

If you don't know where to start, then try Angular Material. The documentation (<https://material.angular.io>) is good, and the package contains the components required by most projects.

Adjusting the HTML File

Installing the Angular Material package requires a change to the `index.html` file to resolve a conflict with the Bootstrap CSS styles that causes a scrollbar to be displayed even when the content fits within the browser window, caused by styles added to the `styles.css` file. Listing 28-3 changes the class to which the `body` element is assigned to resolve the issue.

Listing 28-3. Changing an Element Class in the `index.html` File in the `src` Folder

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>ExampleApp</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link rel="preconnect" href="https://fonts.gstatic.com">
  <link href=
    "https://fonts.googleapis.com/css2?family=Roboto:wght@300;400;500&display=swap"
    rel="stylesheet">
  <link href="https://fonts.googleapis.com/icon?family=Material+Icons"
    rel="stylesheet">
</head>
```

```
<body class="p-1">
  <app-root></app-root>
</body>
</html>
```

Running the Project

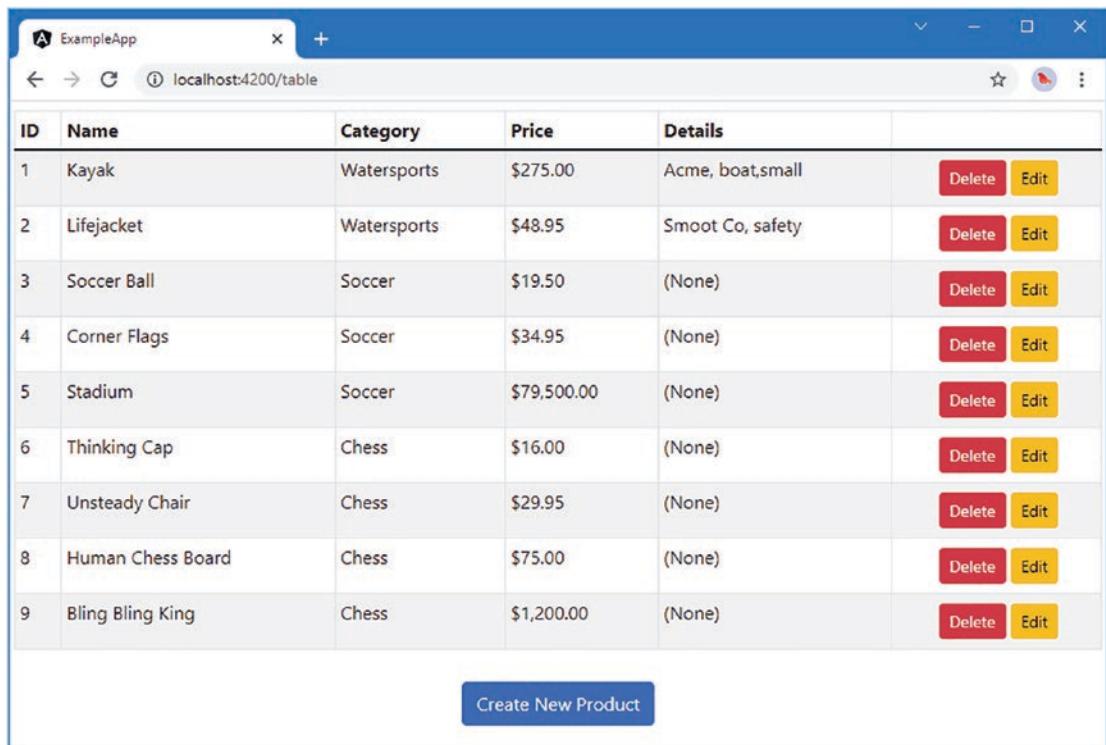
Open a new command prompt, navigate to the exampleApp folder, and run the following command to start the server that provides the RESTful web server:

```
npm run json
```

Open a separate command prompt, navigate to the exampleApp folder, and run the following command to start the Angular development tools:

```
ng serve
```

Open a new browser window and navigate to <http://localhost:4200/table> to see the content shown in Figure 28-1.



The screenshot shows a web browser window titled "ExampleApp" displaying a table of products. The table has columns for ID, Name, Category, Price, and Details. Each row contains a product with its details and two buttons: "Delete" (red) and "Edit" (yellow). A "Create New Product" button is located at the bottom of the table.

ID	Name	Category	Price	Details	
1	Kayak	Watersports	\$275.00	Acme, boat, small	<button>Delete</button> <button>Edit</button>
2	Lifejacket	Watersports	\$48.95	Smooth Co, safety	<button>Delete</button> <button>Edit</button>
3	Soccer Ball	Soccer	\$19.50	(None)	<button>Delete</button> <button>Edit</button>
4	Corner Flags	Soccer	\$34.95	(None)	<button>Delete</button> <button>Edit</button>
5	Stadium	Soccer	\$79,500.00	(None)	<button>Delete</button> <button>Edit</button>
6	Thinking Cap	Chess	\$16.00	(None)	<button>Delete</button> <button>Edit</button>
7	Unsteady Chair	Chess	\$29.95	(None)	<button>Delete</button> <button>Edit</button>
8	Human Chess Board	Chess	\$75.00	(None)	<button>Delete</button> <button>Edit</button>
9	Bling Bling King	Chess	\$1,200.00	(None)	<button>Delete</button> <button>Edit</button>

[Create New Product](#)

Figure 28-1. Running the example application

Using the Library Components

The simplest approach to using a component library is, as you might expect, to use the components it provides. In this section, I demonstrate how to integrate two features from the Angular Material library into the example project.

Using the Angular Button Directive

The Angular Material support for buttons is provided as a directive applied to button or anchor elements, as shown in Listing 28-4.

Listing 28-4. Using the Angular Material Button in the table.component.html File in the src/app/core Folder

```
<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr>
      <th>ID</th><th>Name</th><th>Category</th><th>Price</th>
      <th>Details</th><th></th></th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of getProducts()">
      <td>{{item.id}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price | currency:"USD" }}</td>
      <td>
        <ng-container *ngIf="item.details else empty">
          {{ item.details?.supplier }}, {{ item.details?.keywords}}
        </ng-container>
        <ng-template #empty>(None)</ng-template>
      </td>
      <td class="text-center">
        <button mat-flat-button color="accent" (click)="deleteProduct(item.id)">
          Delete
        </button>
        <button mat-flat-button color="warn"
          [routerLink]="/form", 'edit', item.id]>
          Edit
        </button>
      </td>
    </tr>
  </tbody>
</table>

<div class="p-2 text-center">
  <button mat-flat-button color="primary" routerLink="/form/create">
    Create New Product
  </button>
</div>
```

Angular Material provides several different styles of button, which are applied using the attributes described in Table 28-3.

Table 28-3. The Angular Material Button Attributes

Name	Description
mat-button	This attribute creates a simple borderless button, whose text is styled using an Angular Material theme color.
mat-stroked-button	This attribute adds a rectangular border to the mat-button style.
mat-raised-button	This attribute creates a button that appears to be raised from the page, displayed with a small amount of shadow. The button background is styled using an Angular Material theme color.
mat-flat-button	This attribute creates a button without the raised shadow and whose background is styled using an Angular Material theme color.
mat-icon-button	This attribute creates a button with a transparent background, intended to display an icon, which is styled using an Angular Material theme color.
mat-fab	This attribute creates a circular button with a shadow, whose background is styled using an Angular Material theme color.
mat-mini-fab	This button creates a small circular button with a shadow and a background styled using an Angular Material theme color.

Angular Material uses a color theme that is selected when the package is installed and which defines three color names, as described in Table 28-4.

Table 28-4. The Angular Material Color Names

Name	Description
primary	This name refers to the color used most often throughout the application.
accent	This name refers to the color used to highlight key parts of the user interface.
warn	This name refers to the color used for warnings and errors or to denote operations that require caution.

In Listing 28-4, I applied the mat-flat-button attribute, which will create a button whose appearance most closely matches the buttons I created using the Bootstrap styles. The theme color is specified using the color attribute, like this:

```
...
<button mat-flat-button color="accent" (click)="deleteProduct(item.id)">
...

```

The Angular Material button is applied to a regular HTML button element, which means that the click event is used to respond to user interaction.

Adding the Margin Style

Angular Material doesn't include utility styles for adding margins or padding to elements. Listing 28-5 defines a new global style that adds space around flat buttons.

Caution You may be tempted to mix and match styles from different packages, such as applying the Bootstrap m-1 style to button elements to which the mat-flat-button attribute has been added. Care must be taken because package styles are rarely written with this kind of combination in mind and there can be odd interactions.

Listing 28-5. Adding Styles in the styles.css File in the src Folder

```
html, body { height: 100%; }
body { margin: 0; font-family: Roboto, "Helvetica Neue", sans-serif; }

button.mat-flat-button { margin: 2px; }
```

Angular Material adds elements to classes that correspond to the attribute names described in Table 28-3, which means that I can use class selectors to locate button elements and apply a margin.

Importing the Component Module

Angular Material uses separate modules for each feature, which means that an application includes only the features it requires and doesn't add unused code to the download required by the client.

In a complex project, there can be a large number of dependencies on a component library, and they can be spread throughout the project's modules. To make it easier to manage the dependencies, it is a good idea to use a separate module. Add a file named `material.module.ts` in the `src/app` folder with the content shown in Listing 28-6.

Listing 28-6. The Contents of the material.module.ts File in the src/app Folder

```
import { NgModule } from "@angular/core";
import { MatButtonModule } from "@angular/material/button";

const features = [MatButtonModule];

@NgModule({
  imports: [features],
  exports: [features]
})
export class MaterialFeatures {}
```

The `MaterialFeatures` module imports and exports the `MatButtonModule` module from the Angular Material package. Listing 28-7 adds a dependency on the new module, which will be the only change to the core module to enable Angular Material features.

Listing 28-7. Importing a Module in the core.module.ts File in the src/app/core Folder

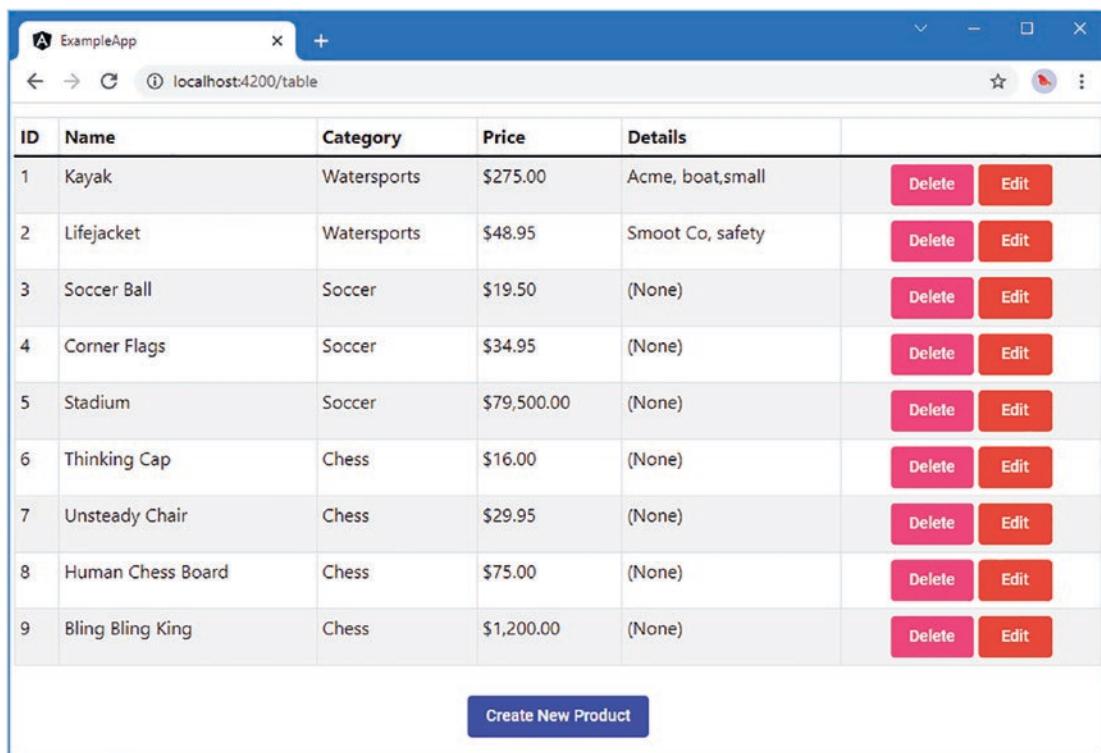
```

import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "./table.component";
import { FormComponent } from "./form.component";
//import { SharedState } from "./sharedState.service";
import { ValidationHelper } from "./validation_helper";
import { ValidationErrorsDirective } from "./validationErrors.directive";
import { HiLowValidatorDirective } from "../validation/hilow";
import { RouterModule } from "@angular/router";
import { ProductCountComponent } from "./productCount.component";
import { CategoryCountComponent } from "./categoryCount.component";
import { NotFoundComponent } from "./notFound.component";
import { UnsavedGuard } from "./unsaved.guard";
import { MaterialFeatures } from "../material.module"

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule, ReactiveFormsModule,
    RouterModule, MaterialFeatures],
  declarations: [TableComponent, FormComponent, ValidationHelper,
    ValidationErrorsDirective, HiLowValidatorDirective, ProductCountComponent,
    CategoryCountComponent, NotFoundComponent],
  exports: [ModelModule, TableComponent, FormComponent],
  providers: [UnsavedGuard]
})
export class CoreModule { }

```

Save the changes, and the Angular Material button will be displayed when the application is reloaded, as shown in Figure 28-2.



ID	Name	Category	Price	Details	
1	Kayak	Watersports	\$275.00	Acme, boat, small	<button>Delete</button> <button>Edit</button>
2	Lifejacket	Watersports	\$48.95	Smooth Co, safety	<button>Delete</button> <button>Edit</button>
3	Soccer Ball	Soccer	\$19.50	(None)	<button>Delete</button> <button>Edit</button>
4	Corner Flags	Soccer	\$34.95	(None)	<button>Delete</button> <button>Edit</button>
5	Stadium	Soccer	\$79,500.00	(None)	<button>Delete</button> <button>Edit</button>
6	Thinking Cap	Chess	\$16.00	(None)	<button>Delete</button> <button>Edit</button>
7	Unsteady Chair	Chess	\$29.95	(None)	<button>Delete</button> <button>Edit</button>
8	Human Chess Board	Chess	\$75.00	(None)	<button>Delete</button> <button>Edit</button>
9	Bling Bling King	Chess	\$1,200.00	(None)	<button>Delete</button> <button>Edit</button>

[Create New Product](#)

Figure 28-2. Using a component library

The Angular Material button feature is simple, but it shows the basic pattern to follow when using a component from a package: apply the component, add some tuning CSS styles, and import the feature module.

Note Styles provided by the Bootstrap CSS package are still being used in the example, with classes such as p-2 and text-center, which are used to center content and add padding. Most projects will use a single package, but Bootstrap and Angular Material will coexist.

Using the Angular Material Table

Buttons are relatively simple, and the main benefit of using the Angular Material button is consistency. Other components are more complex and provide more features, such as tables. Listing 28-8 removes the Bootstrap CSS styles from the table that displays product details and introduces the Angular Material table feature.

Listing 28-8. Changing the Table in the table.component.html File in the src/app/core Folder

```
<table mat-table [dataSource]="getProducts()">

  <mat-text-column name="id"></mat-text-column>
  <mat-text-column name="name"></mat-text-column>
  <mat-text-column name="category"></mat-text-column>

  <ng-container matColumnDef="price">
    <th mat-header-cell *matHeaderCellDef>Price</th>
    <td mat-cell *matCellDef="let item"> {{item.price | currency:"USD"}} </td>
  </ng-container>

  <ng-container matColumnDef="details">
    <th mat-header-cell *matHeaderCellDef>Details</th>
    <td mat-cell *matCellDef="let item">
      <ng-container *ngIf="item.details else empty">
        {{ item.details?.supplier }}, {{ item.details?.keywords }}</ng-container>
      <ng-template #empty>(None)</ng-template>
    </td>
  </ng-container>

  <ng-container matColumnDef="buttons">
    <th mat-header-cell *matHeaderCellDef></th>
    <td mat-cell *matCellDef="let item">
      <button mat-flat-button color="accent"
             (click)="deleteProduct(item.id)">
        Delete
      </button>
      <button mat-flat-button color="warn"
             [routerLink]="/form/edit, item.id">
        Edit
      </button>
    </ng-container>

    <tr mat-header-row *matHeaderRowDef="colsAndRows"></tr>
    <tr mat-row *matRowDef="let row; columns: colsAndRows"></tr>
  </table>

  <div class="p-2 text-center">
    <button mat-flat-button color="primary" routerLink="/form/create" >
      Create New Product
    </button>
  </div>
```

Angular Material tables are created by applying the `mat-table` attribute to a table element and creating a `dataSource` data binding that selects an array of values to display.

Angular Material focuses on defining columns to describe the contents of a table. A `mat-text-column` element is used for simple columns, where the column header is the name of the data property and the value is displayed without modification, like this:

```
...
<mat-text-column name="id"></mat-text-column>
...
```

The name attribute selects the property to be displayed and sets the name by which the column is identified. For more complex columns, the matColumnDef attribute is applied to an ng-container element that contains th and td elements that are included in the table head and body, respectively:

```
...
<ng-container matColumnDef="price">
  <th mat-header-cell *matHeaderCellDef>Price</th>
  <td mat-cell *matCellDef="let item"> {{item.price | currency:"USD"}} </td>
</ng-container>
...
```

The th element is given the mat-header-cell attribute, and the concise syntax is used to apply the matHeaderCellDef directive. The td element is given the mat-cell attribute, and the matCellDef directive is used to create an expression that selects the data used to create the contents of a table cell. There is an implicit value that provides the current data value, and, for the price column, this is formatted as a currency value using a pipe. This approach allows data values to be formatted or composed from multiple data source properties.

If you jump directly to using a component library without taking the time to understand how Angular works, the steps required to set up complex features can be impenetrable. But the knowledge you gained in earlier chapters helps reveal how the Angular Material table works, using features such as the concise directive syntax and implicit values to map the data in the data source to the content in the column descriptions.

The next step is to define the templates for the header and body rows, like this:

```
...
<tr mat-header-row *matHeaderRowDef="colsAndRows"></tr>
<tr mat-row *matRowDef="let row; columns: colsAndRows"></tr>
...
```

Columns are not shown unless they are configured with a row template. Columns are selected using an array containing the names assigned to the column's template, and Listing 28-9 adds a property to the component class to select all of the columns defined in Listing 28-8.

Listing 28-9. Selecting Columns in the table.component.ts File in the src/app/core Folder

```
import { Component, ElementRef } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { ActivatedRoute } from "@angular/router";

@Component({
  selector: "paTable",
  templateUrl: "table.component.html",
})
export class TableComponent {
  category: string | null = null;
```

```

constructor(public model: Model, activeRoute: ActivatedRoute) {
    activeRoute.params.subscribe(params => {
        this.category = params["category"] || null;
    })
}

getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
}

getProducts(): Product[] {
    return this.model.getProducts()
}

deleteProduct(key?: number) {
    if (key != undefined) {
        this.model.deleteProduct(key);
    }
}

colsAndRows: string[] = ['id', 'name', 'category', 'price', 'details', 'buttons'];
}

```

The next step is to define CSS styles that will supplement those used by Angular Material and style the table, as shown in Listing 28-10.

Listing 28-10. Defining Styles in the styles.css File in the src Folder

```

html, body { height: 100%; }
body { margin: 0; font-family: Roboto, "Helvetica Neue", sans-serif; }

button.mat-flat-button { margin: 2px; }

table.mat-table { width: 100%; }
th.mat-header-cell { font-size: large; font-weight: bold; }
td.mat-column-price { font-style: italic; }

```

The table element is added to the mat-table class, which allows me to set the width of the table. As the Angular Material generates the content for the table, it adds the elements it creates to classes that make it easy to adjust the appearance. The mat-cell and mat-header-cell classes are used to denote cells in the header and body. Elements are also added to classes that indicate which column a cell belongs to so that cells for the price column, for example, are added to the mat-column-price class. I use these classes to change the font settings for all th elements in the header and to italicize the values in the price column.

To finish up applying the Angular Material table, Listing 28-11 imports the module that contains the table features.

Listing 28-11. Importing the Component Module in the material.module.ts File in the src/app Folder

```

import { NgModule } from "@angular/core";
import { MatButtonModule } from "@angular/material/button";
import { MatTableModule } from "@angular/material/table";

const features = [MatButtonModule, MatTableModule];

```

```
@NgModule({
  imports: [features],
  exports: [features]
})
export class MaterialFeatures {}
```

When the application reloads, the Angular Material features are used to generate the table content, as shown in Figure 28-3.

ID	Name	Category	Price	Details		
1	Kayak	Watersports	\$275.00	Acme, boat, small	<button>Delete</button>	<button>Edit</button>
2	Lifejacket	Watersports	\$48.95	Smooth Co, safety	<button>Delete</button>	<button>Edit</button>
3	Soccer Ball	Soccer	\$19.50	(None)	<button>Delete</button>	<button>Edit</button>
4	Corner Flags	Soccer	\$34.95	(None)	<button>Delete</button>	<button>Edit</button>
5	Stadium	Soccer	\$79,500.00	(None)	<button>Delete</button>	<button>Edit</button>
6	Thinking Cap	Chess	\$16.00	(None)	<button>Delete</button>	<button>Edit</button>
7	Unsteady Chair	Chess	\$29.95	(None)	<button>Delete</button>	<button>Edit</button>
8	Human Chess Board	Chess	\$75.00	(None)	<button>Delete</button>	<button>Edit</button>
9	Bling Bling King	Chess	\$1,200.00	(None)	<button>Delete</button>	<button>Edit</button>

[Create New Product](#)

Figure 28-3. Using the Angular Material table

Using the Built-in Table Features

The basic table features don't offer much beyond the code with which I started the chapter. But one of the reasons for using a component library is to take advantage of features that are provided by the library authors, which you would otherwise have to write yourself.

The Angular Material table has some useful capabilities, including integrated support for paginating and sorting data. In Listing 28-12, I have declared dependencies on the Angular Material modules that provide these features.

Listing 28-12. Adding Dependencies in the material.module.ts File in the src/app Folder

```

import { NgModule } from "@angular/core";
import { MatButtonModule } from "@angular/material/button";
import { MatTableModule } from "@angular/material/table";
import { MatPaginatorModule } from "@angular/material/paginator"
import { MatSortModule } from "@angular/material/sort"

const features = [MatButtonModule, MatTableModule, MatPaginatorModule, MatSortModule];

@NgModule({
  imports: [features],
  exports: [features]
})
export class MaterialFeatures {}

```

Next, I need to expand the use of the observable in the repository, as shown in Listing 28-13. The Angular Material table will work with an array of data values, as the previous section showed, but its other features require a different approach, which is most easily achieved using the observable added to the repository.

Listing 28-13. Updating the Repository in the repository.model.ts File in the src/app/model Folder

```

import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { StaticDataSource } from "./static.datasource";
import { Observable, ReplaySubject } from "rxjs";
import { RestDataSource } from "./rest.datasource";

@Injectable()
export class Model {
  private products: Product[];
  private locator = (p: Product, id?: number) => p.id == id;
  private replaySubject: ReplaySubject<Product[]>;

  constructor(private dataSource: RestDataSource) {
    this.products = new Array<Product>();
    this.replaySubject = new ReplaySubject<Product[]>(1);
    this.dataSource.getData().subscribe(data => {
      this.products = data
      this.replaySubject.next(data);
      //this.replaySubject.complete();
    });
  }

  getProducts(): Product[] {
    return this.products;
  }

  getProduct(id: number): Product | undefined {
    return this.products.find(p => this.locator(p, id));
  }
}

```

```

getProductObservable(id: number): Observable<Product | undefined> {
    let subject = new ReplaySubject<Product | undefined>(1);
    this.replaySubject.subscribe(products => {
        subject.next(products.find(p => this.locator(p, id)));
        subject.complete();
    });
    return subject;
}

getProductsObservable(): Observable<Product[]> {
    return this.replaySubject;
}

getNextProductId(id?: number): Observable<number> {
    let subject = new ReplaySubject<number>(1);
    this.replaySubject.subscribe(products => {
        let nextId = 0;
        let index = products.findIndex(p => this.locator(p, id));
        if (index > -1) {
            nextId = products[products.length > index + 1
                ? index + 1 : 0].id ?? 0;
        } else {
            nextId = id || 0;
        }
        subject.next(nextId);
        subject.complete();
    });
    return subject;
}

getPreviousProductId(id?: number): Observable<number> {
    let subject = new ReplaySubject<number>(1);
    this.replaySubject.subscribe(products => {
        let nextId = 0;
        let index = products.findIndex(p => this.locator(p, id));
        if (index > -1) {
            nextId = products[index > 0
                ? index - 1 : products.length - 1].id ?? 0;
        } else {
            nextId = id || 0;
        }
        subject.next(nextId);
        subject.complete();
    });
    return subject;
}

saveProduct(product: Product) {
    if (product.id == 0 || product.id == null) {
        this.dataSource.saveProduct(product)
            .subscribe(p => this.products.push(p));
    }
}

```

```

    } else {
      this.dataSource.updateProduct(product).subscribe(p => {
        let index = this.products
          .findIndex(item => this.locator(item, p.id));
        this.products.splice(index, 1, p);
      });
    }
this.replaySubject.next(this.products);
}

deleteProduct(id: number) {
  this.dataSource.deleteProduct(id).subscribe(() => {
    let index = this.products.findIndex(p => this.locator(p, id));
    if (index > -1) {
      this.products.splice(index, 1);
      this.replaySubject.next(this.products);
    }
  });
}
}

```

These changes ensure that a new array of Product objects is sent via the observable when there is a change. Listing 28-14 changes the template that displays the table to add support for sorting data by the price column and for paginating data.

Listing 28-14. Enhancing the Table in the table.component.html File in the src/app/core Folder

```





```

```

        (click)="deleteProduct(item.id)">
    Delete
</button>
<button mat-flat-button color="warn"
        [routerLink]=["/form", 'edit', item.id]">
    Edit
</button>
</ng-container>

<tr mat-header-row *matHeaderRowDef="colsAndRows"></tr>
<tr mat-row *matRowDef="let row; columns: colsAndRows"></tr>
</table>

<mat-paginator [pageSize]="5" [pageSizeOptions]=[3, 5, 10]">
</mat-paginator>

<div class="p-2 text-center">
    <button mat-flat-button color="primary" routerLink="/form/create" >
        Create New Product
    </button>
</div>

```

The `matSort` attribute is applied to the table element, and the `mat-sort-header` attribute is added to headers that will allow the user to sort data. The `mat-paginator` component displays pagination controls for the table data.

The final step is to create a data source that supports sorting and pagination and that is populated with data through the observable exposed by the repository, as shown in Listing 28-15.

Listing 28-15. Creating a Data Source in the table.component.ts File in the src/app/core Folder

```

import { Component, ElementRef, ViewChild } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { ActivatedRoute } from "@angular/router";
import { MatTableDataSource } from "@angular/material/table";
import { MatPaginator } from "@angular/material/paginator";
import { MatSort } from "@angular/material/sort";

@Component({
    selector: "paTable",
    templateUrl: "table.component.html",
})
export class TableComponent {
    category: string | null = null;
    dataSource: MatTableDataSource<Product>;

    constructor(public model: Model, activeRoute: ActivatedRoute) {
        activeRoute.params.subscribe(params => {
            this.category = params["category"] || null;
        })
        this.dataSource = new MatTableDataSource<Product>();
        this.model.getProductsObservable().subscribe(newData => {

```

```

        this.dataSource.data = newData;
    })
}

getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
}

getProducts(): MatTableDataSource<Product> {
    return this.dataSource;
}

deleteProduct(key?: number) {
    if (key != undefined) {
        this.model.deleteProduct(key);
    }
}

colsAndRows: string[] = ['id', 'name', 'category', 'price', 'details', 'buttons'];

@ViewChild(MatPaginator) paginator!: MatPaginator;
@ViewChild(MatSort) sort!: MatSort;

ngAfterViewInit() {
    this.dataSource.paginator = this.paginator;
    this.dataSource.sort = this.sort;
}
}
}

```

The MatTableDataSource<Product> object represents a data source for Product objects, and its data property is used to update the data the table displays. The paginator and sort properties are used to associate the MatPaginator component and MatSort directive with the data source, which I do in the ngAfterViewInit method, to ensure that the child content is queried and assigned to the ViewChild properties. The result is that the data in the table is paginated and can be sorted by clicking the header for the Price column, as shown in Figure 28-4.

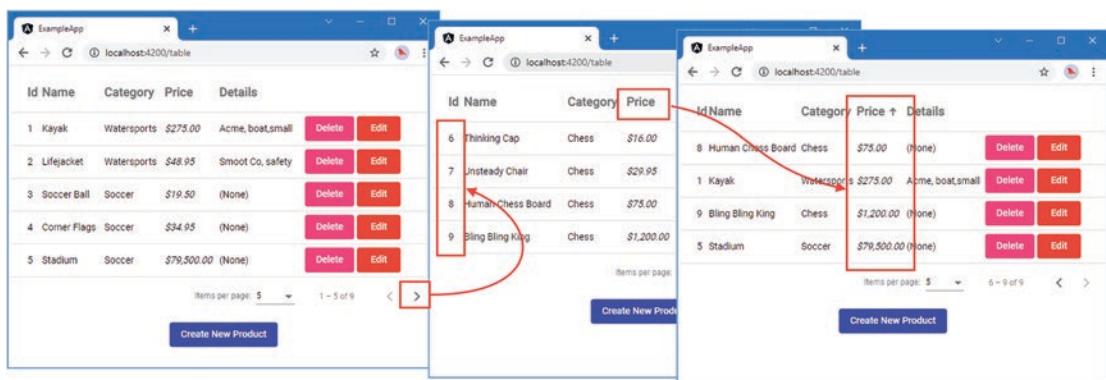


Figure 28-4. Using built-in table features

Writing your own pagination and sorting code isn't difficult—I demonstrated a simple paginator in the SportsStore application, for example—but using a component library means that you can rely on out-of-the-box features that are already tested. The trade-off is that you generally have to fit into a predefined model of how data will be expressed to get the most benefit, such as using the `MatTableDataSource<T>` class with Angular Material.

Matching the Component Library Theme

You can get a long way using just the features provided by a good component library, but some projects require more specialized features, which leads to custom Angular directives or components.

Most component libraries provide access to the underlying themes they use to style content, often expressed using *Sass*, which is a superset of CSS that makes it easier to create complex sets of styles without endless duplication of CSS properties. Sass files have the `.scss` extension and are compiled when the project is built to generate standard CSS stylesheets that the browser can understand. (Confusingly, Sass also supports a closely related syntax in files with the `.sass` file extension. The history of CSS and attempts to improve it are long and complex and can be ignored.)

I am not going to describe Sass in detail in this book—see <https://sass-lang.com> for full details—but I will explain the features that are required to style custom components with the Angular Material theme.

Creating the Custom Component

I am going to create a custom button component, which will let me show the use of themes without getting bogged down in the component itself. Add a file named `customButton.component.ts` to the `src/app/core` folder, with the content shown in Listing 28-16.

Listing 28-16. The Contents of the `customButton.component.ts` File in the `src/app/core` Folder

```
import { Component, ElementRef, Input, ViewChild } from "@angular/core";

@Component({
  selector: "customButton",
  templateUrl: "customButton.component.html"
})
export class CustomButton {

  @Input("themeColor")
  themeColor: string = "primary"

  @ViewChild("buttonTarget")
  button?: ElementRef

  ngAfterViewInit() {
    this.button?.nativeElement.classList.add(`custom-button-${this.themeColor}`);
  }
}
```

The component queries its template to locate a `button` element and assigns it to a class based on the value received through an input property.

To define the template for the component, add a file named `customButton.component.html` to the `src/app/core` folder with the content shown in Listing 28-17.

Listing 28-17. The Contents of the customButton.component.html File in the src/app/core Folder

```
<button #buttonTarget>
  <ng-content></ng-content>
</button>
```

Listing 28-18 adds the custom button component to the core module.

Listing 28-18. Adding the Component to the Module in the core.module.ts File in the src/app/core Folder

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "./table.component";
import { FormComponent } from "./form.component";
//import { SharedState } from "./sharedState.service";
import { ValidationHelper } from "./validation_helper";
import { ValidationErrorsDirective } from "./validationErrors.directive";
import { HiLowValidatorDirective } from "../validation/hilow";
import { RouterModule } from "@angular/router";
import { ProductCountComponent } from "./productCount.component";
import { CategoryCountComponent } from "./categoryCount.component";
import { NotFoundComponent } from "./notFound.component";
import { UnsavedGuard } from "./unsaved.guard";
import { MaterialFeatures } from "../material.module"
import { CustomButton } from "./customButton.component";

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule, ReactiveFormsModule,
    RouterModule, MaterialFeatures],
  declarations: [TableComponent, FormComponent, ValidationHelper,
    ValidationErrorsDirective, HiLowValidatorDirective, ProductCountComponent,
    CategoryCountComponent, NotFoundComponent, CustomButton],
  exports: [ModelModule, TableComponent, FormComponent],
  providers: [UnsavedGuard]
})
export class CoreModule { }
```

The final preparatory step is to use the new component, as shown in Listing 28-19.

Listing 28-19. Applying the Custom Component in the table.component.html File in the src/app/core Folder

```
<table mat-table [dataSource]="getProducts()" matSort>
  <!-- ...elements omitted for brevity... -->
</table>

<mat-paginator [pageSize]="5" [pageSizeOptions]="[3, 5, 10]">
</mat-paginator>
```

```
<div class="p-2 text-center">
  <button mat-flat-button color="primary" routerLink="/form/create" >
    Create New Product
  </button>

  <customButton themeColor="primary" routerLink="/form/create">
    Create New Product
  </customButton>

</div>
```

Save the changes, and you will see the new (unstyled) button shown alongside the standard Angular Material button, as shown in Figure 28-5.

Id	Name	Category	Price	Details	Delete	Edit
2	Lifejacket	Watersports	\$48.95	Smoot Co, safety	Delete	Edit
3	Soccer Ball	Soccer	\$19.50	(None)	Delete	Edit
4	Corner Flags	Soccer	\$34.95	(None)	Delete	Edit
5	Stadium	Soccer	\$79,500.00	(None)	Delete	Edit
6	Thinking Cap	Chess	\$16.00	(None)	Delete	Edit

Items per page: 5

1 – 5 of 8

< >

Create New Product

Create New Product

Figure 28-5. Applying a custom component

Using the Angular Material Theme

The Angular build tools have integrated support for working with SCSS files, which is the file format used by SASS. Add a file named `customButton.component.scss` to the `src/app/core` folder with the content shown in Listing 28-20.

FIGURING OUT THEME DETAILS

An investment of time is required to figure out how to apply the Angular Material themes to custom components, so do not rush into this process expecting it to be quick and easy.

To figure out how to create the styles I needed for the button component, I relied on the Angular Material theme documentation (<https://material.angular.io/guide/theming-your-components>) and the Material Design theme description (<https://material.io/design/material-theming/overview.html>), both of which contain useful guidance. But I spent most of the time reading through the SCSS files in the Angular Material package (<https://github.com/angular/components>) to figure out the purpose of different functions and to understand how the styles for the built-in components are generated. It was also helpful to use the browser F12 developer tools to see how HTML elements are styled.

But don't be put off. Once you have worked your way through the process for one component, you will have learned enough to make subsequent components much simpler.

Listing 28-20. The Contents of the customButton.component.scss File in the src/app/core Folder

```
@use "@angular/material" as material;

$primary: material.define-palette(material.$indigo-palette);
$accent: material.define-palette(material.$pink-palette, A200, A100, A400);
$warn: material.define-palette(material.$red-palette);

$typography: material.define-typography-config();

button[class*="custom-button-"] {
  padding: 7px 12px;
  border: none;
  border-radius: 4px;
  margin: 2px;
}

button.custom-button {
  @each $name, $palette in (primary: $primary, accent: $accent, warn: $warn) {
    &-$name {
      background-color: material.get-color-from-palette($palette, default);
      color: material.get-color-from-palette($palette, default-contrast);
      font: {
        family: material.font-family($typography, button);
        size: material.font-size($typography, button);
        weight: material.font-weight($typography, button);
      }
    }
  }
}

$bg: material.$light-theme-background-palette;
$fg: material.$light-theme-foreground-palette;
```

```
:host[disabled] button[class*="custom-button-"],
  button[class*="custom-button-"]:disabled {
  background-color: material.get-color-from-palette($bg, disabled-button);
  color: material.get-color-from-palette($fg, disabled-button);
}
```

Sass has a concise syntax, which can make it difficult to understand what is happening in the listing until you have at least a little experience. The first statement is an @use expression:

```
...
@use "@angular/material" as material;
...
```

Sass supports functions and variables, which can be used to generate CSS styles, and the @use expression provides access to the Sass features that Angular Material provides. The next group of statements create the primary, accent, and warn palettes from the Angular Material theme:

```
...
$primary: material.define-palette(material.$indigo-palette);
$accent: material.define-palette(material.$pink-palette, A200, A100, A400);
$warn: material.define-palette(material.$red-palette);
...
```

Angular Material defines a set of base palettes, which contain a range of hues for a single color. The term material.\$indigo-palette, for example, refers to the set of indigo hues. (The material prefix was specified in the @use expression and allows me to access Angular Material Sass features, and the \$ sign indicates a variable so that material.\$indigo-palette refers to a variable named indigo-palette defined by the Angular Material package.) The define-palette function is used to select specific hues and give them convenient names, such as default and text, which help ensure consistency when applying styles. The indigo and pink palettes correspond to the default theme, which was chosen when the Angular Material package was installed. If you select a different theme for a project, then you will need to use the palettes that correspond to the colors of that theme.

The next step is to get the font configuration that Angular Material applies to its components:

```
...
$typography: material.define-typography-config();
...
```

The define-typography-config function returns a map where the keys are the names of styles that can be applied to text. A complete list of these styles can be found at <https://material.angular.io/guide/typography>, but the style name I want for this example is button, which provides the font settings for buttons.

Not all of the styles applied to buttons are specific to a palette, and I have used a selector that will match all of the palette-specific classes to apply these styles:

```
...
button[class*="custom-button-"] {
  padding: 7px 12px;
  border: none;
  border-radius: 4px;
  margin: 2px;
}
...
```

The next expression is the most complex and is responsible for generating the styles that are specific to a palette:

```
...
button.custom-button {
  @each $name, $palette in (primary: $primary, accent: $accent, warn: $warn) {
    &-$name {
      background-color: material.get-color-from-palette($palette, default);
      color: material.get-color-from-palette($palette, default-contrast);
      font: {
        family: material.font-family($typography, button);
        size: material.font-size($typography, button);
        weight: material.font-weight($typography, button);
      }
    }
  }
}
...

```

The effect of this expression is to generate a style for each of the primary, accent, and warn palettes, which contains background-color, color, and font properties that are specific to each palette. The get-color-from-palette function is used to get a color from a palette, either by hue or by using one of the names created by the define-palette function. The name default refers to the default color, and the default-contrast name refers to a color that can be used for text:

```
...
background-color: material.get-color-from-palette($palette, default);
color: material.get-color-from-palette($palette, default-contrast);
...

```

The values for the font properties are obtained using the font-family, font-size, and font-weight functions, which read values from the typography configuration settings.

Two more palettes are required to deal with disabled buttons:

```
...
$bg: material.$light-theme-background-palette;
$fg: material.$light-theme-foreground-palette;
...

```

The themes that Angular Material provides are categorized as either *light* or *dark*, and there are additional palettes of foreground and background colors that are shared by these light and dark themes, such as the colors for disabled buttons. The default indigo/pink theme is light, so I have assigned the light theme palettes to variables named fg and bg. These palettes are used to create a style that is applied to disabled buttons:

```
...
:host[disabled] button[class*="custom-button-"],
  button[class*="custom-button-"]:disabled {
  background-color: material.get-color-from-palette($bg, disabled-button);
  color: material.get-color-from-palette($fg, disabled-button);
}
...

```

The foreground and background palettes contain colors named disabled-button, which are used to set the background-color and color properties when a button is disabled. The selector matches button elements that are disabled or whose host element is disabled. The :host selector is required by the Angular view encapsulation feature and allows the component to be disabled by applying the disabled attribute to the customButton element.

SCSS files are applied to the component in just the same way as regular CSS files, as shown in Listing 28-21.

Listing 28-21. Applying Styles in the customButton.component.ts File in the src/app/core Folder

```
import { Component, ElementRef, Input, ViewChild } from "@angular/core";

@Component({
  selector: "customButton",
  templateUrl: "customButton.component.html",
  styleUrls: ["customButton.component.scss"]
})
export class CustomButton {

  @Input("themeColor")
  themeColor: string = "primary"

  @ViewChild("buttonTarget")
  button?: ElementRef

  ngAfterViewInit() {
    this.button?.nativeElement.classList.add(`custom-button-${this.themeColor}`);
  }
}
```

During the build process, the SCSS files are processed to generate CSS files that can be sent to the browser. Figure 28-6 shows the built-in Angular Material button alongside the styles custom component.

Id	Name	Category	Price	Details	Delete	Edit
2	Lifejacket	Watersports	\$48.95	Smooth Co, safety	Delete	Edit
3	Soccer Ball	Soccer	\$19.50	(None)	Delete	Edit
4	Corner Flags	Soccer	\$34.95	(None)	Delete	Edit
5	Stadium	Soccer	\$79,500.00	(None)	Delete	Edit
6	Thinking Cap	Chess	\$16.00	(None)	Delete	Edit

Items per page: 1 - 5 of 8 < >

[Create New Product](#) [Create New Product](#)

Figure 28-6. Applying a theme to a custom component

Applying the Ripple Effect

To finish off this chapter, I am going to add an animation effect to my custom button. Angular Material includes a ripple effect that is used to highlight user interaction, such as when a button is clicked. It is difficult to show this on a printed page, but Figure 28-7 gives an idea of how the color of a built-in Angular Material button is progressively changed when it is clicked.

**Figure 28-7.** The Angular Material button ripple effect

When the user clicks a button, a circle of lighter color spreads out from the pointer. The best way to see this effect is to hold the mouse button down because the animation will be terminated when the mouse is released.

Angular Material makes the ripple feature available as a directive that can be applied to any component. Listing 28-22 imports the ripple module from the Angular Material package.

Listing 28-22. Adding a Dependency in the material.module.ts File in the src/app Folder

```
import { NgModule } from "@angular/core";
import { MatButtonModule } from "@angular/material/button";
import { MatTableModule } from "@angular/material/table";
import { MatPaginatorModule } from "@angular/material/paginator"
import { MatSortModule } from "@angular/material/sort"
import { MatRippleModule } from "@angular/material/core";

const features = [MatButtonModule, MatTableModule, MatPaginatorModule, MatSortModule,
  MatRippleModule];

@NgModule({
  imports: [features],
  exports: [features]
})
export class MaterialFeatures {}
```

Listing 28-23 applies the ripple directive to the button element in the custom component's template.

Listing 28-23. Applying a Ripple in the customButton.component.html File in the src/core/app Folder

```
<button #buttonTarget matRipple>
  <ng-content></ng-content>
</button>
```

Ripples work by adding a div element inside the button, to which the animation is applied. The color used for the ripple is derived from the palette used for the button, as shown in Listing 28-24.

Listing 28-24. Defining a Style in the customButton.component.scss File in the src/app/core Folder

```
@use "@angular/material" as material;

$primary: material.define-palette(material.$indigo-palette);
$accent: material.define-palette(material.$pink-palette, A200, A100, A400);
$warn: material.define-palette(material.$red-palette);

$typography: material.define-typography-config();

button[class*="custom-button-"] {
  padding: 7px 12px;
  border: none;
  border-radius: 4px;
  margin: 2px;
}

button.custom-button {
  @each $name, $palette in (primary: $primary, accent: $accent, warn: $warn) {
    &-$name {
      background-color: material.get-color-from-palette($palette, default);
      color: material.get-color-from-palette($palette, default-contrast);
      font: {
```

```

        family: material.font-family($typography, button);
        size: material.font-size($typography, button);
        weight: material.font-weight($typography, button);
    }
}

&-${$name} ::ng-deep .mat-ripple-element {
    background-color: material.get-color-from-palette($palette,
        default-contrast, 0.1);
}
}

$bg: material.$light-theme-background-palette;
$fg: material.$light-theme-foreground-palette;

:host[disabled] button[class*="custom-button-"],
    button[class*="custom-button-"]:disabled {
    background-color: material.get-color-from-palette($bg, disabled-button);
    color: material.get-color-from-palette($fg, disabled-button);
}

```

The `::ng-deep` pseudoclass is used to prevent Angular from modifying the name of the `mat-ripple-element` class selector for view encapsulation. (The `/deep/` and `>>>` selectors are not supported by Sass.) The `get-color-from-palette` function is used to get a color from the chosen palette with an opacity value, which was chosen to match the one used by the built-in Angular Material button feature. The result is that the custom button displays a ripple when clicked, as shown in Figure 28-8.



Figure 28-8. Applying a ripple effect to a custom component

Summary

In this chapter, I demonstrated how a component library such as Angular Material can be introduced into a project to supplement or replace custom components. I also explained how the theme provided by Angular Material can be applied to custom components to ensure consistency across the application. In the next chapter, I explain how to perform unit testing in an Angular project.

CHAPTER 29



Angular Unit Testing

In this chapter, I describe the tools that Angular provides for unit testing components and directives. Some Angular building blocks, such as pipes and services, can be readily tested in isolation using the basic testing tools that I set up at the start of the chapter. Components (and, to a lesser extent, directives) have complex interactions with their host elements and with their template content and require special features. Table 29-1 puts Angular unit testing in context.

DECIDING WHETHER TO UNIT TEST

Unit testing is a contentious topic. This chapter assumes you do want to do unit testing and shows you how to set up the tools and apply them to Angular components and directives. It isn't an introduction to unit testing, and I make no effort to persuade skeptical readers that unit testing is worthwhile. If you would like an introduction to unit testing, then there is a good article here: https://en.wikipedia.org/wiki/Unit_testing.

I like unit testing, and I use it in my own projects—but not all of them and not as consistently as you might expect. I tend to focus on writing unit tests for features and functions that I know will be hard to write and that are likely to be the source of bugs in deployment. In these situations, unit testing helps structure my thoughts about how to best implement what I need. I find that just thinking about what I need to test helps produce ideas about potential problems, and that's before I start dealing with actual bugs and defects.

That said, unit testing is a tool and not a religion, and only you know how much testing you require. If you don't find unit testing useful or if you have a different methodology that suits you better, then don't feel you need to unit test just because it is fashionable. (However, if you don't have a better methodology and you are not testing at all, then you are probably letting users find your bugs, which is rarely ideal.)

Table 29-1. Putting Angular Unit Testing Context

Question	Answer
What is it?	Angular components and directives require special support for testing so that their interactions with other parts of the application infrastructure can be isolated and inspected.
Why is it useful?	Isolated unit tests can assess the basic logic provided by the class that implements a component or directive but do not capture the interactions with host elements, services, templates, and other important Angular features.
How is it used?	Angular provides a test bed that allows a realistic application environment to be created and then used to perform unit tests.
Are there any pitfalls or limitations?	Like much of Angular, the unit testing tools are complex. It can take some time and effort to get to the point where unit tests are easily written and run and you are sure that you have isolated the correct part of the application for testing.
Are there any alternatives?	As noted, you don't have to unit test your projects. But if you do want to unit testing, then you will need to use the Angular features described in this chapter.

Table 29-2 summarizes the chapter.

Table 29-2. Chapter Summary

Problem	Solution	Listing
Performing a basic test on a component	Initialize a test module and create an instance of the component. If the component has an external template, an additional compilation step must be performed.	1–9, 11–13
Testing a component's data bindings	Use the <code>DebugElement</code> class to query the component's template.	10
Testing a component's response to events	Trigger the events using the debug element.	14–16
Testing a component's output properties	Subscribe to the <code>EventEmitter</code> created by the component.	17, 18
Testing a component's input properties	Create a test component whose template applies the component under test.	19, 20
Testing a directive	Create a test component whose template applies the directive under test.	21, 22

Preparing the Example Project

I continue to use the exampleApp project from earlier chapters. I need a simple target to focus on for unit testing, so Listing 29-1 changes the routing configuration so that the `ondemand` feature module is loaded by default.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 29-1. Changing the Routing Configuration in the app.routing.ts File in the src/app Folder

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";
import { NotFoundComponent } from "./core/notFound.component";
import { UnsavedGuard } from "./core/unsaved.guard";

const routes: Routes = [
  {
    path: "ondemand",
    loadChildren: () => import("./ondemand/ondemand.module")
      .then(m => m.OndemandModule)
  },
  { path: "", redirectTo: "/ondemand", pathMatch: "full" }
]

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

This module contains some simple components that I will use to demonstrate different unit testing features. To keep the content shown by the application simple, Listing 29-2 tidies up the template displayed by the top-level component in the feature module.

Listing 29-2. Simplifying the ondemand.component.html File in the src/app/ondemand Folder

```
<div class="container-fluid">
  <div class="row">
    <div class="col-12 p-2">
      <router-outlet></router-outlet>
    </div>
  </div>
  <div class="row">
    <div class="col-6 p-2">
      <router-outlet name="left"></router-outlet>
    </div>
    <div class="col-6 p-2">
      <router-outlet name="right"></router-outlet>
    </div>
  </div>
</div>
<button class="btn btn-secondary m-2" routerLink="/ondemand">Normal</button>
<button class="btn btn-secondary m-2" routerLink="/ondemand/swap">Swap</button>
```

Open a new command prompt, navigate to the `exampleApp` folder, and run the following command to start the server that provides the RESTful web server:

```
npm run json
```

The RESTful web service isn't used directly in this chapter, but running it prevents errors. Open a separate command prompt, navigate to the `exampleApp` folder, and run the following command to start the Angular development tools:

```
ng serve
```

Open a new browser window and navigate to `http://localhost:4200` to see the content shown in Figure 29-1.

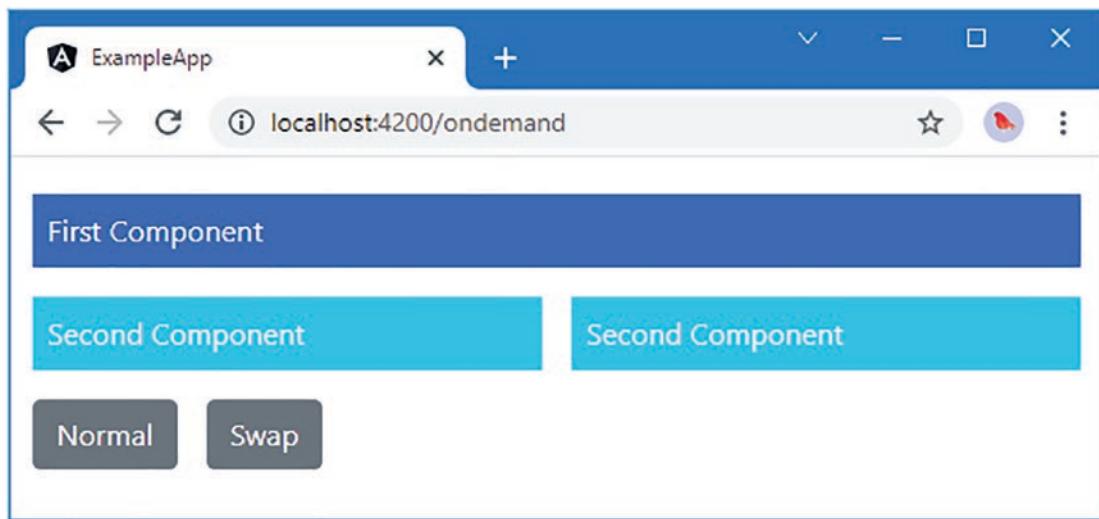


Figure 29-1. Running the example application

Running a Simple Unit Test

When a new project is created using the `ng new` command, all the packages and tools required for unit testing are installed, based on the Jasmine test framework. To create a simple unit test to confirm that everything is working, I created the `src/app/tests` folder and added to it a file named `app.component.spec.ts` with the contents shown in Listing 29-3. The naming convention for unit tests makes it obvious which file the tests apply to.

Listing 29-3. The Contents of the `app.component.spec.ts` File in the `src/app/tests` Folder

```
describe("Jasmine Test Environment", () => {
  it("is working", () => expect(true).toBe(true));
});
```

I explain the basics of working with the Jasmine API shortly, and you can ignore the syntax for the moment. Using a new command prompt, navigate to the exampleApp folder, and run the following command:

```
ng test
```

This command starts the Karma test runner, which opens a new browser tab with the content shown in Figure 29-2.

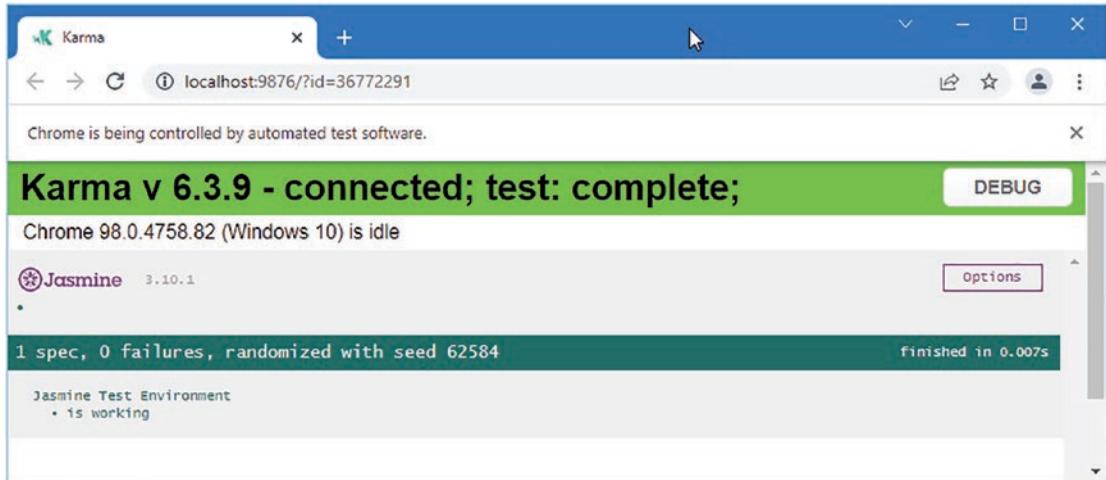


Figure 29-2. Starting the Karma test runner

The browser window is used to run the tests, but the important information is written out to the command prompt used to start the test tools, where you will see a message like this:

```
Chrome 98.0.4758.82 (Windows 10): Executed 1 of 1 SUCCESS (0.106 secs / 0.001 secs)
```

This shows that the single unit test in the project has been located and executed successfully. Whenever you make a change that updates one of the JavaScript files in the project, the unit tests will be located and executed, and any problems will be written to the command prompt. To show what an error looks like, Listing 29-4 changes the unit test so that it will fail.

Listing 29-4. Making a Unit Test Fail in the app.component.spec.ts File in the src/app/tests Folder

```
describe("Jasmine Test Environment", () => {
  it("is working", () => expect(true).toBe(false));
});
```

This test will fail and will result in the following output, which indicates the test that has failed and what went wrong:

```
Chrome 98.0.4758.82 (Windows 10) Jasmine Test Environment is working FAILED
Error: Expected true to be false.
  at <Jasmine>
  at UserContext.<anonymous> (src/app/tests/app.component.spec.ts:2:41)
  at ZoneDelegate.invoke (node_modules/zone.js/fesm2015/zone.js:372:1)
  at ProxyZoneSpec.onInvoke (node_modules/zone.js/fesm2015/zone-testing.js:287:1)
Chrome 98.0.4758.82 (Windows 10): Executed 1 of 1 (1 FAILED)
TOTAL: 1 FAILED, 0 SUCCESS
```

Working with Jasmine

The API that Jasmine provides chains together JavaScript methods to define unit tests. You can find the full documentation for Jasmine at <http://jasmine.github.io>, but Table 29-3 describes the most useful functions for Angular testing.

Table 29-3. Useful Jasmine Methods

Name	Description
describe(description, function)	This method is used to group a set of related tests.
beforeEach(function)	This method is used to specify a task that is performed before each unit test.
afterEach(function)	This method is used to specify a test that is performed after each unit test.
it(description, function)	This method is used to perform the test action.
expect(value)	This method is used to identify the result of the test.
toBe(value)	This method specifies the expected value of the test.

You can see how the methods in Table 29-3 were used to create the unit test in Listing 29-4.

```
...
describe("Jasmine Test Environment", () => {
  it("is working", () => expect(true).toBe(false));
});
...
```

You can also see why the test has failed since the expect and toBe methods have been used to check that true and false are equal. Since this cannot be the case, the test fails.

The toBe method isn't the only way to evaluate the result of a unit test. Table 29-4 shows other evaluation methods provided by Angular.

Table 29-4. Useful Jasmine Evaluation Methods

Name	Description
toBe(value)	This method asserts that a result is the same as the specified value (but need not be the same object).
toEqual(object)	This method asserts that a result is the same object as the specified value.
toMatch(regexp)	This method asserts that a result matches the specified regular expression.
toBeDefined()	This method asserts that the result has been defined.
toBeDefined()	This method asserts that the result has not been defined.
toBeNull()	This method asserts that the result is null.
toBeTruthy()	This method asserts that the result is truthy, as described in Chapter 3.
toBeFalsy()	This method asserts that the result is falsy, as described in Chapter 3.
toContain(substring)	This method asserts that the result contains the specified substring.
toBeLessThan(value)	This method asserts that the result is less than the specified value.
toBeGreaterThan(value)	This method asserts that the result is more than the specified value.

Listing 29-5 shows how these evaluation methods can be used in tests, replacing the failing test from the previous section.

Listing 29-5. Replacing the Unit Test in the app.component.spec.ts File in the src/app/tests Folder

```
describe("Jasmine Test Environment", () => {
  it("test numeric value", () => expect(12).toBeGreaterThan(10));
  it("test string value", () => expect("London").toMatch("^Lon"));
});
```

When you save the changes to the file, the tests will be executed, and the results will be shown in the command prompt.

Testing an Angular Component

The building blocks of an Angular application can't be tested in isolation because they depend on the underlying features provided by Angular and by the other parts of the project, including the services, directives, templates, and modules it contains. As a consequence, testing a building block such as a component means using testing utilities that are provided by Angular to re-create enough of the application to let the component function so that tests can be performed against it. In this section, I walk through the process of performing a unit test on the `FirstComponent` class in the `OnDemand` feature module. As a reminder, here is the definition of the component:

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

@Component({
  selector: "first",
  template: `<div class="bg-primary text-white p-2">First Component</div>`
})
export class FirstComponent { }
```

This component is so simple that it doesn't have functionality of its own to test, but it is enough to demonstrate how the test process is applied.

Working with the TestBed Class

At the heart of Angular unit testing is a class called `TestBed`, which is responsible for simulating the Angular application environment so that tests can be performed. Table 29-5 describes the most useful methods provided by the `TestBed` method, all of which are static.

Table 29-5. Useful `TestBed` Methods

Name	Description
<code>configureTestingModule</code>	This method is used to configure the Angular testing module.
<code>createComponent</code>	This method is used to create an instance of the component.
<code>compileComponents</code>	This method is used to compile components, as described in the “Testing a Component with an External Template” section.

The `configureTestingModule` method is used to configure the Angular module that is used in testing, using the same properties supported by the `@NgModule` decorator. Just like in a real application, a component cannot be used in a unit test unless it has been added to the `declarations` property of the module. This means that the first step in most unit tests is to configure the testing module. To demonstrate, I added a file named `first.component.spec.ts` to the `src/app/tests` folder with the content shown in Listing 29-6.

Listing 29-6. The Contents of the `first.component.spec.ts` File in the `src/app/tests` Folder

```
import { TestBed } from "@angular/core/testing";
import { FirstComponent } from "../ondemand/first.component";

describe("FirstComponent", () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [FirstComponent]
    });
  });
});
```

The `TestBed` class is defined in the `@angular/core/testing` module, and the `configureTestingModule` accepts an object whose `declarations` property tells the test module that the `FirstComponent` class is going to be used.

Tip Notice that the TestBed class is used within the beforeEach function. If you try to use the TestBed outside of this function, you will see an error about using Promises.

The next step is to create a new instance of the component so that it can be used in tests. This is done using the `createComponent` method, as shown in Listing 29-7.

Listing 29-7. Creating a Component in the first.component.spec.ts File in the src/app/tests Folder

```
import { TestBed, ComponentFixture } from "@angular/core/testing";
import { FirstComponent } from "../ondemand/first.component";

describe("FirstComponent", () => {

  let fixture: ComponentFixture<FirstComponent>;
  let component: FirstComponent;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [FirstComponent]
    });
    fixture = TestBed.createComponent(FirstComponent);
    component = fixture.componentInstance;
  });

  it("is defined", () => {
    expect(component).toBeDefined()
  });
});
```

The argument to the `createComponent` method tells the test bed which component type it should instantiate, which is `FirstComponent` in this case. The result is a `ComponentFixture<FirstComponent>` object, which provides features for testing a component, using the methods and properties described in Table 29-6.

Table 29-6. Useful ComponentFixture Methods and Properties

Name	Description
<code>componentInstance</code>	This property returns the component object.
<code>debugElement</code>	This property returns the test host element for the component.
<code>nativeElement</code>	This property returns the DOM object representing the host element for the component.
<code>detectChanges()</code>	This method causes the test bed to detect state changes and reflect them in the component's template.
<code>whenStable()</code>	This method returns a Promise that is resolved when the effect of an operation has been fully applied.

In the listing, I use the `componentInstance` property to get the `FirstComponent` object that has been created by the test bed and perform a simple test to ensure that it has been created by using the `expect` method to select the component object as the target of the test and the `toBeDefined` method to perform the test. I demonstrate the other methods and properties in the sections that follow.

Configuring the Test Bed for Dependencies

One of the most important features of Angular applications is dependency injection, which allows components and other building blocks to receive services by declaring dependencies on them using constructor parameters. Listing 29-8 adds a dependency on the data model repository service to the `FirstComponent` class.

Listing 29-8. Adding a Service Dependency in the `first.component.ts` File in the `src/app/ondemand` Folder

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";

@Component({
  selector: "first",
  template: `<div class="bg-primary p-a-1">
    There are
    <span class="strong"> {{getProducts().length}} </span>
    products
  </div>`
})
export class FirstComponent {

  constructor(private repository: Model) {}

  category: string = "Soccer";

  getProducts(): Product[] {
    return this.repository.getProducts()
      .filter(p => p.category == this.category);
  }
}
```

The component uses the repository to provide a filtered collection of `Product` objects, which are exposed through a method called `getProducts` and filtered using a `category` property. The inline template has a corresponding data binding that displays the number of products that the `getProducts` method returns.

Being able to unit test the component means providing it with a repository service. The Angular test bed will take care of resolving dependencies as long as they are configured through the test module. Effective unit testing generally requires components to be isolated from the rest of the application, which means that mock or fake objects (also known as *test doubles*) are used as substitutes for real services in unit tests. Listing 29-9 configures the test bed so that a fake repository is used to provide the component with its service.

Listing 29-9. Providing a Service in the first.component.spec.ts File in the src/app/tests Folder

```

import { TestBed, ComponentFixture } from "@angular/core/testing";
import { FirstComponent } from "../ondemand/first.component";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";

describe("FirstComponent", () => {

  let fixture: ComponentFixture<FirstComponent>;
  let component: FirstComponent;

  let mockRepository = {
    getProducts: function () {
      return [
        new Product(1, "test1", "Soccer", 100),
        new Product(2, "test2", "Chess", 100),
        new Product(3, "test3", "Soccer", 100),
      ]
    }
  }

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [FirstComponent],
      providers: [
        { provide: Model, useValue: mockRepository }
      ]
    });
    fixture = TestBed.createComponent(FirstComponent);
    component = fixture.componentInstance;
  });

  it("filters categories", () => {
    component.category = "Chess";
    expect(component.getProducts().length).toBe(1);
    component.category = "Soccer";
    expect(component.getProducts().length).toBe(2);
    component.category = "Running";
    expect(component.getProducts().length).toBe(0);
  });
});

```

The `mockRepository` variable is assigned an object that provides a `getProducts` method that returns fixed data that can be used to test for known outcomes. To provide the component with the service, the `providers` property for the object passed to the `TestBed.configureTestingModule` method is configured in the same way as a real Angular module, using the value provider to resolve dependencies on the `Model` class using the `mockRepository` variable. The test invokes the component's `getProducts` method and compares the results with the expected outcome, changing the value of the `category` property to check different filters.

Testing Data Bindings

The previous example showed how a component's properties and methods can be used in a unit test. This is a good start, but many components will also include small fragments of functionality in the data binding expressions contained in their templates, and these should be tested as well. Listing 29-10 checks that the data binding in the component's template correctly displays the number of products in the mock data model.

Listing 29-10. Unit Testing a Data Binding in the first.component.spec.ts File in the src/app/tests Folder

```
import { TestBed, ComponentFixture } from "@angular/core/testing";
import { FirstComponent } from "../ondemand/first.component";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { DebugElement } from "@angular/core";
import { By } from "@angular/platform-browser";

describe("FirstComponent", () => {

  let fixture: ComponentFixture<FirstComponent>;
  let component: FirstComponent;
  let debugElement: DebugElement;
  let bindingElement: HTMLSpanElement;

  let mockRepository = {
    getProducts: function () {
      return [
        new Product(1, "test1", "Soccer", 100),
        new Product(2, "test2", "Chess", 100),
        new Product(3, "test3", "Soccer", 100),
      ]
    }
  }

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [FirstComponent],
      providers: [
        { provide: Model, useValue: mockRepository }
      ]
    });
    fixture = TestBed.createComponent(FirstComponent);
    component = fixture.componentInstance;
    debugElement = fixture.debugElement;
    bindingElement = debugElement.query(By.css("span")).nativeElement;
  });

  it("filters categories", () => {
    component.category = "Chess"
    fixture.detectChanges();
    expect(component.getProducts().length).toBe(1);
    expect(bindingElement.textContent).toContain("1");
  });
})
```

```

component.category = "Soccer";
fixture.detectChanges();
expect(component.getProducts().length).toBe(2);
expect(bindingElement.textContent).toContain("2");

component.category = "Running";
fixture.detectChanges();
expect(component.getProducts().length).toBe(0);
expect(bindingElement.textContent).toContain("0");
});
});

```

The `ComponentFixture.debugElement` property returns a `DebugElement` object that represents the root element from the component's template, and Table 29-7 lists the most useful methods and properties described by the `DebugElement` class.

Tip If you don't see the test output, then restart the `ng test` command.

Table 29-7. Useful `DebugElement` Properties and Methods

Name	Description
<code>nativeElement</code>	This property returns the object that represents the HTML element in the DOM.
<code>children</code>	This property returns an array of <code>DebugElement</code> objects representing the children of this element.
<code>query(selectorFunction)</code>	The <code>selectorFunction</code> is passed a <code>DebugElement</code> object for each HTML element in the component's template, and this method returns the first <code>DebugElement</code> for which the function returns true.
<code>queryAll(selectorFunction)</code>	This is similar to the <code>query</code> method, except the result is all the <code>DebugElement</code> objects for which the function returns true.
<code>triggerEventHandler(name, event)</code>	This method triggers an event. See the “Testing Component Events” section for details.

Locating elements is done through the `query` and `queryAll` methods, which accept functions that inspect `DebugElement` objects and return true if they should be included in the results. The `By` class, defined in the `@angular/platform-browser` module, makes it easier to locate elements in the component's template through the static methods described in Table 29-8.

Table 29-8. The `By` Methods

Name	Description
<code>By.all()</code>	This method returns a function that matches any element.
<code>By.css(selector)</code>	This method returns a function that uses a CSS selector to match elements.
<code>By.directive(type)</code>	This method returns a function that matches elements to which the specified directive class has been applied, as demonstrated in the “Testing Input Properties” section.

In the listing, I use the `By.css` method to locate the first `span` element in the template and access the `DOM` object that represents it through the `nativeElement` property so that I can check the value of the `textContent` property in the unit tests.

Notice that after each change to the component's `category` property, I call the `ComponentFixture` object's `detectChanges` method, like this:

```
...
component.category = "Soccer";
fixture.detectChanges();
expect(component.getProducts().length).toBe(2);
expect(bindingElement.textContent).toContain("2");
...

```

This method tells the Angular testing environment to process any changes and evaluate the data binding expressions in the template. Without this method call, the change to the value of the `category` component would not be reflected in the template, and the test would fail.

Testing a Component with an External Template

Angular components are compiled into factory classes, either within the browser or by the ahead-of-time compiler that I demonstrated in Chapter 8. As part of this process, Angular processes any external templates and includes them as text in the JavaScript code that is generated similar to an inline template. When unit testing a component with an external template, the compilation step must be performed explicitly. In Listing 29-11, I changed the `@Component` decorator applied to the `FirstComponent` class so that it specifies an external template.

Listing 29-11. Specifying a Template in the `first.component.ts` File in the `src/app/ondemand` Folder

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";

@Component({
  selector: "first",
  templateUrl: "first.component.html"
})
export class FirstComponent {

  constructor(private repository: Model) {}

  category: string = "Soccer";

  getProducts(): Product[] {
    return this.repository.getProducts()
      .filter(p => p.category == this.category);
  }
}
```

To provide the template, I created a file called `first.component.html` in the `exampleApp/app/ondemand` folder and added the elements shown in Listing 29-12.

Listing 29-12. The first.component.html File in the exampleApp/app/ondemand Folder

```
<div class="bg-primary text-white p-2">
    There are
        <span class="strong"> {{getProducts().length}} </span>
    products
</div>
```

This is the same content that was previously defined inline. Listing 29-13 updates the unit test for the component to deal with the external template by explicitly compiling the component.

Listing 29-13. Compiling a Component in the first.component.spec.ts File in the src/app/tests Folder

```
import { TestBed, ComponentFixture, waitForAsync } from "@angular/core/testing";
import { FirstComponent } from "../ondemand/first.component";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { DebugElement } from "@angular/core";
import { By } from "@angular/platform-browser";

describe("FirstComponent", () => {
    let fixture: ComponentFixture<FirstComponent>;
    let component: FirstComponent;
    let debugElement: DebugElement;
    let bindingElement: HTMLSpanElement;
    let spanElement: HTMLSpanElement;

    let mockRepository = {
        getProducts: function () {
            return [
                new Product(1, "test1", "Soccer", 100),
                new Product(2, "test2", "Chess", 100),
                new Product(3, "test3", "Soccer", 100),
            ]
        }
    }

    beforeEach(waitForAsync(() => {
        TestBed.configureTestingModule({
            declarations: [FirstComponent],
            providers: [
                { provide: Model, useValue: mockRepository }
            ]
        });
        TestBed.compileComponents().then(() => {
            fixture = TestBed.createComponent(FirstComponent);
            component = fixture.componentInstance;
            debugElement = fixture.debugElement;
            spanElement = debugElement.query(By.css("span")).nativeElement;
        });
    }));
});
```

```

it("filters categories", () => {
  component.category = "Chess"
  fixture.detectChanges();
  expect(component.getProducts().length).toBe(1);
  expect(bindingElement.textContent).toContain("1");

  component.category = "Soccer";
  fixture.detectChanges();
  expect(component.getProducts().length).toBe(2);
  expect(bindingElement.textContent).toContain("2");

  component.category = "Running";
  fixture.detectChanges();
  expect(component.getProducts().length).toBe(0);
  expect(bindingElement.textContent).toContain("0");
});

});

```

Components are compiled using the `TestBed.compileComponents` method. The compilation process is asynchronous, and the `compileComponents` method returns a `Promise`, which must be used to complete the test setup when the compilation is complete. To make it easier to work with asynchronous operations in unit tests, the `@angular/core/testing` module contains a function called `waitForAsync`, which is used with the `beforeEach` method.

Testing Component Events

To demonstrate how to test for a component's response to events, I defined a new property in the `FirstComponent` class and added a method to which the `@HostBinding` decorator has been applied, as shown in Listing 29-14.

Listing 29-14. Adding Event Handling in the `first.component.ts` File in the `src/app/ondemand` Folder

```

import { Component, HostListener } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";

@Component({
  selector: "first",
  templateUrl: "first.component.html"
})
export class FirstComponent {

  constructor(private repository: Model) {}

  category: string = "Soccer";
  highlighted: boolean = false;

  getProducts(): Product[] {
    return this.repository.getProducts()
      .filter(p => p.category == this.category);
  }
}

```

```

@HostListener("mouseenter", ["$event.type"])
@HostListener("mouseleave", ["$event.type"])
setHighlight(type: string) {
  this.highlighted = type == "mouseenter";
}
}

```

The `setHighlight` method has been configured so that it will be invoked when the host element's `mouseenter` and `mouseleave` events are triggered. Listing 29-15 updates the component's template so that it uses the new property in a data binding.

Listing 29-15. Binding to a Property in the first.component.html File in the src/app/ondemand Folder

```

<div class="bg-primary text-white p-2" [class.bg-success]="highlighted">
  There are
  <span class="strong"> {{getProducts().length}} </span>
  products
</div>

```

Events can be triggered in unit tests through the `triggerEventHandler` method defined by the `DebugElement` class, as shown in Listing 29-16.

Listing 29-16. Triggering Events in the first.component.spec.ts File in the src/app/tests Folder

```

import { TestBed, ComponentFixture, waitForAsync } from "@angular/core/testing";
import { FirstComponent } from "../ondemand/first.component";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { DebugElement } from "@angular/core";
import { By } from "@angular/platform-browser";

describe("FirstComponent", () => {

  let fixture: ComponentFixture<FirstComponent>;
  let component: FirstComponent;
  let debugElement: DebugElement;
  // let bindingElement: HTMLSpanElement;
  // let spanElement: HTMLSpanElement;
  let divElement: HTMLDivElement;

  let mockRepository = {
    getProducts: function () {
      return [
        new Product(1, "test1", "Soccer", 100),
        new Product(2, "test2", "Chess", 100),
        new Product(3, "test3", "Soccer", 100),
      ]
    }
  }
}

```

```

beforeEach(waitForAsync(() => {
  TestBed.configureTestingModule({
    declarations: [FirstComponent],
    providers: [
      { provide: Model, useValue: mockRepository }
    ]
  });
  TestBed.compileComponents().then(() => {
    fixture = TestBed.createComponent(FirstComponent);
    component = fixture.componentInstance;
    debugElement = fixture.debugElement;
    //spanElement = debugElement.query(By.css("span")).nativeElement;
    divElement = debugElement.children[0].nativeElement;
  });
}));

// it("filters categories", () => {
//   component.category = "Chess"
//   fixture.detectChanges();
//   expect(component.getProducts().length).toBe(1);
//   expect(bindingElement.textContent).toContain("1");

//   component.category = "Soccer";
//   fixture.detectChanges();
//   expect(component.getProducts().length).toBe(2);
//   expect(bindingElement.textContent).toContain("2");

//   component.category = "Running";
//   fixture.detectChanges();
//   expect(component.getProducts().length).toBe(0);
//   expect(bindingElement.textContent).toContain("0");
// });

it("handles mouse events", () => {
  expect(component.highlighted).toBeFalsy();
  expect(divElement.classList.contains("bg-success")).toBeFalsy();
  debugElement.triggerEventHandler("mouseenter", new Event("mouseenter"));
  fixture.detectChanges();
  expect(component.highlighted).toBeTruthy();
  expect(divElement.classList.contains("bg-success")).toBeTruthy();
  debugElement.triggerEventHandler("mouseleave", new Event("mouseleave"));
  fixture.detectChanges();
  expect(component.highlighted).toBeFalsy();
  expect(divElement.classList.contains("bg-success")).toBeFalsy();
});

});

```

The test in this listing checks the initial state of the component and the template and then triggers the mouseenter and mouseleave events, checking the effect that each has.

Testing Output Properties

Testing output properties is a simple process because the `EventEmitter` objects used to implement them are `Observable` objects that can be subscribed to in unit tests. Listing 29-17 adds an output property to the component under test.

Listing 29-17. Adding an Output Property in the first.component.ts File in the src/app/ondemand Folder

```
import { Component, HostListener, Output, EventEmitter } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";

@Component({
  selector: "first",
  templateUrl: "first.component.html"
})
export class FirstComponent {

  constructor(private repository: Model) {}

  category: string = "Soccer";
  highlighted: boolean = false;

  @Output("pa-highlight")
  change = new EventEmitter<boolean>();

  getProducts(): Product[] {
    return this.repository.getProducts()
      .filter(p => p.category == this.category);
  }

  @HostListener("mouseenter", ["$event.type"])
  @HostListener("mouseleave", ["$event.type"])
  setHighlight(type: string) {
    this.highlighted = type == "mouseenter";
    this.change.emit(this.highlighted);
  }
}
```

The component defines an output property called `change`, which is used to emit an event when the `setHighlight` method is called. Listing 29-18 shows a unit test that targets the output property.

Listing 29-18. Testing an Output Property in the first.component.spec.ts File in the src/app/tests Folder

```
import { TestBed, ComponentFixture, waitForAsync } from "@angular/core/testing";
import { FirstComponent } from "../ondemand/first.component";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { DebugElement } from "@angular/core";
import { By } from "@angular/platform-browser";
```

```

describe("FirstComponent", () => {
  let fixture: ComponentFixture<FirstComponent>;
  let component: FirstComponent;
  let debugElement: DebugElement;
  let divElement: HTMLDivElement;

  let mockRepository = {
    getProducts: function () {
      return [
        new Product(1, "test1", "Soccer", 100),
        new Product(2, "test2", "Chess", 100),
        new Product(3, "test3", "Soccer", 100),
      ]
    }
  }

  beforeEach(waitForAsync(() => {
    TestBed.configureTestingModule({
      declarations: [FirstComponent],
      providers: [
        { provide: Model, useValue: mockRepository }
      ]
    });
    TestBed.compileComponents().then(() => {
      fixture = TestBed.createComponent(FirstComponent);
      component = fixture.componentInstance;
      debugElement = fixture.debugElement;
      divElement = debugElement.children[0].nativeElement;
    });
  }));
  // it("handles mouse events", () => {
  //   expect(component.highlighted).toBeFalsy();
  //   expect(divElement.classList.contains("bg-success")).toBeFalsy();
  //   debugElement.triggerEventHandler("mouseenter", new Event("mouseenter"));
  //   fixture.detectChanges();
  //   expect(component.highlighted).toBeTruthy();
  //   expect(divElement.classList.contains("bg-success")).toBeTruthy();
  //   debugElement.triggerEventHandler("mouseleave", new Event("mouseleave"));
  //   fixture.detectChanges();
  //   expect(component.highlighted).toBeFalsy();
  //   expect(divElement.classList.contains("bg-success")).toBeFalsy();
  // });

  it("implements output property", () => {
    let highlighted: boolean = false;
    component.change.subscribe(value => highlighted = value);
    debugElement.triggerEventHandler("mouseenter", new Event("mouseenter"));
    expect(highlighted).toBeTruthy();
  });
}

```

```

        debugElement.triggerEventHandler("mouseleave", new Event("mouseleave"));
        expect(highlighted).toBeFalsy();
    });
});

```

I could have invoked the component's `setHighlight` method directly in the unit test, but instead I have chosen to trigger the `mouseenter` and `mouseleave` events, which will activate the output property indirectly. Before triggering the events, I use the `subscribe` method to receive the event from the `output` property, which is then used to check for the expected outcomes.

Testing Input Properties

The process for testing input properties requires a little extra work. To get started, I added an input property to the `FirstComponent` class that is used to receive the data model repository, replacing the service that was received by the constructor, as shown in Listing 29-19. I have also removed the host event bindings and the `output` property to keep the example simple.

Listing 29-19. Adding an Input Property in the `first.component.ts` File in the `src/app/ondemand` Folder

```

import { Component, Input } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";

@Component({
  selector: "first",
  templateUrl: "first.component.html"
})
export class FirstComponent {
  constructor(private repository: Model) {}

  category: string = "Soccer";
  highlighted: boolean = false;

  // @Output("pa-highlight")
  // change = new EventEmitter<boolean>();

  getProducts(): Product[] {
    return this.repository == null ? [] : this.repository.getProducts()
      .filter(p => p.category == this.category);
  }

  // @HostListener("mouseenter", ["$event.type"])
  // @HostListener("mouseleave", ["$event.type"])
  // setHighlight(type: string) {
  //   this.highlighted = type == "mouseenter";
  //   this.change.emit(this.highlighted);
  // }

  @Input("pa-model")
  model?: Model;
}

```

The input property is set using an attribute called `pa-model` and is used within the `getProducts` method. Listing 29-20 shows how to write a unit test that targets the `input` property.

Listing 29-20. Testing an Input Property in the `first.component.spec.ts` File in the `src/app/tests` Folder

```
import { TestBed, ComponentFixture, waitForAsync } from "@angular/core/testing";
import { FirstComponent } from "../ondemand/first.component";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { DebugElement } from "@angular/core";
import { By } from "@angular/platform-browser";
import { Component, ViewChild } from "@angular/core";

@Component({
  template: `<first [pa-model]="model"></first>`
})
class TestComponent {

  constructor(public model: Model) { }

  @ViewChild(FirstComponent)
  firstComponent!: FirstComponent;
}

describe("FirstComponent", () => {

  let fixture: ComponentFixture<TestComponent>;
  let component: FirstComponent;
  let debugElement: DebugElement;
  let divElement: HTMLDivElement;

  let mockRepository = {
    getProducts: function () {
      return [
        new Product(1, "test1", "Soccer", 100),
        new Product(2, "test2", "Chess", 100),
        new Product(3, "test3", "Soccer", 100),
      ]
    }
  }

  beforeEach(waitForAsync(() => {
    TestBed.configureTestingModule({
      declarations: [FirstComponent, TestComponent],
      providers: [
        { provide: Model, useValue: mockRepository }
      ]
    });
    TestBed.compileComponents().then(() => {
      fixture = TestBed.createComponent(TestComponent);
      fixture.detectChanges();
      component = fixture.componentInstance.firstComponent;
    })
  }));
})
```

```

        debugElement = fixture.debugElement.query(By.directive(FirstComponent));
    });
});

// it("implements output property", () => {
//     let highlighted: boolean = false;
//     component.change.subscribe(value => highlighted = value);
//     debugElement.triggerEventHandler("mouseenter", new Event("mouseenter"));
//     expect(highlighted).toBeTruthy();
//     debugElement.triggerEventHandler("mouseleave", new Event("mouseleave"));
//     expect(highlighted).toBeFalsy();
// });

it("receives the model through an input property", () => {
    component.category = "Chess";
    fixture.detectChanges();
    let products = mockRepository.getProducts()
        .filter(p => p.category == component.category);
    let componentProducts = component.getProducts();
    for (let i = 0; i < componentProducts.length; i++) {
        expect(componentProducts[i]).toEqual(products[i]);
    }
    expect(debugElement.query(By.css("span")).nativeElement.textContent)
        .toContain(products.length);
});
});

```

The trick here is to define a component that is only required to set up the test and whose template contains an element that matches the selector of the component you want to target. In this example, I defined a component class called `TestComponent` with an inline template defined in the `@Component` decorator that contains a `first` element with a `pa-model` attribute, which corresponds to the `@Input` decorator applied to the `FirstComponent` class.

The test component class is added to the declarations array for the testing module, and an instance is created using the `TestBed.createComponent` method. I used the `@ViewChild` decorator in the `TestComponent` class so that I can get hold of the `FirstComponent` instance I require for the test. To get the `FirstComponent` root element, I used the `DebugElement.query` method with the `By.directive` method.

The result is that I can access both the component and its root element for the test, which sets the `category` property and then validates the results both from the component and via the data binding in its template.

Testing an Angular Directive

The process for testing directives is similar to the one required to test input properties, in that a test component and template are used to create an environment for testing in which the directive can be applied. To have a directive to test, I added a file called `attr.directive.ts` to the `src/app/ondemand` folder and added the code shown in Listing 29-21.

Note I have shown an attribute directive in this example, but the technique in this section can be used to test structural directives equally well.

Listing 29-21. The Contents of the attr.directive.ts File in the src/app/ondemand Folder

```
import {
  Directive, ElementRef, Attribute, Input, SimpleChange
} from "@angular/core";

@Directive({
  selector: "[pa-attr]"
})
export class PaAttrDirective {

  constructor(private element: ElementRef) { }

  @Input("pa-attr")
  bgClass?: string;

  ngOnChanges(changes: { [property: string]: SimpleChange }) {
    let change = changes["bgClass"];
    let classList = this.element.nativeElement.classList;
    if (!change.isFirstChange() && classList.contains(change.previousValue)) {
      classList.remove(change.previousValue);
    }
    if (!classList.contains(change.currentValue)) {
      classList.add(change.currentValue);
    }
  }
}
```

This is an attribute directive based on an example from Chapter 13. To create a unit test that targets the directive, I added a file called attr.directive.spec.ts to the src/app/tests folder and added the code shown in Listing 29-22.

Listing 29-22. The Contents of the attr.directive.spec.ts File in the src/app/tests Folder

```
import { TestBed, ComponentFixture } from "@angular/core/testing";
import { Component, DebugElement, ViewChild } from "@angular/core";
import { By } from "@angular/platform-browser";
import { PaAttrDirective } from "../ondemand/attr.directive";

@Component({
  template: `<div><span [pa-attr]="className">Test Content</span></div>`
})
class TestComponent {
  className = "initialClass"

  @ViewChild(PaAttrDirective)
  attrDirective!: PaAttrDirective;
}

describe("PaAttrDirective", () => {
  let fixture: ComponentFixture<TestComponent>;
```

```

let directive: PaAttrDirective;
let spanElement: HTMLSpanElement;

beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [TestComponent, PaAttrDirective],
  });
  fixture = TestBed.createComponent(TestComponent);
  fixture.detectChanges();
  directive = fixture.componentInstance.attrDirective;
  spanElement = fixture.debugElement.query(By.css("span")).nativeElement;
});

it("generates the correct number of elements", () => {
  fixture.detectChanges();
  expect(directive.bgClass).toBe("initialClass");
  expect(spanElement.className).toBe("initialClass");

  fixture.componentInstance.className = "nextClass";
  fixture.detectChanges();
  expect(directive.bgClass).toBe("nextClass");
  expect(spanElement.className).toBe("nextClass");
});
});
});

```

The text component has an inline template that applies the directive and a property that is referred to in the data binding. The `@ViewChild` decorator provides access to the directive object that Angular creates when it processes the template, and the unit test can check that changing the value used by the data binding has an effect on the directive object and the element it has been applied to.

Summary

In this chapter, I demonstrated the different ways in which Angular components and directives can be unit tested. I explained the process of installing the test framework and tools and how to create the testbed through which tests are applied. I demonstrated how to test the different aspects of components and how the same techniques can be applied to directives as well.

That is all I have to teach you about Angular. I started by creating a simple application and then took you on a comprehensive tour of the different building blocks in the framework, showing you how they can be created, configured, and applied to create web applications.

I wish you every success in your Angular projects, and I can only hope that you have enjoyed reading this book as much as I enjoyed writing it.

Index

A

Ajax, *see* Web services

@angular/cli, 213

installing, 13

ng new, 14

ng serve command, 15

@angular/forms module, 304

Angular Material, 24, 171, 175, 178

Animations, 792

adding and removing
elements, 803

applying framework styles, 813

built-in states, 801

defining, 792

element states, 794

enabling, 792

guidance for use, 795

parallel effects, 808

style groups, 793, 809

timing functions, 805

transitions, 794, 802

triggers, 796

Applications

round-trip, 4

single-page, 4

Authentication, *see* SportsStore

B

Bootstrap CSS framework, 50, 178, 221, 227,
229, 579

Browser, choosing, 13

Building application, production, 203

C

Cascading Style Sheets (CSS), 50

Change detection, NgZone class, 689

Component libraries

additional styles, 827

Angular Material, 823

choosing, 823

Covalent, 823

data APIs, 833

feature modules, 827

installing, 822

Material Design, 823

mixing styles packages, 827

ng-bootstrap, 823

ngx-bootstrap, 823

sass files, 839

scss files, 839

themes, 839

using components, 825

Components, 239

application structure, 407

@Component decorator, 409

content projection, 422

creating, 408

decorator, 239

dynamic, 424

input properties, 416

lifecycle methods

ngAfterViewChecked, 437

ngAfterViewInit, 437

output properties, 420

styles

external, 427

inline, 425

shadow DOM, 428

view encapsulation, 428

template queries, 436

@ViewChild decorator, 437

@ViewChildren decorator, 437

templates

data bindings, 415

external, 414

inline, 413

Cross-origin HTTP requests (CORS), 680

CSS stylesheets

configuring, 227

style bundle, 227

D

Data bindings, 19
 attribute bindings, 254, 256, 259
 class bindings, 254
 classes, 261
 directive, 253
 event binding, 306
 brackets, 307
 event data, 310
 expression, 307
 filtering key events, 315
 host element, 307
 template references variables, 314
 expressions, 252, 254
 host element, 252, 256
 live data updates, 270
 one-way bindings, 251, 252
 property bindings, 254, 256
 restrictions, 297
 idempotent expressions, 297
 limited expression context, 300
 square brackets, 252, 255
 string interpolation, 258
 style bindings, 254
 styles, 261
 target, 252
 two-way bindings, 315, 317
 Data model, 17, 21, 33, 108, 729
 Dependency injection, *see* Services
 Development environment, 6, 10, 11
 Directives, 253
 attribute directives, 344
 data-bound inputs, 350
 host element attributes, 347
 built-in directives, 278
 custom directive, 123
 custom events, 355, 356
 @Directive decorator, 372
 host element bindings, 358
 host element content, 395
 @ContentChild decorator, 396
 @ContentChildren decorator, 399
 @Input decorator, 351
 lifecycle hooks, 352
 micro-templates, 280
 ngClass, 253
 ngClass directive, 263
 ng-container element, 297
 ngFor directive, 284
 even variable, 285
 expanding micro-template syntax, 289
 first variable, 285
 index variable, 285

of keyword, 284
 let keyword, 284
 minimizing changes, 290
 odd variable, 285
 trackBy, 293
 using variables in child elements, 285
 ngIf directive, 253, 279, 282
 ngModel directive, 317
 ngStyle directive, 253, 266
 ngSwitch directive, 281
 ngTemplateOutlet, 253
 ngTemplateOutlet directive, 294
 context data, 295
 ng-template element, 294
 @Output decorator, 355
 structural directives, 369
 collection changes, 384
 concise syntax, 375
 context data, 379
 detecting changes, 372
 iterating directives, 376
 ngDoCheck method, 385
 ng-template element, 373
 property changes, 383
 ViewContainerRef class, 372
 using services, 500
 Docker containers, 195, 207, 208
 DOM Events, common properties, 311

E

Editor, choosing, 13
 Errata, reporting, 7
 Events, 306

F, G

Forms, 319
 API, 595
 dynamic forms, 635
 FormArray class, 635
 adding controls, 641
 methods, 635
 properties, 635
 removing controls, 641
 validating controls, 643
 FormControl class, 596
 change frequency, 601
 constructor, 601
 events, 601
 state, 602
 updateOn property, 601
 formControl directive, 594, 596, 598, 607, 609, 616

formControlName directive, 616, 622, 628, 641
 FormGroup class, 612
 resetting, 614
 setting values, 614
 formGroup directive, 616–618
 nesting form elements, 625
 observable properties, 599
 reactive forms, 595
 ReactiveFormsModule, 596
 validation, 322, 605, 622
 asynchronous, 660
 custom, 648, 654
 directives, 650
 validation classes, 323, 338
 whole-form validation, 331

H, I

HTML
 attributes, 47
 literal values, 48
 without values, 47
 document object model, 49
 document structure, 49
 elements, 46
 content, 48
 hierarchy, 48
 tags, 46
 void elements, 47

J, K

JavaScript, 52
 access control, 89
 arrays, 79
 built-in methods, 83
 enumerating, 81
 modifying, 80
 reading, 80
 spread operator, 82
 boolean type, 56
 classes, 87
 inheritance, 90
 closures, 79
 coalescing values, 70
 conditional statements, 65
 constructor, 89
 functions
 as arguments to other functions, 77
 default parameters, 76
 defining, 74
 optional parameters, 75
 rest parameters, 76
 results, 77

literal values in directive expressions, 282
 modules, 92
 export keyword, 92
 import keyword, 93
 NPM packages, 94
 resolution, 94
 null, 56
 null coalescing operator, 70
 nullish coalescing operator, 70
 number type, 56
 objects
 literal syntax, 84
 optional properties, 86
 operators, 64, 66
 optional chaining operator, 71
 primitive types, 56
 statements, conditional, 65
 string type, 56
 template strings, 63
 truthy and falsy values, 67, 263
 types, 62
 booleans, 62
 converting explicitly, 67
 null, 64
 numbers, 64
 strings, 62, 63
 undefined, 64
 undefined, 56
 variable closure, 79
 variables and constants, 60
 JSON Web Token, 161

L

Linting, ESLint, 230
 Listings
 complete, 7
 partial, 8
 Live data model, 34

M

Material Design, 823, 842
 Micro-templates, use by directives, 280
 Modules
 bootstrap property, 553
 declarations property, 552
 dynamic (*see* URL routing)
 dynamic loading, SportsStore, 156
 feature modules, creating, 555
 imports property, 552
 JavaScript modules, 561
 @NgModule decorator, 551
 providers property, 553
 root module, 550

■ N, O

ng add Command, 221, 222, 823
 ng command, 14
 ng config Command, 227, 228
 ng-container element, 276, 296, 297
 ng lint command, 234
 ng new command, 237, 238, 246, 696, 852
 ng serve command, 16, 26, 115, 145, 151, 171, 172, 188, 221, 223
Node.js
 installing, 11
 NPM, 12
 package manager, 12
Node Package Manager (NPM), 12

■ P, Q

Pipes
 applying, 444, 445
 async pipe, 479, 480
 combining, 449
 creating, 445
 formatting currency amounts, 458
 formatting dates, 463
 formatting numbers, 454
 formatting percentages, 461
 formatting string case, 469
 impure pipes, 450
 JSON serialization, 471
 key/value pairs, 474
 @Pipe decorator, 446
 pluralizing values, 477
 pure pipes, 449
 selecting values, 475, 477
 slicing arrays, 472
 using services, 497
Polyfills, 218, 225, 226
Progressive Web Applications, 5, 195, 200, 204
Projects

- ahead-of-time compilation, 241
- angular.json file, 215
- AoT compilation, 241
- build process, 224
- bundles, 224
- components, 239
- contents, 214
- data model, 242
- development tools, 223
- .editorconfig file, 216
- .gitignore file, 216
- hot reloading, 226
- HTML document, 236
- node_modules folder, 215, 218

package.json file, 216
packages, 218, 221

- global packages, 220
- scripts, 220
- versions, 219

root module, 238
src/app folder, 217
src/assets folder, 217
src/environments folder, 217
src folder, 217
src/index.html file, 222
src/main.ts file, 228
structure, 214
tsconfig.json file, 215
tslint.json file, 215
webpack, 224

■ R

React, 5, 94
Reactive extensions, 94

- async pipe, 480
- Observable, subscribe method, 95
- Observer, 96
- Subject, types of, 97

Reactive forms, 593–595
REST, *see* **Web services**
RESTful web services, *see* **Web services**
Root module, 106, 107, 238, 667, 696, 775, 778, 792
Round-trip applications, 4
RxJS, 73, 94

■ S

Sass, 839, 841, 843, 848
Schematics API, 221, 222
Services

- component isolation, 505
- dependency injection, 491
- @Host decorator, 543
- @Injectable decorator, 491
- local providers, 534
 - providers property, 541
 - viewProviders property, 542
- @Optional decorator, 544
- providers, 516
 - class provider, 519
 - existing service provider, 533
 - factory provider, 530
 - multiple service objects, 526
 - service tokens, 520
 - value provider, 528
- providers property, 494
- receiving services, 492

registering services, 494
 services in directives, 500
 services in pipes, 497
 shared object problem, 485
`@SkipSelf` decorator, 544

Single-page applications, 4

SportsStore
 additional packages, 100
 Angular Material, 171
 authentication, JSON Web Token, 161
 bootstrap file, 107
 cart, summary component, 127, 130
 category selection, 117
 component library, 171
 containerizing, 206
 creating the container, 208
 creating the image, 207
 deployment packages, 206
 Dockerfile, 207
 stopping the container, 209
 creating the project, 99
 data model, 108, 109
 data source, 109
 displaying products, 115
 dynamic module, 156
 navigation, 137
 orders, 145
 pagination, 119
 persistent data, 199
 production build, 203
 progressive features, 195
 caching, 196
 connectivity, 197
 project structure, 104
 REST data, 152
 root component, 106
 root module, 106
 route guard, 140
 URL routing, 134
 web service, 101
 String interpolation, 258

T

Templates, variables, 36
 TypeScript, 52
 any type, 54
 concise constructor, 18
 specific types, 55
 type annotation, 54
 type union, 57
 variables and constants, 60

U

Unit testing
 components, 855
 configuring dependencies, 858
 data bindings, 860
 events, 864
 input properties, 869
 output properties, 867
 templates, 862
 directives, 871
 Jasmine, 852, 854
 Karma test runner, 852
 ng test command, 852
 TestBed class, 856
 URL routing, 134, 693
 ActivatedRoute class, 703
 basic configuration, 694
 change notifications, 736
 child routes, 743
 parameters, 746
 route outlets, 744
 dynamic modules, 773
 guarding, 777
 specifying, 775
 using, 776
 guarding, 140
 guards, 752
 displaying a loading message, 758
 preventing navigation, 760
 preventing route activation, 761
 resolvers, 753, 767
 named outlets, 780, 783
 navigating within a component, 736
 navigation events, 716
 navigation links, 698
 optional URL segments, 711
 programmatic navigation, 703, 713
 redirections, 734
 route parameters, 705
 routerLink directive, 698
 router-outlet element, 696, 780
 Routes class, 694
 styles for active elements, 738
 wildcard routes, 731

V

Vue.js, 5

■ INDEX

■ **W, X, Y, Z**

- Web services, 669
 - cross-origin requests, 680
 - errors, 685
 - HttpClient class, 671
 - consolidating requests, 678
- methods, 671
- responses, 672
- HTTP verbs, 670
- JSONP requests, 681
- NgZone class, 689
- request headers, 683