CHAPTER 9

SportsStore: Administration

In this chapter, I continue building the SportsStore application by adding administration features. Relatively few users will need to access the administration features, so it would be wasteful to force all users to download the administration code and content when it is unlikely to be used. Instead, I am going to put the administration features in a feature module that will be loaded only when administration is required. In this section, I prepare the application by creating the feature module, adding some initial content, and setting up the rest of the application so that the module can be loaded dynamically.

Preparing the Example Application

No preparation is required for this chapter, which continues using the SportsStore project from Chapter 8. To start the RESTful web service, open a command prompt and run the following command in the `SportsStore` folder:

```
npm run json
```

Open a second command prompt and run the following command in the `SportsStore` folder to start the development tools and HTTP server:

```
ng serve --port 3000 --open
```

Tip The free source code download from `apress.com` that accompanies this book contains the SportsStore project from each individual chapter if you don't want to have to start with the earlier chapters.

Creating the Module

The process for creating the feature module follows the same pattern for the rest of the application, except it is important that no other part of the application has dependencies on the module or the classes it contains, which would undermine the dynamic loading of the module and cause the JavaScript module to load the administration code, even if it is not used.

The starting point for the administration features will be to authenticate, so I created a file called `auth.component.ts` in the `SportsStore/src/app/admin` folder and used it to define the component shown in Listing 9-1.

Listing 9-1. The Content of the auth.component.ts File in the SportsStore/src/app/admin Folder

```
import { Component } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Router } from "@angular/router";

@Component({
    moduleId: module.id,
    templateUrl: "auth.component.html"
})
export class AuthComponent {
    public username: string;
    public password: string;
    public errorMessage: string;

    constructor(private router: Router) {}

    authenticate(form: NgForm) {
        if (form.valid) {
            // perform authentication
            this.router.navigateByUrl("/admin/main");
        } else {
            this.errorMessage = "Form Data Invalid";
        }
    }
}
```

The component defines properties for the username and password that will be used to authenticate the user, an `errorMessage` property that will be used to display messages to the

user when there are problems, and an `authenticate` method that will perform the authentication process (but that does nothing at the moment).

To provide the component with a template, I created a file called `auth.component.html` in the `SportsStore/src/app/admin` folder and added the content shown in Listing 9-2.

Listing 9-2. The Content of the auth.component.html File in the SportsStore/src/app/admin Folder

```
<div class="bg-info p-a-1 text-xs-center">
    <h3>SportsStore Admin</h3>
</div>
<div class="bg-danger m-t-1 p-a-1 text-xs-center"
        *ngIf="errorMessage != null">
    {{errorMessage}}
</div>
<div class="p-a-1">
    <form novalidate #form="ngForm" (ngSubmit)="authenticate(form)">
        <div class="form-group">
            <label>Name</label>
            <input class="form-control" name="username"
                [(ngModel)]="username" required />
        </div>
        <div class="form-group">
            <label>Password</label>
            <input class="form-control" type="password" name="password"
                [(ngModel)]="password" required />
        </div>
        <div class="text-xs-center">
            <button class="btn btn-secondary" routerLink="/">Go back</button>
            <button class="btn btn-primary" type="submit">Log In</button>
        </div>
    </form>
</div>
```

The template contains an HTML form that uses two-way data binding expressions for the component's properties. There is a button that will submit the form, a button that navigates back to the root URL, and a `div` element that is visible only when there is an error message to display.

To create a placeholder for the administration features, I added a file called `admin.component.ts` in the `SportsStore/src/app/admin` folder and defined the component shown in Listing 9-3.

Listing 9-3. The Contents of the admin.component.ts File in the SportsStore/src/app/admin Folder

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

@Component({
```

```
    moduleId: module.id,
    templateUrl: "admin.component.html"
})
export class AdminComponent {}
```

The component doesn't contain any functionality at the moment. To provide a template for the component, I added a file called `admin.component.html` to the `SportsStore/src/app/admin` folder and the placeholder content shown in Listing 9-4.

Listing 9-4. The Contents of the admin.component.html File in the SportsStore/src/app/admin Folder

```
<div class="bg-info p-a-1">
    <h3>Placeholder for Admin Features</h3>
</div>
```

To define the feature module, I added a file called `admin.module.ts` in the `SportsStore/src/app/admin` folder and added the code shown in Listing 9-5.

Listing 9-5. The Contents of the admin.module.ts File in the SportsStore/src/app/admin Folder

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { FormsModule } from "@angular/forms";
import { RouterModule } from "@angular/router";
import { AuthComponent } from "./auth.component";
import { AdminComponent } from "./admin.component";

let routing = RouterModule.forChild([
    { path: "auth", component: AuthComponent },
    { path: "main", component: AdminComponent },
    { path: "**", redirectTo: "auth" }
]);

@NgModule({
    imports: [CommonModule, FormsModule, routing],
    declarations: [AuthComponent, AdminComponent]
})
export class AdminModule { }
```

The main difference when creating a dynamically loaded module is that the feature module must be self-contained and include all the information that Angular requires, including the routing URLs that are supported and the components they display.

The `RouterModule.forChild` method is used to define the routing configuration for the feature module, which is then included in the module's `imports` property.

The prohibition on exporting classes from a dynamically loaded module doesn't apply to imports. This module relies on the functionality in the model feature module, which has been

added to the module's `imports` so that components can access the model classes and the repositories.

Configuring the URL Routing System

Dynamically loaded modules are managed through the routing configuration, which triggers the loading process when the application navigates to a specific URL. Listing 9-6 extends the routing configuration of the application so that the `/admin` URL will load the administration feature module.

Listing 9-6. Configuring a Dynamically Loaded Module in the app.module.ts File

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { AppComponent } from "./app.component";
import { StoreModule } from "./store/store.module";
import { StoreComponent } from "./store/store.component";
import { CheckoutComponent } from "./store/checkout.component";
import { CartDetailComponent } from "./store/cartDetail.component";
import { RouterModule } from "@angular/router";
import { StoreFirstGuard } from "./storeFirst.guard";

@NgModule({
    imports: [BrowserModule, StoreModule,
        RouterModule.forRoot([
            {
                path: "store", component: StoreComponent,
                canActivate: [StoreFirstGuard]
            },
            {
                path: "cart", component: CartDetailComponent,
                canActivate: [StoreFirstGuard]
            },
            {
                path: "checkout", component: CheckoutComponent,
                canActivate: [StoreFirstGuard]
            },
            {
                path: "admin",
                loadChildren: "app/admin/admin.module#AdminModule",
                canActivate: [StoreFirstGuard]
            },
            { path: "**", redirectTo: "/store" }
        ])],
    providers: [StoreFirstGuard],
    declarations: [AppComponent],
```

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

The new route tells Angular that when the application navigates to the `/admin` URL, it should load a feature module defined by a class called `AdminModule` from the `/app/admin/admin.module.ts` file. When Angular processes the admin module, it will incorporate the routing information it contains into the overall set of routes and complete the navigation.

Navigating to the Administration URL

The final preparatory step is to provide the user with the ability to navigate to the `/admin` URL so that the administration feature module will be loaded and its component displayed to the user. Listing 9-7 adds a button to the store component's template that will perform the navigation.

Listing 9-7. Adding a Navigation Button in the store.component.html File

```
<div class="navbar navbar-inverse bg-inverse">
    <a class="navbar-brand">SPORTS STORE</a>
    <cart-summary></cart-summary>
</div>
<div class="col-xs-3 p-a-1">
    <button class="btn btn-block btn-outline-primary"
            (click)="changeCategory()">
        Home
    </button>
    <button *ngFor="let cat of categories"
            class="btn btn-outline-primary btn-block"
            [class.active]="cat == selectedCategory"
            (click)="changeCategory(cat)">
        {{cat}}
    </button>
    <button class="btn btn-block btn-danger m-t-3" routerLink="/admin">
        Admin
    </button>
</div>
<div class="col-xs-9 p-a-1">

    <!-- ...elements omitted for brevity... -->

</div>
```

To see the result, use the browser's F12 developer tools to see the network requests made by the browser as the application is loaded. The files for the administration module will not be

loaded until you click the Admin button, at which point Angular will request the files and display the login page shown in Figure 9-1.

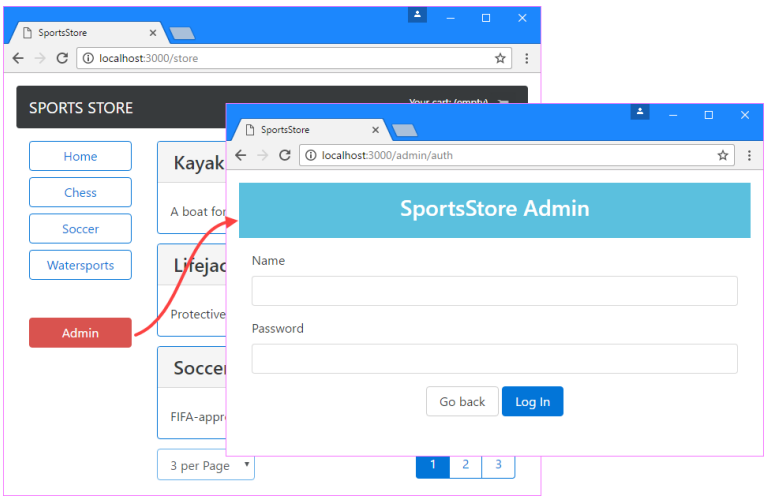

Figure 9-1. Using a dynamically loaded module

Enter any name and password into the form fields and click the Log In button to see the placeholder content, as shown in Figure 9-2. If you leave either of the form fields empty, a warning message will be displayed.

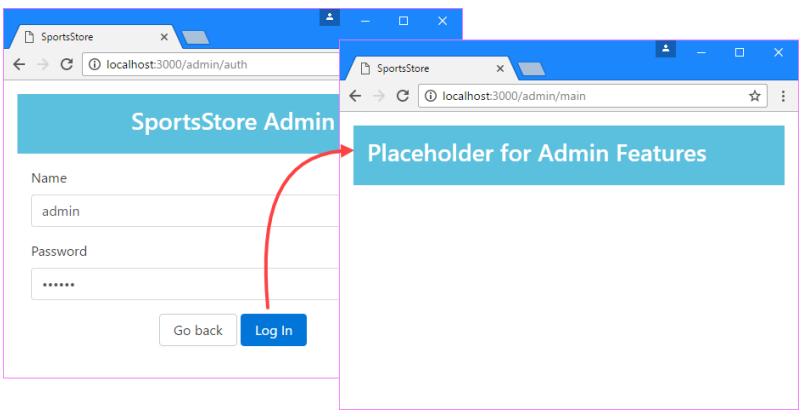

Figure 9-2. The placeholder administration features

Implementing Authentication

The RESTful web service has been configured so that it requires authentication for the requests that the administration feature will require. In the sections that follow, I extend the data model to support authenticated HTTP requests and integrate them into the administration module.

Understanding the Authentication System

The result of authentication with the RESTful web service is a JSON Web Token (JWT), which is returned by the server and must be included in any subsequent requests to show that the application is authorized to perform protected operations. You can read the JWT specification at `https://tools.ietf.org/html/rfc7519`, but for the purposes of the SportsStore application it is enough to know that the Angular application can authenticate the user by sending a POST request to the `/login` URL, including a JSON-formatted object in the request body that contains name and password properties. There is only one set of valid credentials in the authentication code used in Chapter 7, which is shown in Table 9-1.

Table 9-1. *The Authentication Credentials Supported by the RESTful Web Service*

Username	Password
admin	secret

As I noted in Chapter 7, you should not hard-code credentials in real projects, but this is the username and password that you will need for the SportsStore application.

If the correct credentials are sent to the `/login` URL, then the response from the RESTful web service will contain a JSON object like this:

```
{
  "success": true,
  "token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjoiYWRtaW4iLCJleHBpcmVz
           SW4iOiIxaCIsImlhdCI6MTQ3ODk1NjI1Mn0.lJaDDrSu-bHBtdWrz0312p_DG5tKypGv6cA
           NgOyzlg8"
}
```

The `success` property describes the outcome of the authentication operation, and the `token` property contains the JWT, which should be included in subsequent requests using the `Authorization` HTTP header in this format:

```
Authorization: Bearer<eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjoiYWRtaW4iLC
                      JleHBpcmVzSW4iOiIxaCIsImlhdCI6MTQ3ODk1NjI1Mn0.lJaDDrSu-bHBtd
                      Wrz0312p_DG5tKypGv6cANgOyzlg8>
```

I configured the JWT tokens returned by the server so they expire after one hour.

If the wrong credentials are sent to the server, then the JSON object returned in the response will just contain a `success` property set to `false`, like this:

```
{
  "success": false
}
```

Extending the Data Source

The RESTful data source is the class that will do most of the work because it is responsible for sending the authentication request to the `/login` URL and including the JWT in later requests. Listing 9-8 adds authentication to the `RestDataSource` class and extends the `sendRequest` method so that it can include the JWT in requests.

Listing 9-8. Adding Authentication in the rest.datasource.ts File

```
import { Injectable } from "@angular/core";
import { Http, Request, RequestMethod } from "@angular/http";
import { Observable } from "rxjs/Observable";
import { Product } from "./product.model";
import { Cart } from "./cart.model";
import { Order } from "./order.model";
import "rxjs/add/operator/map";

const PROTOCOL = "http";
const PORT = 3500;

@Injectable()
export class RestDataSource {
    baseUrl: string;
    auth_token: string;

    constructor(private http: Http) {
        this.baseUrl = `${PROTOCOL}://${location.hostname}:${PORT}/`;
    }

    authenticate(user: string, pass: string): Observable<boolean> {
        return this.http.request(new Request({
```

```
            method: RequestMethod.Post,
            url: this.baseUrl + "login",
            body: { name: user, password: pass }
        })).map(response => {
            let r = response.json();
            this.auth_token = r.success ? r.token : null;
            return r.success;
        });
    }

    getProducts(): Observable<Product[]> {
        return this.sendRequest(RequestMethod.Get, "products");
    }

    saveOrder(order: Order): Observable<Order> {
        return this.sendRequest(RequestMethod.Post,
            "orders", order);
    }

    private sendRequest(verb: RequestMethod,
            url: string, body?: Product | Order, auth: boolean = false)
                : Observable<Product | Product[] | Order | Order[]> {

        let request = new Request({
            method: verb,
            url: this.baseUrl + url,
            body: body
        });
        if (auth && this.auth_token != null) {
            request.headers.set("Authorization", `Bearer<${this.auth_token}>`);
        }
        return this.http.request(request).map(response => response.json());
    }
}
```

Creating the Authentication Service

Rather than expose the data source directly to the rest of the application, I am going to create a service that can be used to perform authentication and determine whether the application has been authenticated. I added a file called `auth.service.ts` in the `SportsStore/src/app/model` folder and defined the class shown in Listing 9-9.

Listing 9-9. The Contents of the auth.service.ts File in the SportsStore/src/app/model Folder

```
import { Injectable } from "@angular/core";
import { Observable } from "rxjs/Observable";
import { RestDataSource } from "./rest.datasource";
```

```
import "rxjs/add/operator/map";

@Injectable()
export class AuthService {

    constructor(private datasource: RestDataSource) {}

    authenticate(username: string, password: string): Observable<boolean> {
        return this.datasource.authenticate(username, password);
    }

    get authenticated(): boolean {
        return this.datasource.auth_token != null;
    }

    clear() {
        this.datasource.auth_token = null;
    }
}
```

The `authenticate` method receives the user's credentials and passes them on to the data source `authenticate` method, returning an `Observable` that will yield `true` if the authentication process has succeeded and `false` otherwise. The `authenticated` property is a getter-only property that returns `true` if the data source has obtained an authentication token. The `clear` method removes the token from the data source.

Listing 9-10 registers the new service with the model feature module. It also adds a `providers` entry for the `RestDataSource` class, which has been used only as a substitute for the `StaticDataSource` class in earlier chapters. Since the `AuthService` class has a `RestDataSource` constructor parameter, it needs its own entry in the module.

Listing 9-10. Configuring the Services in the model.module.ts File

```
import { NgModule } from "@angular/core";
import { ProductRepository } from "./product.repository";
import { StaticDataSource } from "./static.datasource";
import { Cart } from "./cart.model";
import { Order } from "./order.model";
import { OrderRepository } from "./order.repository";
import { RestDataSource } from "./rest.datasource";
import { HttpModule } from "@angular/http";
import { AuthService } from "./auth.service";

@NgModule({
    imports: [HttpModule],
    providers: [ProductRepository, Cart, Order, OrderRepository,
        { provide: StaticDataSource, useClass: RestDataSource },
        RestDataSource, AuthService]
```

```
})
export class ModelModule { }
```

Enabling Authentication

The next step is to wire up the component that obtains the credentials from the user so that it will perform authentication through the new service, as shown in Listing 9-11.

Listing 9-11. Enabling Authentication in the auth.component.ts File

```
import { Component } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Router } from "@angular/router";
import { AuthService } from "../model/auth.service";

@Component({
    moduleId: module.id,
    templateUrl: "auth.component.html"
})
export class AuthComponent {
    public username: string;
    public password: string;
    public errorMessage: string;

    constructor(private router: Router,
                private auth: AuthService) { }

    authenticate(form: NgForm) {
        if (form.valid) {
            this.auth.authenticate(this.username, this.password)
                .subscribe(response => {
                    if (response) {
                        this.router.navigateByUrl("/admin/main");
                    }
                    this.errorMessage = "Authentication Failed";
                })
        } else {
            this.errorMessage = "Form Data Invalid";
        }
    }
}
```

To prevent the application from navigating directly to the administration features, which will lead to HTTP requests being sent without a token, I added a file called `auth.guard.ts` in the `SportsStore/src/app/admin` folder and defined the route guard shown in Listing 9-12.

Listing 9-12. The Contents of the auth.guard.ts File in the SportsStore/src/app/admin Folder

```
import { Injectable } from "@angular/core";
import { ActivatedRouteSnapshot, RouterStateSnapshot,
            Router } from "@angular/router";
import { AuthService } from "../model/auth.service";

@Injectable()
export class AuthGuard {

    constructor(private router: Router,
                private auth: AuthService) { }

    canActivate(route: ActivatedRouteSnapshot,
        state: RouterStateSnapshot): boolean {

        if (!this.auth.authenticated) {
            this.router.navigateByUrl("/admin/auth");
            return false;
        }
        return true;
    }
}
```

Listing 9-13 applies the route guard to one of the routes defined by the administration feature module.

Listing 9-13. Guarding a Route in the admin.module.ts File

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { FormsModule } from "@angular/forms";
import { RouterModule } from "@angular/router";
import { AuthComponent } from "./auth.component";
import { AdminComponent } from "./admin.component";
import { AuthGuard } from "./auth.guard";

let routing = RouterModule.forChild([
    { path: "auth", component: AuthComponent },
    { path: "main", component: AdminComponent, canActivate: [AuthGuard] },
    { path: "**", redirectTo: "auth" }
]);

@NgModule({
    imports: [CommonModule, FormsModule, routing],
    providers: [AuthGuard],
    declarations: [AuthComponent, AdminComponent]
})
export class AdminModule {}
```

To test the authentication system, click the `Admin` button, enter some credentials, and click the Log In button. If the credentials are the ones from Table 9-1, then you will see the placeholder for the administration features. If you enter other credentials, you will see an error message. Both outcomes are illustrated in Figure 9-3.

Tip The token isn't stored persistently, so if you can, reload the application in the browser to start again and try a different set of credentials.

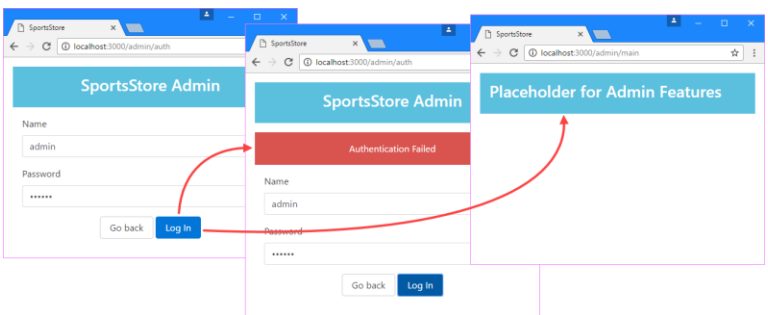

Figure 9-3. Testing the authentication feature

Extending the Data Source and Repositories

With the authentication system in place, the next step is to extend the data source so that it can send authenticated requests and to expose those features through the order and product repository classes. Listing 9-14 adds methods to the data source that include the authentication token.

Listing 9-14. Adding New Operations in the rest.datasource.ts File

```
import { Injectable } from "@angular/core";
import { Http, Request, RequestMethod } from "@angular/http";
import { Observable } from "rxjs/Observable";
import { Product } from "./product.model";
import { Cart } from "./cart.model";
import { Order } from "./order.model";
import "rxjs/add/operator/map";

const PROTOCOL = "http";
const PORT = 3500;
```

```
@Injectable()
export class RestDataSource {
    baseUrl: string;
    auth_token: string;

    constructor(private http: Http) {
        this.baseUrl = `${PROTOCOL}://${location.hostname}:${PORT}/`;
    }

    authenticate(user: string, pass: string): Observable<boolean> {
        return this.http.request(new Request({
            method: RequestMethod.Post,
            url: this.baseUrl + "login",
            body: { name: user, password: pass }
        })).map(response => {
            let r = response.json();
            this.auth_token = r.success ? r.token : null;
            return r.success;
        });
    }

    getProducts(): Observable<Product[]> {
        return this.sendRequest(RequestMethod.Get, "products");
    }

    saveProduct(product: Product): Observable<Product> {
        return this.sendRequest(RequestMethod.Post, "products",
            product, true);
    }

    updateProduct(product): Observable<Product> {
        return this.sendRequest(RequestMethod.Put,
            `products/${product.id}`, product, true);
    }

    deleteProduct(id: number): Observable<Product> {
        return this.sendRequest(RequestMethod.Delete,
            `products/${id}`, null, true);
    }

    getOrders(): Observable<Order[]> {
        return this.sendRequest(RequestMethod.Get,
            "orders", null, true);
    }

    deleteOrder(id: number): Observable<Order> {
        return this.sendRequest(RequestMethod.Delete,
            `orders/${id}`, null, true);
```

```
    }

    updateOrder(order: Order): Observable<Order> {
        return this.sendRequest(RequestMethod.Put,
            `orders/${order.id}`, order, true);
    }

    saveOrder(order: Order): Observable<Order> {
        return this.sendRequest(RequestMethod.Post,
            "orders", order);
    }

    private sendRequest(verb: RequestMethod,
            url: string, body?: Product | Order, auth: boolean = false)
                : Observable<Product | Product[] | Order | Order[]> {

        let request = new Request({
            method: verb,
            url: this.baseUrl + url,
            body: body
        });
        if (auth && this.auth_token != null) {
            request.headers.set("Authorization", `Bearer<${this.auth_token}>`);
        }
        return this.http.request(request).map(response => response.json());
    }
}
```

Listing 9-15 adds new methods to the product repository class that allow products to be created, updated, or deleted. The `saveProduct` method is responsible for creating and updating products, which is an approach that works well when using a single object managed by a component, which you will see demonstrated later in this chapter. The listing also changes the type of the constructor argument to `RestDataSource`.

Listing 9-15. Adding New Operations in the product.repository.ts File

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

@Injectable()
export class ProductRepository {
    private products: Product[] = [];
    private categories: string[] = [];

    constructor(private dataSource: RestDataSource) {
        dataSource.getProducts().subscribe(data => {
            this.products = data;
```

```
            this.categories = data.map(p => p.category)
                .filter((c, index, array) => array.indexOf(c) == index).sort();
        });
    }

    getProducts(category: string = null): Product[] {
        return this.products
            .filter(p => category == null || category == p.category);
    }

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

    getCategories(): string[] {
        return this.categories;
    }

    saveProduct(product: Product) {
        if (product.id == null || product.id == 0) {
            this.dataSource.saveProduct(product)
                .subscribe(p => this.products.push(p));
        } else {
            this.dataSource.updateProduct(product)
                .subscribe(p => {
                    this.products.splice(this.products.
                        findIndex(p => p.id == product.id), 1, product);
                });
        }
    }

    deleteProduct(id: number) {
        this.dataSource.deleteProduct(id).subscribe(p => {
            this.products.splice(this.products.
                findIndex(p => p.id == id), 1);
        })
    }
}
```

Listing 9-16 makes the corresponding changes to the order repository, adding methods that allow orders to be modified and deleted.

Listing 9-16. Adding New Operations in the order.repository.ts File

```
import { Injectable } from "@angular/core";
import { Observable } from "rxjs/Observable";
import { Order } from "./order.model";
import { RestDataSource } from "./rest.datasource";
```

```
@Injectable()
export class OrderRepository {
    private orders: Order[] = [];
    private loaded: boolean = false;

    constructor(private dataSource: RestDataSource) {}

    loadOrders() {
        this.loaded = true;
        this.dataSource.getOrders()
            .subscribe(orders => this.orders = orders);
    }

    getOrders(): Order[] {
        if (!this.loaded) {
            this.loadOrders();
        }
        return this.orders;
    }

    saveOrder(order: Order): Observable<Order> {
        return this.dataSource.saveOrder(order);
    }

    updateOrder(order: Order) {
        this.dataSource.updateOrder(order).subscribe(order => {
            this.orders.splice(this.orders.
                findIndex(o => o.id == order.id), 1, order);
        });
    }

    deleteOrder(id: number) {
        this.dataSource.deleteOrder(id).subscribe(order => {
            this.orders.splice(this.orders.findIndex(o => id == o.id));
        });
    }
}
```

The order repository defines a `loadOrders` method that gets the orders from the repository and that is used to ensure that the request isn't sent to the RESTful web service until authentication has been performed.

Creating the Administration Feature Structure

Now that the authentication system is in place and the repositories provide the full range of operations, I can create the structure that will display the administration features, which I am

going to create by building on the existing URL routing configuration. Table 9-2 lists the URLs that I am going to support and the functionality that each will present to the user.

Table 9-2. *The URLs for Administration Features*

Name	Description
`/admin/main/products`	Navigating to this URL will display all the products in a table, along with buttons that allow an existing product to be edited or deleted and a new product to be created.
`/admin/main/products/create`	Navigating to this URL will present the user with an empty editor for creating a new product.
`/admin/main/products/edit/1`	Navigating to this URL will present the user with a populated editor for editing an existing product.
`/admin/main/orders`	Navigating to this URL will present the user with all the orders in a table, along with buttons to mark an order shipped and to cancel an order by deleting it.

Creating the Placeholder Components

I find that the easiest way to add features to an Angular project is to define components that have placeholder content and build the structure of the application around them. Once the structure is in place, then I return to the components and implement the features in detail. For the administration features, I started by adding a file called **`productTable.component.ts`** to the **`SportsStore/src/app/admin`** folder and defined the component shown in Listing 9-17. This component will be responsible for showing a list of products, along with buttons required to edit and delete them or to create a new product.

Listing 9-17. The Contents of the productTable.component.ts File in the SportsStore/src/app/admin Folder

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

@Component({
    template: `<div class="bg-info p-a-1">
                 <h3>Product Table Placeholder</h3>
               </div>`
})
export class ProductTableComponent {}
```

I added a file called `productEditor.component.ts` in the `SportsStore/src/app/admin` folder and used it to define the component shown in Listing 9-18, which will be used to allow the user to enter the details required to create or edit a component.

Listing 9-18. The Contents of the productEditor.component.ts File in the SportsStore/src/app/admin Folder

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

@Component({
    template: `<div class="bg-warning p-a-1">
                 <h3>Product Editor Placeholder</h3>
               </div>`
})
export class ProductEditorComponent { }
```

To create the component that will be responsible for managing customer orders, I added a file called `orderTable.component.ts` to the `SportsStore/src/app/admin` folder and added the code shown in Listing 9-19.

Listing 9-19. The Contents of the orderTable.component.ts File in the SportsStore/src/app/admin Folder

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

@Component({
    template: `<div class="bg-primary p-a-1">
                 <h3>Order Table Placeholder</h3>
               </div>`
})
export class OrderTableComponent { }
```

Preparing the Common Content and the Feature Module

The components created in the previous section will be responsible for specific features. To bring those features together and allow the user to navigate between them, I need to modify the template of the placeholder component that I have been using to demonstrate the result of a successful authentication attempt. I replaced the placeholder content with the elements shown in Listing 9-20.

Listing 9-20. Replacing the Content in the admin.component.html File

```
<div class="navbar navbar-inverse bg-info">
    <a class="navbar-brand">SPORTS STORE Admin</a>
</div>
<div class="m-t-1">
    <div class="col-xs-3">
```

```
        <button class="btn btn-outline-info btn-block"
                routerLink="/admin/main/products"
                routerLinkActive="active">
            Products
        </button>
        <button class="btn btn-outline-info btn-block"
                routerLink="/admin/main/orders"
                routerLinkActive="active">
            Orders
        </button>
        <button class="btn btn-outline-danger btn-block" (click)="logout()">
            Logout
        </button>
    </div>
    <div class="col-xs-9">
        <router-outlet></router-outlet>
    </div>
</div>
```

This template contains a `router-outlet` element that will be used to display the components from the previous section. There are also buttons that will navigate the application to the `/admin/main/products` and `/admin/main/orders` URLs, which will select the products or orders features. These buttons use the `routerLinkActive` attribute, which is used to add the element to a CSS class when the route specified by the `routerLink` attribute is active.

The template also contains a `Logout` button that has an event binding that targets a method called `logout`. Listing 9-21 adds this method to the component, which uses the authentication service to remove the bearer token and navigates the application to the default URL.

Listing 9-21. Implementing the Logout Method in the admin.component.ts File

```
import { Component } from "@angular/core";
import { Router } from "@angular/router";
import { AuthService } from "../model/auth.service";

@Component({
    moduleId: module.id,
    templateUrl: "admin.component.html"
})
export class AdminComponent {

    constructor(private auth: AuthService,
                private router: Router) { }

    logout() {
```

```
            this.auth.clear();
            this.router.navigateByUrl("/");
        }
}
```

Listing 9-22 enables the placeholder components that will be used for each administration feature and extends the URL routing configuration to implement the URLs from Table 9-2.

Listing 9-22. Configuring the Feature Module in the admin.module.ts File

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { FormsModule } from "@angular/forms";
import { RouterModule } from "@angular/router";
import { AuthComponent } from "./auth.component";
import { AdminComponent } from "./admin.component";
import { AuthGuard } from "./auth.guard";
import { ProductTableComponent } from "./productTable.component";
import { ProductEditorComponent } from "./productEditor.component";
import { OrderTableComponent } from "./orderTable.component";

let routing = RouterModule.forChild([
    { path: "auth", component: AuthComponent },
    {
        path: "main", component: AdminComponent, canActivate: [AuthGuard],
        children: [
            { path: "products/:mode/:id", component: ProductEditorComponent },
            { path: "products/:mode", component: ProductEditorComponent },
            { path: "products", component: ProductTableComponent },
            { path: "orders", component: OrderTableComponent },
            { path: "**", redirectTo: "products" }
        ]
    },
    { path: "**", redirectTo: "auth" }
]);

@NgModule({
    imports: [CommonModule, FormsModule, routing],
    providers: [AuthGuard],
    declarations: [AuthComponent, AdminComponent,
        ProductTableComponent, ProductEditorComponent, OrderTableComponent]
})
export class AdminModule {}
```

Individual routes can be extended using the `children` property, which is used to define routes that will target a nested `router-outlet` element, which I describe in Chapter 25. As you will see shortly, components can get details of the active route from Angular so they can adapt their behavior. Routes can include route parameters, such as `:mode` or `:id`, that match any URL

segment and that can be used to provide information to components that can be used to change their behavior.

When all the changes have been saved, click the Admin button and authenticate as `admin` with the password `secret`. You will see the new layout, as shown in Figure 9-4. Clicking the Products and Orders buttons will change the component displayed by the `router-outlet` element from Listing 9-20, and clicking the Logout button will exit the administration area.

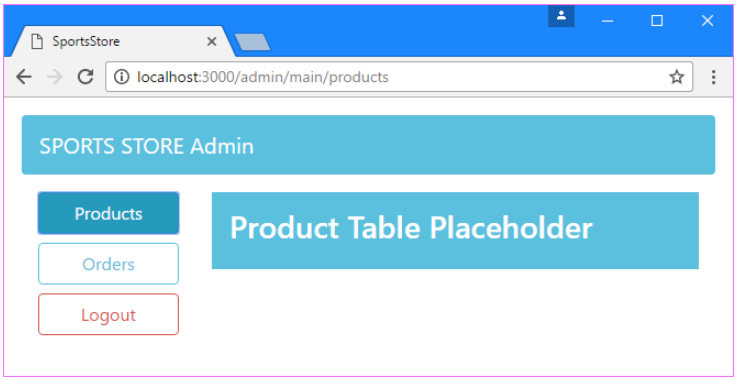

Figure 9-4. The administration layout structure

Implementing the Product Feature

The initial administration feature presented to the user will be a list of products, with the ability to create a new product and delete or edit an existing one. Listing 9-23 removes the placeholder content from the product table component and adds the logic required to implement this feature.

Listing 9-23. Replacing the Placeholder Content in the productTable.component.ts File

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

@Component({
    moduleId: module.id,
    templateUrl: "productTable.component.html"
})
export class ProductTableComponent {

    constructor(private repository: ProductRepository) { }
```

```
    getProducts(): Product[] {
        return this.repository.getProducts();
    }

    deleteProduct(id: number) {
        this.repository.deleteProduct(id);
    }
}
```

The component methods provide access to the products in the repository and allow products to be deleted. The other operations will be handled by the editor component, which will be activated using routing URLs in the component's template. To provide the template, I added a file called `productTable.component.html` in the `SportsStore/src/app/admin` folder and added the markup shown in Listing 9-24.

Listing 9-24. The productTable.component.html File in the SportsStore/src/app/admin Folder

```
<table class="table table-sm table-striped">
    <thead>
        <tr>
            <th>ID</th><th>Name</th><th>Category</th><th>Price</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        <tr *ngFor="let p of getProducts()">
            <td>{{p.id}}</td>
            <td>{{p.name}}</td>
            <td>{{p.category}}</td>
            <td>{{p.price | currency:"USD":true:"2.2-2"}}</td>
            <td>
                <button class="btn btn-sm btn-warning"
                        [routerLink]="['/admin/main/products/edit', p.id]">
                    Edit
                </button>
                <button class="btn btn-sm btn-danger" (click)="deleteProduct(p.id)">
                    Delete
                </button>
            </td>
        </tr>
    </tbody>
</table>
<button class="btn btn-primary" routerLink="/admin/main/products/create">
    Create New Product
</button>
```

The template contains a table that uses the `ngFor` directive to generate a row for each product returned by the component's `getProducts` method. Each row contains a Delete button

that invokes the component's `delete` method and an `Edit` button that navigates to a URL that targets the editor component. The editor component is also the target of the `Create New Product`, although a different URL is used.

Implementing the Product Editor

Components can receive information about the current routing URL and adapt their behavior accordingly. The editor component needs to use this feature to differentiate between requests to create a new component and edit an existing one. Listing 9-25 adds the functionality to the editor component required to create or edit products.

Listing 9-25. Adding Functionality in the productEditor.component.ts File

```
import { Component } from "@angular/core";
import { Router, ActivatedRoute } from "@angular/router";
import { NgForm } from "@angular/forms";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";

@Component({
    moduleId: module.id,
    templateUrl: "productEditor.component.html"
})
export class ProductEditorComponent {
  editing: boolean = false;
    product: Product = new Product();

    constructor(private repository: ProductRepository,
                private router: Router,
                activeRoute: ActivatedRoute) {

        this.editing = activeRoute.snapshot.params["mode"] == "edit";
        if (this.editing) {
            Object.assign(this.product,
                repository.getProduct(activeRoute.snapshot.params["id"]));
        }
    }

    save(form: NgForm) {
        this.repository.saveProduct(this.product);
        this.router.navigateByUrl("/admin/main/products");
    }
}
```

Angular will provide an `ActivatedRoute` object as a constructor argument when it creates a new instance of the component class and that can be used to inspect the activated route. In

this case, the component works out whether it should be editing or creating a product and, if editing, retrieves the current details from the repository. There is also a `save` method, which uses the repository to save changes that the user has made.

To provide the component with a template, I added a file called `productEditor.component.html` in the `SportsStore/src/app/admin` folder and added the markup shown in Listing 9-26.

Listing 9-26. The productEditor.component.html File in the SportsStore/src/app/admin Folder

```
<div class="bg-primary p-a-1" [class.bg-warning]="editing">
    <h5>{{editing  ? "Edit" : "Create"}} Product</h5>
</div>
<form novalidate #form="ngForm" (ngSubmit)="save(form)" >
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" name="name" [(ngModel)]="product.name" />
    </div>
    <div class="form-group">
        <label>Category</label>
        <input class="form-control" name="category" [(ngModel)]="product.category" />
    </div>
    <div class="form-group">
        <label>Description</label>
        <textarea class="form-control" name="description"
                  [(ngModel)]="product.description">
        </textarea>
    </div>
    <div class="form-group">
        <label>Price</label>
        <input class="form-control" name="price" [(ngModel)]="product.price" />
    </div>
    <button type="submit" class="btn btn-primary" [class.btn-warning]="editing">
        {{editing ? "Save" : "Create"}}
    </button>
    <button type="reset" class="btn btn-secondary" routerLink="/admin/main/products">
        Cancel
    </button>
</form>
```

The template contains a form with fields for the properties defined by the `Product` model class, with the exception of the `id` property, which is assigned automatically by the RESTful web service.

The elements in the form adapt their appearance to differentiate between the editing and creating features. To see how the component works, authenticate to access the Admin features and click the Create New Product button that appears under the table of products. Fill

out the form, click the Create button, and the new product will be sent to the RESTful web service where it will be assigned an ID property and displayed in the product table, as shown in Figure 9-5.

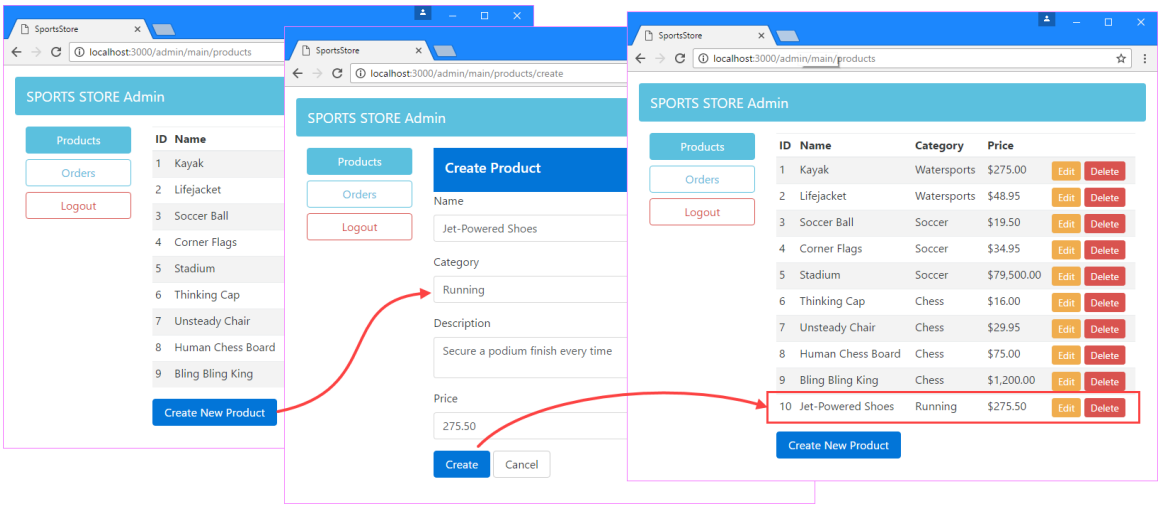

Figure 9-5. Creating a new product

The editing process works in a similar way. Click one of the Edit buttons to see the current details, edit them using the form fields, and click the Save button to save the changes, as shown in Figure 9-6.

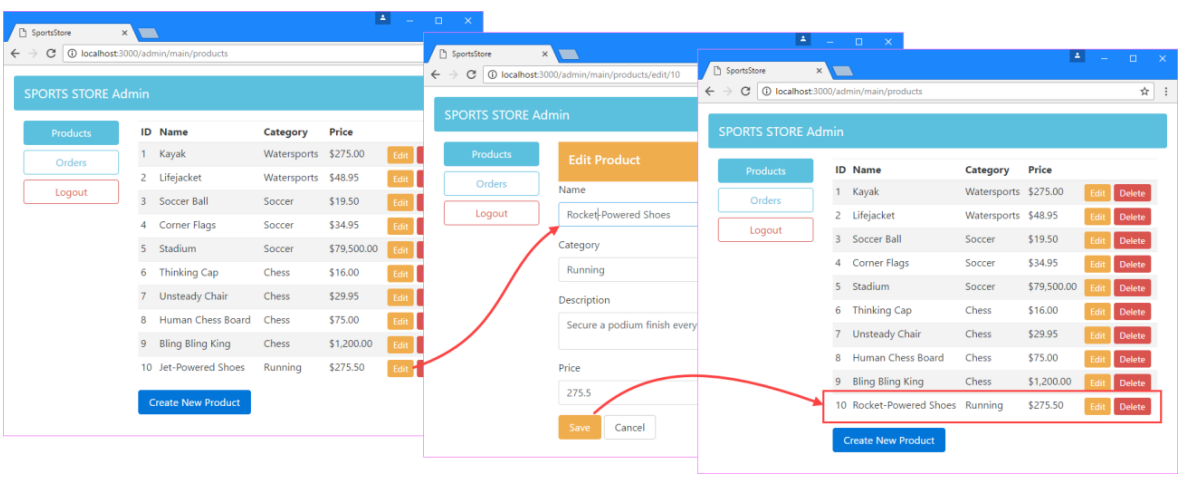

Figure 9-6. Editing an existing product

Implementing the Orders Feature

The order management feature is nice and simple. It requires a table that lists the set of orders, along with buttons that will set the shipped property to true or delete an order entirely. Listing 9-27 replaces the placeholder content in the component with the logic required to support these operations.

Listing 9-27. Adding Operations in the orderTable.component.ts File

```
import { Component } from "@angular/core";
import { Order } from "../model/order.model";
import { OrderRepository } from "../model/order.repository";

@Component({
    moduleId: module.id,
    templateUrl: "orderTable.component.html"
})
export class OrderTableComponent {
    includeShipped = false;

    constructor(private repository: OrderRepository) {}

    getOrders(): Order[] {
        return this.repository.getOrders()
            .filter(o => this.includeShipped || !o.shipped);
    }

    markShipped(order: Order) {
        order.shipped = true;
        this.repository.updateOrder(order);
    }

    delete(id: number) {
        this.repository.deleteOrder(id);
    }
}
```

In addition to providing methods for marking orders as shipped and deleting orders, the component defines a `getOrders` method that allows shipped orders to be included or excluded based on the value of a property called `includeShipped`. This property is used in the template, which I created by adding a file called `orderTable.component.html` to the `SportsStore/src/app/admin` folder with the markup shown in Listing 9-28.

Listing 9-28. The Contents of the orderTable.component.html in the SportsStore/src/app/admin Folder

```
<div class="form-check">
```

```
    <label class="form-check-label">
    <input type="checkbox" class="form-check-input" [(ngModel)]="includeShipped"/>
        Display Shipped Orders
    </label>
</div>
<table class="table table-sm">
    <thead>
        <tr><th>Name</th><th>Zip</th><th colspan="2">Cart</th><th></th></tr>
    </thead>
    <tbody>
        <tr *ngIf="getOrders().length == 0">
            <td colspan="5">There are no orders</td>
        </tr>
        <ng-template ngFor let-o [ngForOf]="getOrders()">
            <tr>
                <td>{{o.name}}</td><td>{{o.zip}}</td>
                <th>Product</th><th>Quantity</th>
                <td>
                    <button class="btn btn-warning" (click)="markShipped(o)">
                        Ship
                    </button>
                    <button class="btn btn-danger" (click)="delete(o.id)">
                        Delete
                    </button>
                </td>
            </tr>
            <tr *ngFor="let line of o.cart.lines">
                <td colspan="2"></td>
                <td>{{line.product.name}}</td>
                <td>{{line.quantity}}</td>
            </tr>
        </ng-template>
    </tbody>
</table>
```

Remember that the data presented by the RESTful web service is reset each time the process is started, which means you will have to use the shopping cart and checkout to create orders. Once that's done, you can inspect and manage them using the Orders section of the administration tool, as shown in Figure 9-7.

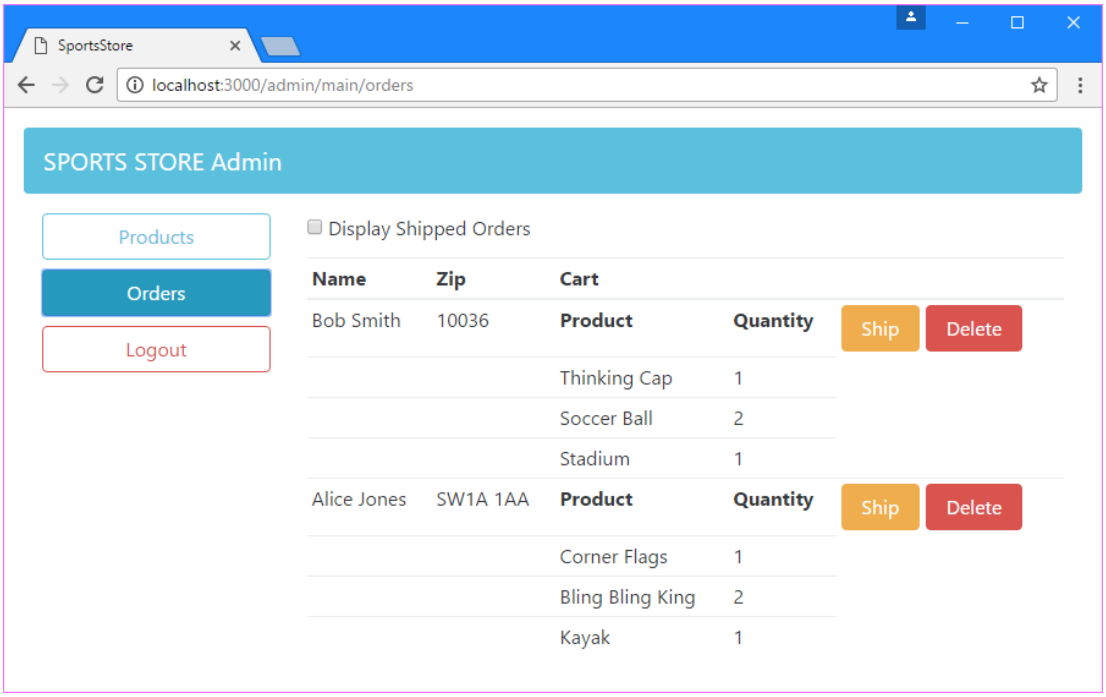

Figure 9-7. Managing orders

Summary

In this chapter, I created a dynamically loaded Angular feature module that contains the administration tools required to manage the catalog of products and process orders. In the next chapter, I prepare the SportsStore application for deployment into production.