

CHAPTER 8



SportsStore: Orders and Checkout

In this chapter, I continue adding features to the SportsStore application that I created in Chapter 7. I add support for a shopping cart and a checkout process and replace the dummy data with the data from the RESTful web service.

Preparing the Example Application

No preparation is required for this chapter, which continues using the SportsStore project from Chapter 7. 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** 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-6>.

Creating the Cart

The user needs a cart into which products can be placed and used to start the checkout process. In the sections that follow, I'll add a cart to the application and integrate it into the store so that the user can select the products they want.

Creating the Cart Model

The starting point for the cart feature is a new model class that will be used to gather together the products that the user has selected. I added a file called `cart.model.ts` in the `src/app/model` folder and used it to define the class shown in Listing 8-1.

Listing 8-1. The Contents of the `cart.model.ts` File in the `src/app/model` Folder

```

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

@Injectable()
export class Cart {
  public lines: CartLine[] = [];
  public itemCount: number = 0;
  public cartPrice: number = 0;

  addLine(product: Product, quantity: number = 1) {
    let line = this.lines.find(line => line.product.id == product.id);
    if (line != undefined) {
      line.quantity += quantity;
    } else {
      this.lines.push(new CartLine(product, quantity));
    }
    this.recalculate();
  }

  updateQuantity(product: Product, quantity: number) {
    let line = this.lines.find(line => line.product.id == product.id);
    if (line != undefined) {
      line.quantity = Number(quantity);
    }
    this.recalculate();
  }

  removeLine(id: number) {
    let index = this.lines.findIndex(line => line.product.id == id);
    this.lines.splice(index, 1);
    this.recalculate();
  }

  clear() {
    this.lines = [];
    this.itemCount = 0;
    this.cartPrice = 0;
  }

  private recalculate() {
    this.itemCount = 0;
    this.cartPrice = 0;
    this.lines.forEach(l => {
      this.itemCount += l.quantity;
      this.cartPrice += (l.quantity * l.product.price);
    })
  }
}

```

```
export class CartLine {

    constructor(public product: Product,
               public quantity: number) {}

    get lineTotal() {
        return this.quantity * this.product.price;
    }
}
```

Individual product selections are represented as an array of `CartLine` objects, each of which contains a `Product` object and a quantity. The `Cart` class keeps track of the total number of items that have been selected and their total cost.

There should be a single `Cart` object used throughout the entire application, ensuring that any part of the application can access the user's product selections. To achieve this, I am going to make the `Cart` a service, which means that Angular will take responsibility for creating an instance of the `Cart` class and will use it when it needs to create a component that has a `Cart` constructor argument. This is another use of the Angular dependency injection feature, which can be used to share objects throughout an application and which is described in detail in Chapters 19 and 20. The `@Injectable` decorator, which has been applied to the `Cart` class in the listing, indicates that this class will be used as a service.

■ **Note** 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.

Listing 8-2 registers the `Cart` class as a service in the `providers` property of the `model` feature module class.

Listing 8-2. Registering the `Cart` as a Service in the `model.module.ts` File in the `src/app/model` Folder

```
import { NgModule } from "@angular/core";
import { ProductRepository } from "../product.repository";
import { StaticDataSource } from "../static.datasource";
import { Cart } from "../cart.model";

@NgModule({
    providers: [ProductRepository, StaticDataSource, Cart]
})
export class ModelModule { }
```

Creating the Cart Summary Components

Components are the essential building blocks for Angular applications because they allow discrete units of code and content to be easily created. The `SportsStore` application will show users a summary of their product selections in the title area of the page, which I am going to implement by creating a component. I added a file called `cartSummary.component.ts` in the `src/app/store` folder and used it to define the component shown in Listing 8-3.

Listing 8-3. The Contents of the `cartSummary.component.ts` File in the `src/app/store` Folder

```
import { Component } from "@angular/core";
import { Cart } from "../model/cart.model";

@Component({
  selector: "cart-summary",
  templateUrl: "cartSummary.component.html"
})
export class CartSummaryComponent {

  constructor(public cart: Cart) { }
```

When Angular needs to create an instance of this component, it will have to provide a `Cart` object as a constructor argument, using the service that I configured in the previous section by adding the `Cart` class to the feature module's `providers` property. The default behavior for services means that a single `Cart` object will be created and shared throughout the application, although there are different service behaviors available (as described in Chapter 20).

To provide the component with a template, I created an HTML file called `cartSummary.component.html` in the same folder as the component class file and added the markup shown in Listing 8-4.

Listing 8-4. The Contents of the `cartSummary.component.html` File in the `src/app/store` Folder

```
<div class="float-right">
  <small>
    Your cart:
    <span *ngIf="cart.itemCount > 0">
      {{ cart.itemCount }} item(s)
      {{ cart.cartPrice | currency:"USD":"symbol":"2.2-2" }}
    </span>
    <span *ngIf="cart.itemCount == 0">
      (empty)
    </span>
  </small>
  <button class="btn btn-sm bg-dark text-white"
    [disabled]="cart.itemCount == 0">
    <i class="fa fa-shopping-cart"></i>
  </button>
</div>
```

This template uses the `Cart` object provided by its component to display the number of items in the cart and the total cost. There is also a button that will start the checkout process when I add it to the application later in the chapter.

■ **Tip** The button element in Listing 8-4 is styled using classes defined by Font Awesome, which is one of the packages in the `package.json` file from Chapter 7. This open source package provides excellent support for icons in web applications, including the shopping cart I need for the SportsStore application. See <http://fontawesome.io> for details.

Listing 8-5 registers the new component with the store feature module, in preparation for using it in the next section.

Listing 8-5. Registering the Component in the store.module.ts File in the src/app/store Folder

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { StoreComponent } from "../store.component";
import { CounterDirective } from "../counter.directive";
import { CartSummaryComponent } from "../cartSummary.component";

@NgModule({
  imports: [ModelModule, BrowserModule, FormsModule],
  declarations: [StoreComponent, CounterDirective, CartSummaryComponent],
  exports: [StoreComponent]
})
export class StoreModule { }
```

Integrating the Cart into the Store

The store component is the key to integrating the cart and the cart widget into the application. Listing 8-6 updates the store component so that its constructor has a Cart parameter and defines a method that will add a product to the cart.

Listing 8-6. Adding Cart Support in the store.component.ts File in the src/app/store Folder

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

@Component({
  selector: "store",
  templateUrl: "store.component.html"
})
export class StoreComponent {
  public selectedCategory = null;
  public productsPerPage = 4;
  public selectedPage = 1;

  constructor(private repository: ProductRepository,
    private cart: Cart) { }

  get products(): Product[] {
    let pageIndex = (this.selectedPage - 1) * this.productsPerPage
    return this.repository.getProducts(this.selectedCategory)
      .slice(pageIndex, pageIndex + this.productsPerPage);
  }
}
```

```

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

    changeCategory(newCategory?: string) {
        this.selectedCategory = newCategory;
    }

    changePage(newPage: number) {
        this.selectedPage = newPage;
    }

    changePageSize(newSize: number) {
        this.productsPerPage = Number(newSize);
        this.changePage(1);
    }

    get pageCount(): number {
        return Math.ceil(this.repository
            .getProducts(this.selectedCategory).length / this.productsPerPage)
    }

    addProductToCart(product: Product) {
        this.cart.addLine(product);
    }
}

```

To complete the integration of the cart into the store component, Listing 8-7 adds the element that will apply the cart summary component to the store component's template and adds a button to each product description with the event binding that calls the `addProductToCart` method.

Listing 8-7. Applying the Component in the `store.component.html` File in the `src/app/store` Folder

```

<div class="container-fluid">
    <div class="row">
        <div class="col bg-dark text-white">
            <a class="navbar-brand">SPORTS STORE</a>
            <cart-summary></cart-summary>
        </div>
    </div>
    <div class="row">

        <div class="col-3 p-2">
            <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>
        </div>
    </div>

```

```

<div class="col-9 p-2">
  <div *ngFor="let product of products" class="card m-1 p-1 bg-light">
    <h4>
      {{product.name}}
      <span class="badge badge-pill badge-primary float-right">
        {{ product.price | currency:"USD":"symbol":"2.2-2" }}
      </span>
    </h4>
    <div class="card-text bg-white p-1">
      {{product.description}}
      <button class="btn btn-success btn-sm float-right"
        (click)="addProductToCart(product)">
        Add To Cart
      </button>
    </div>
  </div>

  <div class="form-inline float-left mr-1">
    <select class="form-control" [value]="productsPerPage"
      (change)="changePageSize($event.target.value)">
      <option value="3">3 per Page</option>
      <option value="4">4 per Page</option>
      <option value="6">6 per Page</option>
      <option value="8">8 per Page</option>
    </select>
  </div>

  <div class="btn-group float-right">
    <button *counter="let page of pageCount" (click)="changePage(page)"
      class="btn btn-outline-primary" [class.active]="page == selectedPage">
      {{page}}
    </button>
  </div>
</div>
</div>
</div>

```

The result is a button for each product that adds it to the cart, as shown in Figure 8-1. The full cart process isn't complete yet, but you can see the effect of each addition in the cart summary at the top of the page.

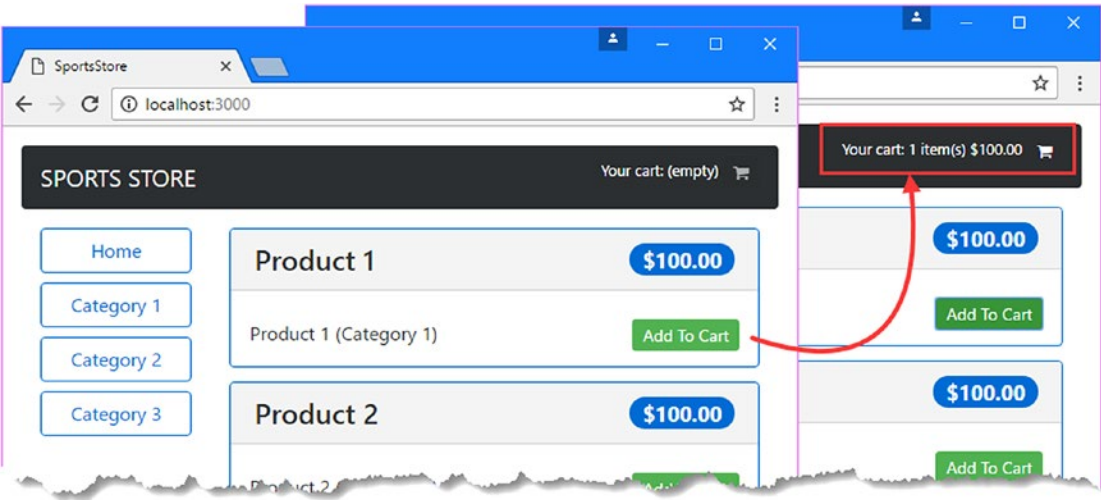


Figure 8-1. Adding cart support to the SportsStore application

Notice how clicking one of the Add To Cart buttons updates the summary component’s content automatically. This happens because there is a single Cart object being shared between two components and changes made by one component are reflected when Angular evaluates the data binding expressions in the other component.

Adding URL Routing

Most applications need to show different content to the user at different times. In the case of the SportsStore application, when the user clicks one of the Add To Cart buttons, they should be shown a detailed view of their selected products and given the chance to start the checkout process.

Angular supports a feature called *URL routing*, which uses the current URL displayed by the browser to select the components that are displayed to the user. This is an approach that makes it easy to create applications whose components are loosely coupled and easy to change without needing corresponding modifications elsewhere in the applications. URL routing also makes it easy to change the path that a user follows through an application.

For the SportsStore application, I am going to add support for three different URLs, which are described in Table 8-1. This is a simple configuration, but the routing system has a lot of features, which are described in detail in Chapters 25 to 27.

Table 8-1. The URLs Supported by the SportsStore Application

URL	Description
/store	This URL will display the list of products.
/cart	This URL will display the user’s cart in detail.
/checkout	This URL will display the checkout process.

In the sections that follow, I create placeholder components for the SportsStore cart and order checkout stages and then integrate them into the application using URL routing. Once the URLs are implemented, I will return to the components and add more useful features.

Creating the Cart Detail and Checkout Components

Before adding URL routing to the application, I need to create the components that will be displayed by the /cart and /checkout URLs. I only need some basic placeholder content to get started, just to make it obvious which component is being displayed. I started by adding a file called `cartDetail.component.ts` in the `src/app/store` folder and defined the component shown in Listing 8-8.

Listing 8-8. The Contents of the `cartDetail.component.ts` File in the `src/app/store` Folder

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

@Component({
  template: `<div><h3 class="bg-info p-1 text-white">Cart Detail Component</h3></div>`
})
export class CartDetailComponent {}
```

Next, I added a file called `checkout.component.ts` in the `src/app/store` folder and defined the component shown in Listing 8-9.

Listing 8-9. The Contents of the `checkout.component.ts` File in the `src/app/store` Folder

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

@Component({
  template: `<div><h3 class="bg-info p-1 text-white">Checkout Component</h3></div>`
})
export class CheckoutComponent { }
```

This component follows the same pattern as the cart component and displays a placeholder message. Listing 8-10 registers the components in the store feature module and adds them to the `exports` property, which means they can be used elsewhere in the application.

Listing 8-10. Registering Components in the `store.module.ts` File in the `src/app/store` Folder

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { StoreComponent } from "./store.component";
import { CounterDirective } from "./counter.directive";
import { CartSummaryComponent } from "./cartSummary.component";
import { CartDetailComponent } from "./cartDetail.component";
import { CheckoutComponent } from "./checkout.component";

@NgModule({
  imports: [ModelModule, BrowserModule, FormsModule],
  declarations: [StoreComponent, CounterDirective, CartSummaryComponent,
    CartDetailComponent, CheckoutComponent],
```

```

    exports: [StoreComponent, CartDetailComponent, CheckoutComponent]
  })
  export class StoreModule { }

```

Creating and Applying the Routing Configuration

Now that I have a range of components to display, the next step is to create the routing configuration that tells Angular how to map URLs into components. Each mapping of a URL to a component is known as a *URL route* or just a *route*. In Part 3, where I create more complex routing configurations, I define the routes in a separate file, but for this project, I am going to follow a simpler approach and define the routes within the `@NgModule` decorator of the application's root module, as shown in Listing 8-11.

■ **Tip** The Angular routing feature requires a base element in the HTML document, which provides the base URL against which routes are applied. This element was added to the `index.html` file by the `ng new` command when I created the SportsStore project in Chapter 7. If you omit the element, Angular will report an error and be unable to apply the routes.

Listing 8-11. Creating the Routing Configuration 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 { 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";

@NgModule({
  imports: [BrowserModule, StoreModule,
    RouterModule.forRoot([
      { path: "store", component: StoreComponent },
      { path: "cart", component: CartDetailComponent },
      { path: "checkout", component: CheckoutComponent },
      { path: "**", redirectTo: "/store" }
    ])],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

The `RouterModule.forRoot` method is passed a set of routes, each of which maps a URL to a component. The first three routes in the listing match the URLs from Table 8-1. The final route is a wildcard that redirects any other URL to `/store`, which will display `StoreComponent`.

When the routing feature is used, Angular looks for the `router-outlet` element, which defines the location in which the component that corresponds to the current URL should be displayed. Listing 8-12 replaces the `store` element in the root component's template with the `router-outlet` element.

Listing 8-12. Defining the Routing Target in the app.component.ts File in the src/app Folder

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

@Component({
  selector: "app",
  template: "<router-outlet></router-outlet>"
})
export class AppComponent { }
```

Angular will apply the routing configuration when you save the changes and the browser reloads the HTML document. The content displayed in the browser window hasn't changed, but if you examine the browser's URL bar, you will be able to see that the routing configuration has been applied, as shown in Figure 8-2.

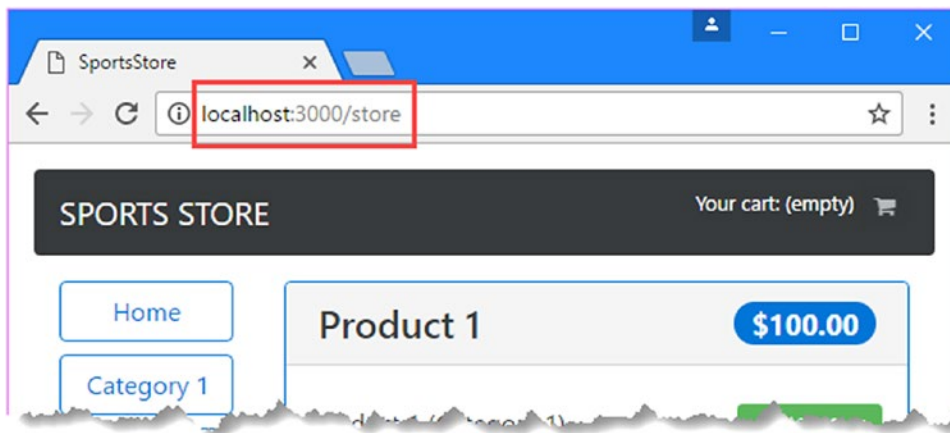


Figure 8-2. The effect of URL routing

Navigating Through the Application

With the routing configuration in place, it is time to add support for navigating between components by changing the browser's URL. The URL routing feature relies on a JavaScript API provided by the browser, which means the user can't simply type the target URL into the browser's URL bar. Instead, the navigation has to be performed by the application, either by using JavaScript code in a component or other building block or by adding attributes to HTML elements in the template.

When the user clicks one of the Add To Cart buttons, the cart detail component should be shown, which means that the application should navigate to the /cart URL. Listing 8-13 adds navigation to the component method that is invoked when the user clicks the button.

Listing 8-13. Navigating Using JavaScript in the store.component.ts File in the app/src/store Folder

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

```

@Component({
  selector: "store",
  templateUrl: "store.component.html"
})
export class StoreComponent {
  public selectedCategory = null;
  public productsPerPage = 4;
  public selectedPage = 1;

  constructor(private repository: ProductRepository,
    private cart: Cart,
    private router: Router) { }

  get products(): Product[] {
    let pageIndex = (this.selectedPage - 1) * this.productsPerPage
    return this.repository.getProducts(this.selectedCategory)
      .slice(pageIndex, pageIndex + this.productsPerPage);
  }

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

  changeCategory(newCategory?: string) {
    this.selectedCategory = newCategory;
  }

  changePage(newPage: number) {
    this.selectedPage = newPage;
  }

  changePageSize(newSize: number) {
    this.productsPerPage = Number(newSize);
    this.changePage(1);
  }

  get pageCount(): number {
    return Math.ceil(this.repository
      .getProducts(this.selectedCategory).length / this.productsPerPage)
  }

  addProductToCart(product: Product) {
    this.cart.addLine(product);
    this.router.navigateByUrl("/cart");
  }
}

```

The constructor has a Router parameter, which is provided by Angular through the dependency injection feature when a new instance of the component is created. In the `addProductToCart` method, the `Router.navigateByUrl` method is used to navigate to the `/cart` URL.

Navigation can also be done by adding the `routerLink` attribute to elements in the template. In Listing 8-14, the `routerLink` attribute has been applied to the cart button in the cart summary component's template.

Listing 8-14. Adding a Navigation in the `cartSummary.component.html` File in the `src/app/store` Folder

```
<div class="float-right">
  <small>
    Your cart:
    <span *ngIf="cart.itemCount > 0">
      {{ cart.itemCount }} item(s)
      {{ cart.cartPrice | currency:"USD":"symbol":"2.2-2" }}
    </span>
    <span *ngIf="cart.itemCount == 0">
      (empty)
    </span>
  </small>
  <button class="btn btn-sm bg-dark text-white"
    [disabled]="cart.itemCount == 0" routerLink="/cart">
    <i class="fa fa-shopping-cart"></i>
  </button>
</div>
```

The value specified by the `routerLink` attribute is the URL that the application will navigate to when the button is clicked. This particular button is disabled when the cart is empty, so it will perform the navigation only when the user has added a product to the cart.

To add support for the `routerLink` attribute, the `RouterModule` module must be imported into the feature module, as shown in Listing 8-15.

Listing 8-15. Importing the Router Module in the `store.module.ts` File in the `src/app/store` Folder

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { StoreComponent } from "./store.component";
import { CounterDirective } from "./counter.directive";
import { CartSummaryComponent } from "./cartSummary.component";
import { CartDetailComponent } from "./cartDetail.component";
import { CheckoutComponent } from "./checkout.component";
import { RouterModule } from "@angular/router";

@NgModule({
  imports: [ModelModule, BrowserModule, FormsModule, RouterModule],
  declarations: [StoreComponent, CounterDirective, CartSummaryComponent,
    CartDetailComponent, CheckoutComponent],
  exports: [StoreComponent, CartDetailComponent, CheckoutComponent]
})
export class StoreModule { }
```

To see the effect of the navigation, save the changes of the files and, once the browser has reloaded the HTML document, click one of the Add To Cart buttons. The browser will navigate to the `/cart` URL, as shown in Figure 8-3.

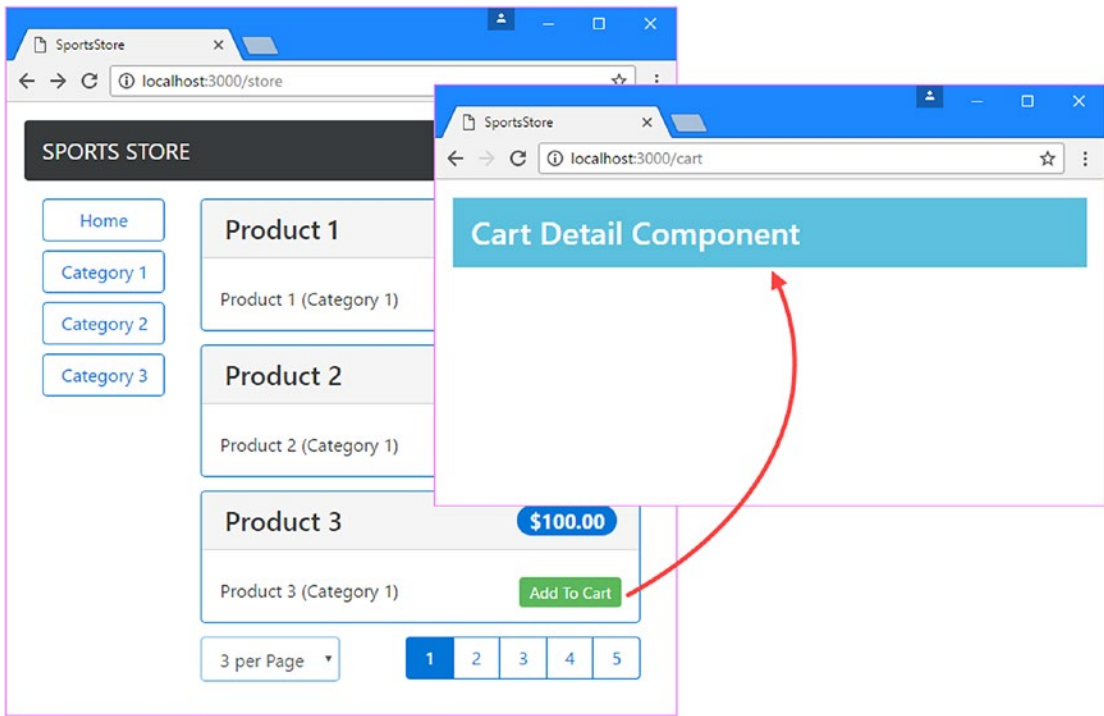


Figure 8-3. Using URL routing

Guarding the Routes

Remember that navigation can be performed only by the application. If you change the URL directly in the browser's URL bar, the browser will request the URL you enter from the web server. The Angular development server that is responding to HTTP requests will respond to any URL that doesn't correspond to a file by returning the contents of `index.html`. This is generally a useful behavior because it means you won't receive an HTTP error when the browser reload button is clicked. But it can cause problems if the application expects the user to navigate through the application following a specific path.

As an example, if you click one of the Add To Cart buttons and then click the browser's reload button, the HTTP server will return the contents of the `index.html` file, and Angular will immediately jump to the cart detail component, skipping over the part of the application that allows the user to select products.

For some applications, being able to start using different URLs makes sense, but if that's not the case, then Angular supports *route guards*, which are used to govern the routing system.

To prevent the application from starting with the `/cart` or `/order` URL, I added a file called `storeFirst.guard.ts` in the `SportsStore/src/app` folder and defined the class shown in Listing 8-16.

Listing 8-16. The Contents of the `storeFirst.guard.ts` File in the `src/app` Folder

```
import { Injectable } from "@angular/core";
import {
  ActivatedRouteSnapshot, RouterStateSnapshot,
  Router
} from "@angular/router";
import { StoreComponent } from "../store/store.component";
```

```

@Injectables()
export class StoreFirstGuard {
  private firstNavigation = true;

  constructor(private router: Router) { }

  canActivate(route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): boolean {
    if (this.firstNavigation) {
      this.firstNavigation = false;
      if (route.component !== StoreComponent) {
        this.router.navigateByUrl("/");
        return false;
      }
    }
    return true;
  }
}

```

There are different ways to guard routes, as described in Chapter 27, and this is an example of a guard that prevents a route from being activated, which is implemented as a class that defines a `canActivate` method. The implementation of this method uses the context objects that Angular provides that describe the route that is about to be navigated to and checks to see whether the target component is a `StoreComponent`. If this is the first time that the `canActivate` method has been called and a different component is about to be used, then the `Router.navigateByUrl` method is used to navigate to the root URL.

The `@Injectable` decorator has been applied in the listing because route guards are services. Listing 8-17 registers the guard as a service using the root module's `providers` property and guards each route using the `canActivate` property.

Listing 8-17. Guarding Routes 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 { 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: "**", redirectTo: "/store" }
  ]],
  providers: [StoreFirstGuard],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

If you reload the browser after clicking one of the Add To Cart buttons now, then you will see the browser is automatically directed back to safety, as shown in Figure 8-4.

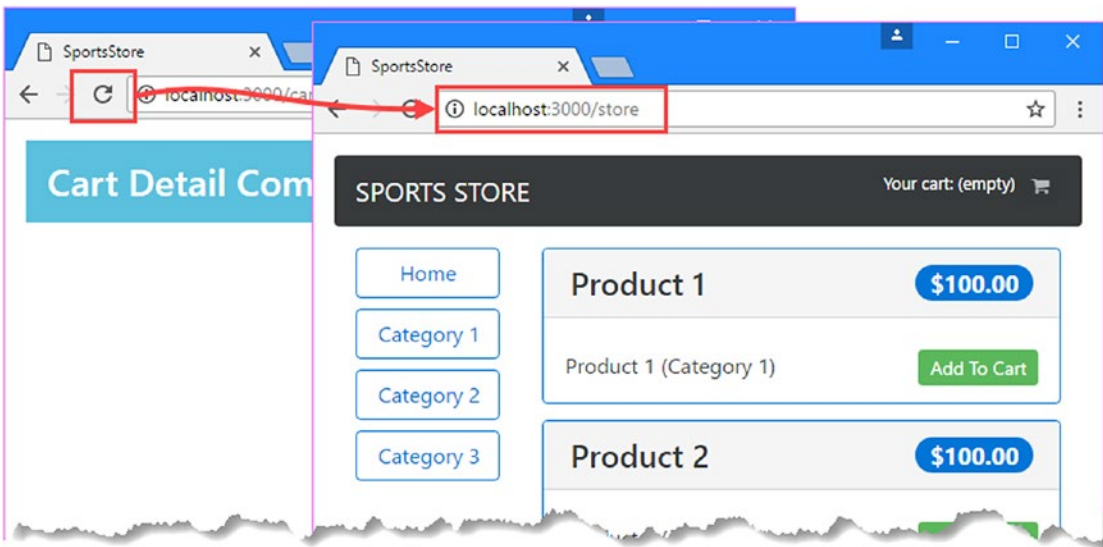


Figure 8-4. Guarding routes

Completing the Cart Detail Feature

Now that the application has navigation support, it is time to complete the view that details the contents of the user's cart. Listing 8-18 removes the inline template from the cart detail component, specifies an external template in the same directory, and adds a `Cart` parameter to the constructor, which will be accessible in the template through a property called `cart`.

Listing 8-18. Changing the Template in the `cartDetail.component.ts` File in the `src/app/store` Folder

```

import { Component } from "@angular/core";
import { Cart } from "../../model/cart.model";

@Component({

```



```

    templateUrl: "cartDetail.component.html"
  })
  export class CartDetailComponent {

    constructor(public cart: Cart) { }
  }

```

To complete the cart detail feature, I created an HTML file called `cartDetail.component.html` in the `src/app/store` folder and added the content shown in Listing 8-19.

Listing 8-19. The Contents of the `cartDetail.component.html` File in the `src/app/store` Folder

```

<div class="container-fluid">
  <div class="row">
    <div class="col bg-dark text-white">
      <a class="navbar-brand">SPORTS STORE</a>
    </div>
  </div>
  <div class="row">
    <div class="col mt-2">
      <h2 class="text-center">Your Cart</h2>
      <table class="table table-bordered table-striped p-2">
        <thead>
          <tr>
            <th>Quantity</th>
            <th>Product</th>
            <th class="text-right">Price</th>
            <th class="text-right">Subtotal</th>
          </tr>
        </thead>
        <tbody>
          <tr *ngIf="cart.lines.length == 0">
            <td colspan="4" class="text-center">
              Your cart is empty
            </td>
          </tr>
          <tr *ngFor="let line of cart.lines">
            <td>
              <input type="number" class="form-control-sm"
                style="width:5em"
                [value]="line.quantity"
                (change)="cart.updateQuantity(line.product,
                  $event.target.value)" />
            </td>
            <td>{{line.product.name}}</td>
            <td class="text-right">
              {{line.product.price | currency:"USD":true:"2.2-2"}}
            </td>
            <td class="text-right">
              {{(line.lineTotal) | currency:"USD":true:"2.2-2" }}
            </td>
            <td class="text-center">

```

```

        <button class="btn btn-sm btn-danger"
            (click)="cart.removeLine(line.product.id)">
            Remove
        </button>
    </td>
</tr>
</tbody>
<tfoot>
    <tr>
        <td colspan="3" class="text-right">Total:</td>
        <td class="text-right">
            {{cart.cartPrice | currency:"USD": "symbol":"2.2-2"}}
        </td>
    </tr>
</tfoot>
</table>
</div>
</div>
<div class="row">
    <div class="col">
        <div class="text-center">
            <button class="btn btn-primary m-1" routerLink="/store">
                Continue Shopping
            </button>
            <button class="btn btn-secondary m-1" routerLink="/checkout"
                [disabled]="cart.lines.length == 0">
                Checkout
            </button>
        </div>
    </div>
</div>
</div>

```

This template displays a table showing the user's product selections. For each product, there is an input element that can be used to change the quantity and a Remove button that deletes it from the cart. There are also two navigation buttons that allow the user to return to the list of products or continue to the checkout process, as shown in Figure 8-5. The combination of the Angular data bindings and the shared Cart object means that any changes made to the cart take immediate effect, recalculating the prices; and if you click the Continue Shopping button, the changes are reflected in the cart summary component shown above the list of products.

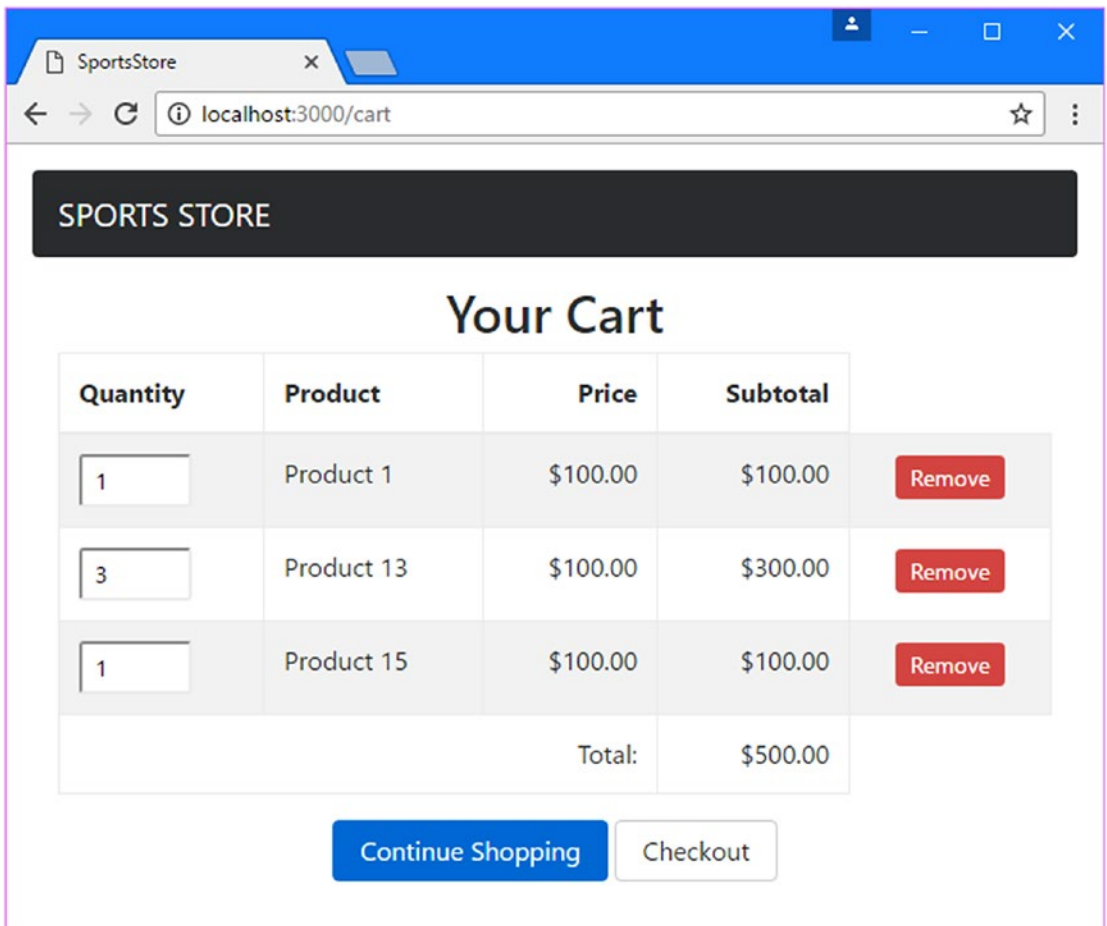


Figure 8-5. Completing the cart detail feature

Processing Orders

Being able to receive orders from customers is the most important aspect of an online store. In the sections that follow, I build on the application to add support for receiving the final details from the user and checking them out. To keep the process simple, I am going to avoid dealing with payment and fulfillment platforms, which are generally back-end services that are not specific to Angular applications.

Extending the Model

To describe orders placed by users, I added a file called `order.model.ts` in the `src/app/model` folder and defined the code shown in Listing 8-20.

Listing 8-20. The Contents of the order.model.ts File in the src/app/model Folder

```
import { Injectable } from "@angular/core";
import { Cart } from "../cart.model";

@Injectable()
export class Order {
  public id: number;
  public name: string;
  public address: string;
  public city: string;
  public state: string;
  public zip: string;
  public country: string;
  public shipped: boolean = false;

  constructor(public cart: Cart) { }

  clear() {
    this.id = null;
    this.name = this.address = this.city = null;
    this.state = this.zip = this.country = null;
    this.shipped = false;
    this.cart.clear();
  }
}
```

The Order class will be another service, which means there will be one instance shared throughout the application. When Angular creates the Order object, it will detect the Cart constructor parameter and provide the same Cart object that is used elsewhere in the application.

Updating the Repository and Data Source

To handle orders in the application, I need to extend the repository and the data source so they can receive Order objects. Listing 8-21 adds a method to the data source that receives an order. Since this is still the dummy data source, the method simply produces a JSON string from the order and writes it to the JavaScript console. I'll do something more useful with the objects in the next section when I create a data source that uses HTTP requests to communicate with the RESTful web service.

Listing 8-21. Handling Orders in the static.datasource.ts File in the src/app/model Folder

```
import { Injectable } from "@angular/core";
import { Product } from "../product.model";
import { Observable, from } from "rxjs";
import { Order } from "../order.model";

@Injectable()
export class StaticDataSource {
  private products: Product[] = [
    new Product(1, "Product 1", "Category 1", "Product 1 (Category 1)", 100),
    new Product(2, "Product 2", "Category 1", "Product 2 (Category 1)", 100),
    new Product(3, "Product 3", "Category 1", "Product 3 (Category 1)", 100),
  ]
}
```

```

    new Product(4, "Product 4", "Category 1", "Product 4 (Category 1)", 100),
    new Product(5, "Product 5", "Category 1", "Product 5 (Category 1)", 100),
    new Product(6, "Product 6", "Category 2", "Product 6 (Category 2)", 100),
    new Product(7, "Product 7", "Category 2", "Product 7 (Category 2)", 100),
    new Product(8, "Product 8", "Category 2", "Product 8 (Category 2)", 100),
    new Product(9, "Product 9", "Category 2", "Product 9 (Category 2)", 100),
    new Product(10, "Product 10", "Category 2", "Product 10 (Category 2)", 100),
    new Product(11, "Product 11", "Category 3", "Product 11 (Category 3)", 100),
    new Product(12, "Product 12", "Category 3", "Product 12 (Category 3)", 100),
    new Product(13, "Product 13", "Category 3", "Product 13 (Category 3)", 100),
    new Product(14, "Product 14", "Category 3", "Product 14 (Category 3)", 100),
    new Product(15, "Product 15", "Category 3", "Product 15 (Category 3)", 100),
  ];

  getProducts(): Observable<Product[]> {
    return from([this.products]);
  }

  saveOrder(order: Order): Observable<Order> {
    console.log(JSON.stringify(order));
    return from([order]);
  }
}

```

To manage orders, I added a file called `order.repository.ts` to the `src/app/model` folder and used it to define the class shown in Listing 8-22. There is only one method in the order repository at the moment, but I will add more functionality in Chapter 9 when I create the administration features.

■ **Tip** You don't have to use different repositories for each model type in the application, but I often do so because a single class responsible for multiple model types can become complex and difficult to maintain.

Listing 8-22. The Contents of the `order.repository.ts` File in the `src/app/model` Folder

```

import { Injectable } from "@angular/core";
import { Observable } from "rxjs";
import { Order } from "../order.model";
import { StaticDataSource } from "../static.datasource";

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

  constructor(private dataSource: StaticDataSource) {}

  getOrders(): Order[] {
    return this.orders;
  }
}

```

```

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

```

Updating the Feature Module

Listing 8-23 registers the `Order` class and the new repository as services using the `providers` property of the `model` feature module.

Listing 8-23. Registering Services in the `model.module.ts` File in the `src/app/model` Folder

```

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";

@NgModule({
    providers: [ProductRepository, StaticDataSource, Cart,
                Order, OrderRepository]
})
export class ModelModule { }

```

Collecting the Order Details

The next step is to gather the details from the user required to complete the order. Angular includes built-in directives for working with HTML forms and validating their contents. Listing 8-24 prepares the checkout component, switching to an external template, receiving the `Order` object as a constructor parameter, and providing some additional support to help the template.

Listing 8-24. Preparing for a Form in the `checkout.component.ts` File in the `src/app/store` Folder

```

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

@Component({
    templateUrl: "checkout.component.html",
    styleUrls: ["checkout.component.css"]
})
export class CheckoutComponent {
    orderSent: boolean = false;
    submitted: boolean = false;

    constructor(public repository: OrderRepository,
                public order: Order) {}

    submitOrder(form: NgForm) {

```

```

    this.submitted = true;
    if (form.valid) {
        this.repository.saveOrder(this.order).subscribe(order => {
            this.order.clear();
            this.orderSent = true;
            this.submitted = false;
        });
    }
}
}

```

The `submitOrder` method will be invoked when the user submits a form, which is represented by an `NgForm` object. If the data that the form contains is valid, then the `Order` object will be passed to the repository's `saveOrder` method, and the data in the cart and the order will be reset.

The `@Component` decorator's `styleUrls` property is used to specify one or more CSS stylesheets that should be applied to the content in the component's template. To provide validation feedback for the values that the user enters into the HTML form elements, I created a file called `checkout.component.css` in the `src/app/store` folder and defined the styles shown in Listing 8-25.

Listing 8-25. The Contents of the `checkout.component.css` File in the `src/app/store` Folder

```

input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
input.ng-dirty.ng-valid { border: 2px solid #6bc502 }

```

Angular adds elements to the `ng-dirty`, `ng-valid`, and `ng-invalid` classes to indicate their validation status. The full set of validation classes is described in Chapter 14, but the effect of the styles in Listing 8-25 is to add a green border around input elements that are valid and a red border around those that are invalid.

The final piece of the puzzle is the template for the component, which presents the user with the form fields required to populate the properties of an `Order` object, as shown in Listing 8-26.

Listing 8-26. The Contents of the `checkout.component.html` File in the `src/app/store` Folder

```

<div class="container-fluid">
  <div class="row">
    <div class="col bg-dark text-white">
      <a class="navbar-brand">SPORTS STORE</a>
    </div>
  </div>
</div>

<div *ngIf="orderSent" class="m-2 text-center">
  <h2>Thanks!</h2>
  <p>Thanks for placing your order.</p>
  <p>We'll ship your goods as soon as possible.</p>
  <button class="btn btn-primary" routerLink="/store">Return to Store</button>
</div>

<form *ngIf="!orderSent" #form="ngForm" novalidate
  (ngSubmit)="submitOrder(form)" class="m-2">
  <div class="form-group">
    <label>Name</label>
    <input class="form-control" #name="ngModel" name="name"
      [(ngModel)]="order.name" required />
  </div>

```

```

    <span *ngIf="submitted && name.invalid" class="text-danger">
      Please enter your name
    </span>
  </div>
  <div class="form-group">
    <label>Address</label>
    <input class="form-control" #address="ngModel" name="address"
      [(ngModel)]="order.address" required />
    <span *ngIf="submitted && address.invalid" class="text-danger">
      Please enter your address
    </span>
  </div>
  <div class="form-group">
    <label>City</label>
    <input class="form-control" #city="ngModel" name="city"
      [(ngModel)]="order.city" required />
    <span *ngIf="submitted && city.invalid" class="text-danger">
      Please enter your city
    </span>
  </div>
  <div class="form-group">
    <label>State</label>
    <input class="form-control" #state="ngModel" name="state"
      [(ngModel)]="order.state" required />
    <span *ngIf="submitted && state.invalid" class="text-danger">
      Please enter your state
    </span>
  </div>
  <div class="form-group">
    <label>Zip/Postal Code</label>
    <input class="form-control" #zip="ngModel" name="zip"
      [(ngModel)]="order.zip" required />
    <span *ngIf="submitted && zip.invalid" class="text-danger">
      Please enter your zip/postal code
    </span>
  </div>
  <div class="form-group">
    <label>Country</label>
    <input class="form-control" #country="ngModel" name="country"
      [(ngModel)]="order.country" required />
    <span *ngIf="submitted && country.invalid" class="text-danger">
      Please enter your country
    </span>
  </div>
  <div class="text-center">
    <button class="btn btn-secondary m-1" routerLink="/cart">Back</button>
    <button class="btn btn-primary m-1" type="submit">Complete Order</button>
  </div>
</form>

```


The form and input elements in this template use Angular features to ensure that the user provides values for each field, and they provide visual feedback if the user clicks the Complete Order button without completing the form. Part of this feedback comes from applying the styles that were defined in Listing 8-25, and part comes from span elements that remain hidden until the user tries to submit an invalid form.

■ **Tip** Requiring values is only one of the ways that Angular can validate form fields, and as I explained in Chapter 14, you can easily add your own custom validation as well.

To see the process, start with the list of products and click one of the Add To Cart buttons to add a product to the cart. Click the Checkout button and you will see the HTML form shown in Figure 8-6. Click the Complete Order button without entering text into any of the input elements, and you will see the validation feedback messages. Fill out the form and click the Complete Order button; you will see the confirmation message shown in the figure.

The figure consists of three overlapping browser window screenshots of the SportsStore checkout page at localhost:3000/checkout.

- Leftmost window:** Shows an empty form with red validation messages below each input field: "Please enter your name", "Please enter your address", "Please enter your city", "Please enter your state", "Please enter your zip/postal code", and "Please enter your country".
- Middle window:** Shows the form filled out with the following values: Name: Joe Smith, Address: 123 Main Street, City: Smallville, State: NY, Zip/Postal Code: 10036, and Country: USA. The "Complete Order" button is highlighted in blue.
- Rightmost window:** Shows a confirmation screen with the heading "Thanks!", the text "Thanks for placing your order. We'll ship your goods as soon as possible.", and a blue "Return to Store" button.

Figure 8-6. Completing an order

If you look at the browser's JavaScript console, you will see a JSON representation of the order like this:

```
{
  "cart": {
    "lines": [
      {
        "product": {
          "id": 1,
          "name": "Product 1",
          "category": "Category 1",
          "description": "Product 1 (Category 1)",
          "price": 100,
          "quantity": 1
        },
        "itemCount": 1,
        "cartPrice": 100
      },
      {
        "shipped": false,
        "name": "Joe Smith",
        "address": "123 Main Street",
        "city": "Smallville",
        "state": "NY",
        "zip": "10036",
        "country": "USA"
      }
    ]
  }
}
```

Using the RESTful Web Service

Now that the basic SportsStore functionality is in place, it is time to replace the dummy data source with one that gets its data from the RESTful web service that was created during the project setup in Chapter 7.

To create the data source, I added a file called `rest.datasource.ts` in the `src/app/model` folder and added the code shown in Listing 8-27.

Listing 8-27. The Contents of the `rest.datasource.ts` File in the `src/app/model` Folder

```
import { Injectable } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { Observable } from "rxjs";
import { Product } from "../product.model";
import { Cart } from "../cart.model";
import { Order } from "../order.model";

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

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

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

  getProducts(): Observable<Product[]> {
    return this.http.get<Product[]>(this.baseUrl + "products");
  }

  saveOrder(order: Order): Observable<Order> {
    return this.http.post<Order>(this.baseUrl + "orders", order);
  }
}
```

Angular provides a built-in service called `HttpClient` that is used to make HTTP requests. The `RestDataSource` constructor receives the `HttpClient` service and uses the global location object provided by the browser to determine the URL that the requests will be sent to, which is port 3500 on the same host that the application has been loaded from.

The methods defined by the `RestDataSource` class correspond to the ones defined by the static data source but are implemented using the `HttpClient` service, described in Chapter 24.

■ **Tip** When obtaining data via HTTP, it is possible that network congestion or server load will delay the request and leave the user looking at an application that has no data. In Chapter 27, I explain how to configure the routing system to prevent this problem.

Applying the Data Source

To complete this chapter, I am going to apply the RESTful data source by reconfiguring the application so that the switch from the dummy data to the REST data is done with changes to a single file. Listing 8-28 changes the behavior of the data source service in the model feature module.

Listing 8-28. Changing the Service Configuration in the `model.module.ts` File in the `src/app/model` Folder

```
import { NgModule } from "@angular/core";
import { ProductRepository } from "../product.repository";
import { StaticDataSource } from "../static.datasources";
import { Cart } from "../cart.model";
import { Order } from "../order.model";
import { OrderRepository } from "../order.repository";
import { RestDataSource } from "../rest.datasources";
import { HttpClientModule } from "@angular/common/http";

@NgModule({
  imports: [HttpClientModule],
  providers: [ProductRepository, Cart, Order, OrderRepository,
    { provide: StaticDataSource, useClass: RestDataSource }],
})
export class ModelModule { }
```

The `imports` property is used to declare a dependency on the `HttpClientModule` feature module, which provides the `HttpClient` service used in Listing 8-27. The change to the `providers` property tells Angular that when it needs to create an instance of a class with a `StaticDataSource` constructor parameter, it should use a `RestDataSource` instead. Since both objects define the same methods, the dynamic JavaScript type system means that the substitution is seamless. When all the changes have been saved and the browser reloads the application, you will see the dummy data has been replaced with the data obtained via HTTP, as shown in Figure 8-7.

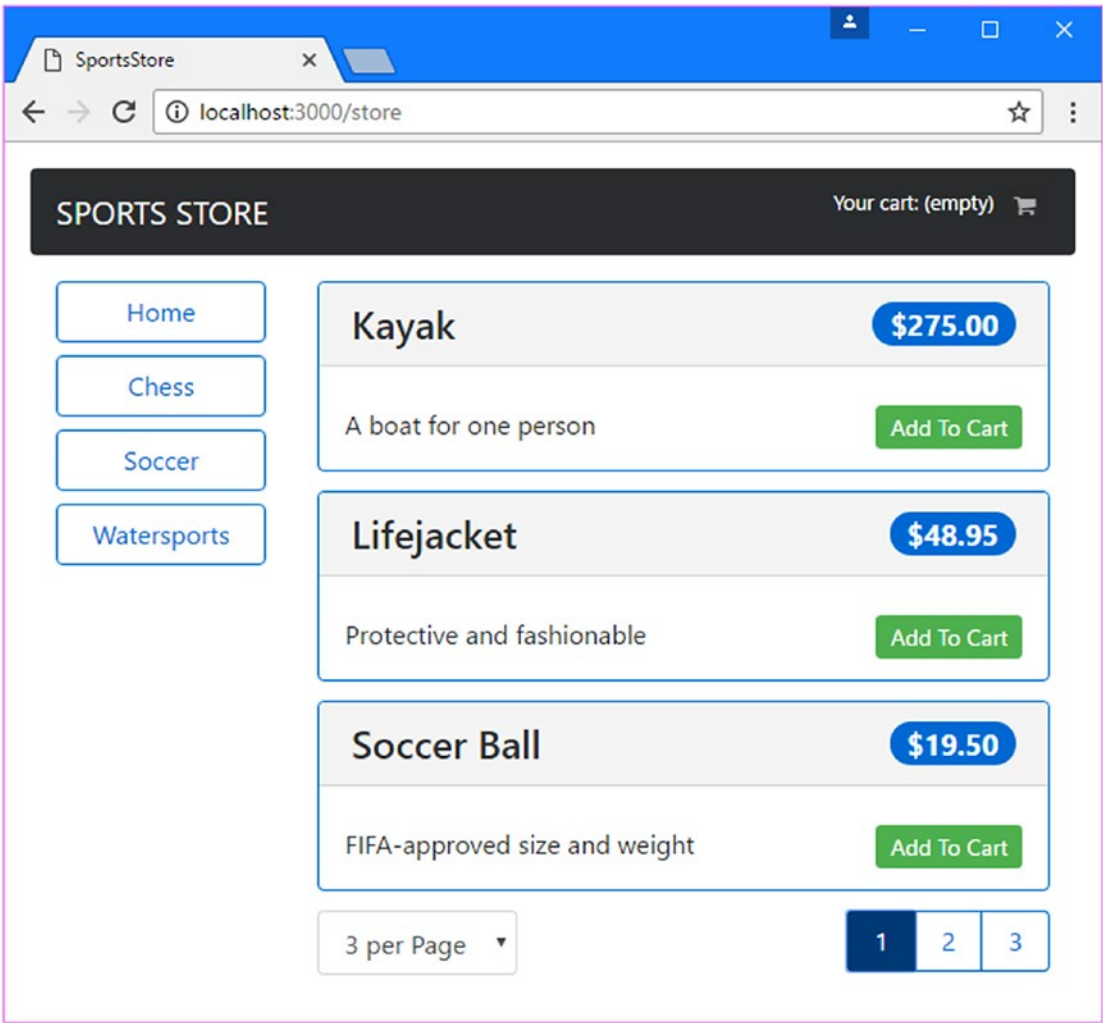


Figure 8-7. Using the RESTful web service

If you go through the process of selecting products and checking out, you can see that the data source has written the order to the web service by navigating to this URL:

`http://localhost:3500/db`

This will display the full contents of the database, including the collection of orders. You won't be able to request the `/orders` URL because it requires authentication, which I set up in the next chapter.

Tip Remember that the data provided by the RESTful web service is reset when you stop the server and start it again using the `npm run json` command.

Summary

In this chapter, I continued adding features to the SportsStore application, adding support for a shopping cart into which the user can place products and a checkout process that completes the shopping process. To complete the chapter, I replaced the dummy data source with one that sends HTTP requests to the RESTful web service. In the next chapter, I create the administration features that allow the SportsStore data to be managed.